Programação para Arquitectura

António Menezes Leitão

Setembro 2020

Conteúdo

Prefácio

Este livro nasceu em 2007, após um convite para leccionar uma disciplina introdutória de Programação aos alunos de Arquitectura do Instituto Superior Técnico (IST). A motivação original para a introdução da disciplina no curso de Arquitectura era a mesma que para muitos outros cursos: à semelhança da Matemática e da Física, a Programação tornou-se parte da formação básica de qualquer aluno do IST.

Com esta premissa, não parecia ser uma disciplina que viesse a despertar grande interesse nos alunos de Arquitectura, particularmente porque era pouco clara a contribuição que ela pudesse ter para o curso. Para contrariar essa impressão inicial, decidi incorporar no programa da disciplina algumas aplicações da Programação em Arquitectura. Nesse sentido, fui falar com alunos e docentes de Arquitectura, e pedi-lhes que me explicassem o que faziam e como o faziam. O que vi e ouvi foi revelador.

Apesar dos enormes progressos que as ferramentas de *Computer-Aided Design* (CAD) vieram trazer à profissão, a verdade é que a sua utilização continua manual, laboriosa, repetitiva, aborrecida. A elaboração de um modelo digital numa ferramenta de CAD implica uma enorme atenção ao pormenor, impedindo a concentração no fundamental: a ideia. Frequentemente, os obstáculos encontrados acabam por forçar o Arquitecto a ter de simplificar a ideia original. Infelizmente, esses obstáculos não terminam com a criação do modelo. Pelo contrário, agravam-se quando surge a inevitável necessidade de alterações ao modelo.

Em geral, as ferramentas de CAD são concebidas para facilitar a realização das tarefas mais comuns, em detrimento de outras menos frequentes ou mais sofisticadas. Na verdade, para o Arquitecto interessado em modelar formas mais complexas, a ferramenta de CAD poderá apresentar sérias limitações. E, contudo, essas limitações são apenas aparentes, pois é possível ultrapassá-las por intermédio da programação. A programação permite que uma ferramenta de CAD seja ampliada com novas capacidades, eliminando os obstáculos que impedem o trabalho do Arquitecto.

A actividade da programação é intelectualmente muito estimulante, mas é também um desafio. Implica dominar uma nova linguagem, implica adoptar uma nova forma de pensar. Frequentemente, esse esforço faz muitos desistirem, mas os que conseguem ultrapassar as dificuldades iniciais

ficam com a capacidade de ir mais longe na criação de soluções arquitectónicas inovadoras.

Este livro pretende ir ao encontro desses Arquitectos.

Capítulo 1

Programação

1.1 Introdução

A transmissão de conhecimento é um dos problemas que desde cedo preocupou a humanidade. Sendo o homem capaz de acumular conhecimento ao longo de toda a sua vida, é com desânimo que enfrenta a ideia de que, com a morte, todo esse conhecimento se perca.

Para evitar esta perda, a humanidade inventou toda uma série de mecanismos de transmissão de conhecimento. O primeiro, a transmissão oral, consiste na transmissão do conhecimento de uma pessoa para um grupo reduzido de outras pessoas, de certa forma transferindo o problema da perda de conhecimento para a geração seguinte. O segundo, a transmissão escrita, consiste em registar em documentos o conhecimento que se pretende transmitir. Esta forma tem a grande vantagem de, por um lado, poder chegar a muitas mais pessoas e, por outro, de reduzir significativamente o risco de se perder o conhecimento por problemas de transmissão. De facto, a palavra escrita permite preservar por muito tempo e sem qualquer tipo de adulteração o conhecimento que o autor pretendeu transmitir.

É graças à palavra escrita que hoje conseguimos compreender e acumular um vastíssimo conjunto de conhecimentos, muitos deles registados há milhares de anos atrás.

Infelizmente, nem sempre a palavra escrita conseguiu transmitir com rigor aquilo que o autor pretendia. A língua natural tem inúmeras ambiguidades e evolui substancialmente com o tempo, o que leva a que a interpretação dos textos seja sempre uma tarefa subjectiva. Quer quando escrevemos um texto, quer quando o lemos e o interpretamos, existem omissões, imprecisões, incorrecções e ambiguidades que podem tornar a transmissão de conhecimento falível. Se o conhecimento que se está a transmitir for simples, o receptor da informação, em geral, consegue ter a cultura e imaginação suficientes para conseguir ultrapassar os obstáculos. No caso da transmissão de conhecimentos mais complexos já isso poderá ser muito

mais difícil.

Quando se exige rigor na transmissão de conhecimento, fazer depender a compreensão desse conhecimento da capacidade de interpretação de quem o recebe pode ter consequências desastrosas e, de facto, a história da humanidade está repleta de acontecimentos catastróficos cuja causa é, tão somente, uma insuficiente ou errónea transmissão de conhecimento, ou uma deficiente compreensão do conhecimento transmitido.

Para evitar estes problemas, inventaram-se linguagens mais rigorosas. A matemática, em particular, tem-se obssessivamente preocupado ao longo dos últimos milénios com a construção de uma linguagem onde o rigor seja absoluto. Isto permite que a transmissão do conhecimento matemático seja muito mais rigorosa que nas outras áreas, reduzindo ao mínimo essencial a capacidade de imaginação necessária de quem está a absorver esse conhecimento.

Para melhor percebermos do que estamos a falar, consideremos um caso concreto de transmissão de conhecimento, por exemplo, o cálculo do factorial de um número. Se assumirmos, como ponto de partida, que a pessoa a quem queremos transmitir esse conhecimento já sabe de antemão o que são os números e as operações aritméticas, podemos dizer-lhe que *para calcular o factorial de um número qualquer, terá de multiplicar todos os números desde a unidade até esse número*. Infelizmente, esta descrição é demasiado extensa e, pior, é pouco rigorosa, pois não dá ao ouvinte a informação de que os números que ele tem de multiplicar são apenas os números inteiros. Para evitar estas imprecisões e, simultaneamente, tornar mais compacta a informação a transmitir, a Matemática inventou todo um conjunto de símbolos e conceitos cujo significado deve ser compreendido por todos. Por exemplo, para indicar a sequência de números inteiros entre 1 e 9, a Matemática permite-nos escrever $1, 2, 3, \ldots, 9$. Do mesmo modo, para evitarmos falar de "*um número qualquer*," a Matemática inventou o conceito de *variável*: um nome que designa qualquer "coisa" e que pode ser reutilizado em várias partes de uma afirmação matemática com o significado óbvio de representar sempre essa mesma "coisa." Deste modo, a linguagem Matemática permite-nos formular a mesma afirmação sobre o cálculo do factorial nos seguintes termos:

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

Será a definição anterior suficientemente rigorosa? Será possível interpretá-la sem necessitar de imaginar a intenção do autor? Aparentemente, sim mas, na verdade, há um detalhe da definição que *exige* imaginação: as reticências. Aquelas reticências indicam ao leitor que ele terá de imaginar o que deveria estar no lugar delas. Embora a maioria dos leitores imagine correctamente que o autor pretendia a multiplicação dos sucessores dos números anteriores, leitores haverá cuja imaginação delirante poderá levá-los a tentar substituir aquelas reticências por outra coisa qualquer.

Mesmo que excluamos do nosso público-alvo as pessoas de imaginação delirante, há ainda outros problemas com a definição anterior. Pensemos, por exemplo, no factorial de 2. Qual será o seu valor? Se substituirmos na fórmula, para $n = 2$ obtemos:

$$2! = 1 \times 2 \times 3 \times \cdots \times 2$$

Neste caso, o cálculo deixa de fazer sentido, o que mostra que, na verdade, a imaginação necessária para a interpretação da fórmula não se restringe apenas às reticências mas sim a toda a fórmula: o número de termos a considerar depende do número do qual queremos saber o factorial.

Admitindo que o nosso leitor teria tido a imaginação suficiente para descobrir esse detalhe, ele conseguiria tranquilamente calcular que $2! = 1 \times 2 = 2$. Ainda assim, casos haverá que o deixarão significativamente menos tranquilo. Por exemplo, qual é o factorial de zero? A resposta não parece óbvia. E quanto ao factorial de -1? Novamente, não está claro. E quanto ao factorial de 4.5? Mais uma vez, a fórmula nada diz e a nossa imaginação também não consegue adivinhar.

Será possível encontrar uma forma de transmitir o conhecimento da função factorial que minimize as imprecisões, lacunas e ambiguidades? Experimentemos a seguinte variante da definição da função factorial:

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n \cdot (n-1)!, & \text{se } n \in \mathbb{N}. \end{cases}$$

Teremos agora atingido o rigor suficiente que dispensa imaginação por parte do leitor? A resposta estará na análise dos casos que causaram problemas à definição anterior. Em primeiro lugar, não existem reticências, o que é positivo. Em segundo lugar, para o factorial de 2 temos, pela definição:

$$2! = 2 \times 1! = 2 \times (1 \times 0!) = 2 \times (1 \times 1) = 2 \times 1 = 2$$

ou seja, não há qualquer ambiguidade. Finalmente, vemos que não faz sentido determinar o factorial de -1 ou de 4.5 pois a definição apresentada apenas se aplica a um determinado conjunto de entidades, nomeadamente, aos membros de $\mathbb{N}_0$.

Este exemplo mostra que, mesmo na matemática, há diferentes graus de rigor nas diferentes formas como se expõe o conhecimento. Algumas dessas formas exigem um pouco mais de imaginação e outras um pouco menos mas, em geral, qualquer delas tem sido suficiente para que a humanidade tenha conseguido preservar o conhecimento adquirido ao longo da sua história.

Acontece que, actualmente, a humanidade conta com um parceiro que tem dado uma contribuição gigantesca para o seu progresso: o *computador*. Esta máquina tem a extraordinária capacidade de poder ser instruída

de forma a saber executar um conjunto complexo de tarefas. A actividade da *programação* consiste, precisamente, na transmissão, a um computador, do conhecimento necessário para resolver um determinado problema. Esse conhecimento é denominado um *programa*. Por serem *programáveis*, os computadores têm sido usados para os mais variados fins e, sobretudo nas últimas décadas, têm transformado radicalmente a forma como trabalhamos. Infelizmente, a extraordinária capacidade de "aprendizagem" dos computadores vem acompanhada duma igualmente extraordinária *incapacidade* de imaginação. O computador não imagina, apenas interpreta rigorosamente o conhecimento que lhe transmitimos sob a forma de um programa.

Uma vez que não tem imaginação, o computador depende criticamente do modo como lhe apresentamos o conhecimento que queremos transmitir: esse conhecimento tem de estar descrito numa linguagem tal que não possa haver margem para qualquer ambiguidade, lacuna ou imprecisão. Uma linguagem com estas características denomina-se, genericamente, de *linguagem de programação*.

1.1.1 Linguagens de Programação

Para que um computador possa resolver um problema é necessário que consigamos fazer uma descrição do processo de resolução desse problema numa linguagem que o computador entenda. Infelizmente, a linguagem que os computadores entendem de forma "inata" é extraordinariamente pobre, levando a que qualquer problema não trivial acabe por exigir uma exaustiva, entediante e extremamente complexa descrição do processo de resolução. As inúmeras linguagens de programação que têm sido inventadas visam precisamente aliviar o programador desse fardo, introduzindo elementos linguísticos capazes de simplificar enormemente essas descrições. Por exemplo, os conceitos de *função*, *matriz*, *somatório*, ou *número racional* não existem nativamente nos computadores, mas muitas linguagens de programação permitem a sua utilização de modo a facilitar a descrição de cálculos científicos. Isto implica, naturalmente, que tem de existir um processo capaz de transformar as descrições que o programador faz em descrições que o computador entenda. Embora este processo seja relativamente complexo, o que nos importa saber é que ele permite termos linguagens de programação mais próximas das capacidades do pensamento humano do que das capacidades de pensamento do computador.

Este último facto tem uma importância crucial pois permite que passemos a usar linguagens de programação, não só para instruir um computador sobre uma forma de resolver um problema, mas também para explicar rigorosamente esse processo a outros seres humanos. A linguagem de programação torna-se assim um meio de transmissão de conhecimento, tal como a linguagem da matemática o foi nos últimos milhares de anos.

Existe uma enorme diversidade de linguagens de programação, umas

mais apetrechadas para a resolução de um determinado tipo de problemas, outras mais apetrechadas para a resolução de outro. A escolha de uma linguagem de programação deve estar condicionada, naturalmente, pelo tipo de problemas que queremos resolver, mas não deve ser um comprometimento total. Para quem programa é muito mais importante compreender os fundamentos e técnicas da programação do que dominar esta ou aquela linguagem. No entanto, para mais rigorosamente se explicar aqueles fundamentos e técnicas, convém exemplificá-los numa linguagem de programação concreta.

Uma vez que este texto se irá focar na programação aplicada à Arquitectura, vamos empregar uma linguagem de programação que esteja particularmente vocacionada para resolver problems geométricos. Várias linguagens existem com esta vocação, em geral associadas a ferramentas de desenho assistido por computador–*Computer-Aided Design* (CAD). O ArchiCAD, por exemplo, disponibiliza uma linguagem de programação denominada GDL— acrónimo de *Geometric Description Language*—que permite ao utilizador programar as várias formas geométricas pretendidas. Já no caso do AutoCAD, a linguagem de programação empregue é o AutoLisp, um dialecto de uma famosa linguagem de programação denominada Lisp. Uma terceira hipótese será a linguagem RhinoScript, disponível no Rhinoceros. Apesar destas linguagens parecerem ser muito diferentes entre si, os conceitos subjacentes são muito semelhantes. É sobre estes conceitos fundamentais da programação que nos iremos debruçar, embora, por motivos pedagógicos, seja conveniente particularizá-los numa única linguagem.

Infelizmente, as linguagens GDL, AutoLisp e RhinoScript foram desenvolvidas há já bastante tempo e não foram actualizadas, possuindo várias características arcaicas que tornam mais difícil a sua aprendizagem e o seu uso. Para facilitar a aprendizagem e, simultaneamente, permitir que os programas que vamos desenvolver possam ser usados em diferentes ferramentas de CAD, vamos empregar uma nova linguagem, denominada Julia, que foi desenvolvida com propósitos pedagógicos e que foi adaptada propositadamente para a programação aplicada à Arquitectura. Assim, neste texto iremos explicar os fundamentos da programação através da utilização da linguagem Julia, não só pela sua facilidade de aprendizagem, mas também pela sua aplicabilidade prática. No entanto, uma vez apreendidos esses fundamentos, o leitor deverá ser capaz de os traduzir para qualquer outra linguagem de programação.

A linguagem Julia, bem como algumas ferramentas auxiliares, encontram-se disponíveis de forma gratuita em `http://julialang.org/`.

1.1.2 Exercícios

Exercício 1.1.1 A exponenciação b^n é uma operação entre dois números b e n designados base e expoente, respectivamente. Quando n é um inteiro

positivo, a exponenciação define-se como uma multiplicação repetida:

$$b^n = \underbrace{b \times b \times \cdots \times b}_{n}$$

Para um leitor que nunca tenha utilizado o operador de exponenciação, a definição anterior levanta várias questões cujas respostas poderão não lhe ser evidentes. Quantas multiplicações são realmente feitas? Serão n multiplicações? Serão $n - 1$ multiplicações? O que fazer no caso b^1? E no caso b^0?

Proponha uma definição matemática de exponenciação que não levante estas questões.

Exercício 1.1.2 O que é um programa? Para que serve?

Exercício 1.1.3 O que é uma linguagem de programação? Para que serve?

1.2 A Linguagem Julia

Nesta secção vamos descrever a linguagem de programação Julia que iremos usar ao longo deste texto. Antes, porém, vamos discutir alguns aspectos que são comuns a todas as linguagens.

1.2.1 Sintaxe, Semântica e Pragmática

Todas as linguagens possuem *sintaxe*, *semântica* e *pragmática*.

Em termos muito simples, podemos descrever a sintaxe de uma linguagem como o conjunto de regras que determinam as frases que se podem construir nessa linguagem. Sem sintaxe qualquer concatenação arbitrária de palavras constituiria uma frase. Por exemplo, dadas as palavras "João," "o," "comeu," e "bolo," as regras da sintaxe da língua Portuguesa dizem-nos que "o João comeu o bolo" é uma frase e que "o comeu João bolo bolo" *não* é uma frase. Note-se que, de acordo com a sintaxe do Português, "o bolo comeu o João" é também uma frase sintaticamente correcta.

A sintaxe regula a construção das frases mas nada diz acerca do seu significado. É a semântica de uma linguagem que nos permite atribuir significado às frases da linguagem e que nos permite perceber que a frase "o bolo comeu o João" não faz sentido.

Finalmente, a pragmática diz respeito à forma usual como se escrevem as frases da linguagem. Para uma mesma linguagem, a pragmática varia de contexto para contexto: a forma como dois amigos íntimos falam entre si é diferente da forma usada por duas pessoas que mal se conhecem.

Estes três aspectos das linguagens estão também presentes quando discutimos linguagens de programação. Contrariamente ao que acontece com as linguagens *naturais* que empregamos para comunicarmos uns com os outros, as linguagens de programação caracterizam-se por serem *formais*,

obedecendo a um conjunto de regras muito mais simples e rígido que permite que sejam analisadas e processadas *mecanicamente*.

Neste texto iremos descrever a sintaxe e semântica da linguagem Julia. Embora existam formalismos matemáticos que permitem descrever rigorosamente aqueles dois aspectos das linguagens, eles exigem uma sofisticação matemática que, neste trabalho, é desapropriada. Por este motivo, iremos apenas empregar descrições informais.

Quanto à pragmática, será explicada à medida que formos introduzindo os elementos da linguagem.

1.2.2 Sintaxe e Semântica do Julia

À semelhança de muitas outras linguagens de programação, a linguagem Julia possui uma sintaxe inspirada na matemática mas adaptada para ficar rigorosa. Um programa Julia é composto por *expressões*.

Uma expressão, em Julia, pode ser construída empregando expressões elementares como, por exemplo, os números, como `7` ou `123`; ou combinando outras expressões entre si como, por exemplo, na soma de dois números `7 + 123`. Como iremos ver, esta simples definição permite-nos construir expressões de complexidade arbitrária. No entanto, convém relembrar que a sintaxe restringe aquilo que podemos escrever: o facto de podermos combinar expressões para produzir outras expressões mais complexas não nos autoriza a escrever *qualquer* combinação de subexpressões. As combinações obedecem a regras sintáticas que iremos descrever ao longo do texto.

A semântica do Julia é determinada pelas *operações* que as nossas expressões empregam e, naturalmente, é também inspirada na semântica usual da matemática. A operação `+`, por exemplo, permite somar dois números. Uma combinação que junte este operador e, por exemplo, os números `3` e `4` tem, como significado, a soma de `3` com `4`, i.e., `7`.

1.2.3 O Avaliador

As implementações de Julia, em geral, providenciam um avaliador, i.e., um programa destinado a interagir com o utilizador de modo a avaliar as expressões por este fornecidas.

No caso do Julia, o avaliador surge assim que começamos a trabalhar no ambiente de programação, sendo possível alternar entre o editor e o avaliador a qualquer instante.

Assim, quando o utilizador começa a trabalhar com o Julia, é-lhe apresentado um sinal (denominado "*prompt*") e o Julia fica à espera que o utilizador escreva algo.

```
julia>
```

O texto ">" é a "prompt" do Julia, à frente da qual vão aparecer as expressões que o utilizador escrever. O Julia interage com o utilizador executando um ciclo em que lê um fragmento de um programa, determina o seu valor e escreve o resultado, se houver algum. Este ciclo de acções designa-se, tradicionalmente, de *read-eval-print-loop* (e abrevia-se para REPL).

Na fase de leitura, o Julia lê um texto. Na fase de avaliação, o texto lido é analisado empregando as regras da linguagem que determinam, para cada caso, o que fazer ou qual o valor resultante. Finalmente, se for produzido algum valor, é apresentado ao utilizador na fase de escrita através de uma representação textual desse valor.

A vantagem do *read-eval-print-loop* em Julia está na rapidez com que se desenvolvem protótipos de programas, escrevendo, testando e corrigindo pequenos fragmentos de cada vez.

1.3 Elementos da Linguagem

Em qualquer linguagem de programação precisamos de lidar com duas espécies de objectos: dados e procedimentos. Os dados são as entidades que pretendemos manipular. Os procedimentos são descrições das regras para manipular esses dados.

Se considerarmos a linguagem da matemática, podemos identificar os números como dados e as operações algébricas como procedimentos. As operações algébricas permitem-nos *combinar* os números entre si. Por exemplo, 2×2 é uma combinação. Uma outra combinação envolvendo mais dados será $2 \times 2 \times 2$ e, usando ainda mais dados, $2 \times 2 \times 2 \times 2$. No entanto, a menos que pretendamos ficar eternamente a resolver problemas de aritmética elementar, convém considerar operações mais elaboradas que representem padrões de cálculos. Na sequência de combinações que apresentámos é evidente que o padrão que está a emergir é o da operação de potenciação, i.e, multiplicação sucessiva, tendo esta operação sido definida na matemática há já muito tempo. A potenciação é, portanto, uma abstracção de uma sucessão de multiplicações.

Tal como a linguagem da matemática, uma linguagem de programação deve possuir dados e procedimentos primitivos, deve ser capaz de combinar quer os dados quer os procedimentos para produzir dados e procedimentos mais complexos e deve ser capaz de abstrair padrões de cálculo de modo a permitir tratá-los como operações simples, definindo novas operações que representem esses padrões de cálculo.

Mais à frente iremos ver como é possível definir essas abstracções em Julia. Por agora, vamos debruçar-nos sobre os dados primitivos da linguagem. Estes dados primitivos encontram-se divididos em diferentes *tipos*

de dados, sendo que cada tipo de dados emprega uma sintaxe específica para a descrição dos seus elementos e permite a sua combinação usando operações específicas.

1.3.1 Números

Um número é uma das entidades mais básicas da linguagem Julia. Em Julia, os números podem ser *exactos* ou *inexactos*. Os números exactos incluem os inteiros, como o `7`. Os números inexactos são tipicamente escritos em notação decimal ou científica, como por exemplo, `123.45` ou `1.2345e2`.

Em termos de semântica, um número é considerado um conceito fundamental, de tal forma que o significado de um número é o próprio número. É precisamente esse o resultado que vemos quando experimentamos os números na REPL do Julia:

```
julia> 1
1
julia> 12345
12345
julia> 4.5
4.5
```

1.3.2 Combinações

Uma combinação é uma expressão que descreve a aplicação de um operador aos seus operandos. Na matemática, os números podem ser combinados usando operações como a soma ou o produto. Como exemplo, temos $1 + 2$ e $1 + 2 \times 3$. A soma e o produto de números são operações elementares denominadas procedimentos primitivos. A sintaxe e semântica do Julia para as expressões aritméticas é muito semelhante à da Matemática, e segue as mesmas precedências entre operadores. Isto quer dizer que, em Julia, `1+2*3` é interpretado como $1 + (2 \times 3)$.

A semântica de uma combinação é obtida através da aplicação da operação (ou operações) aos seus operandos:

```
julia> 1+2
3
julia> 12345+54321
66666
julia> 12345*54321
670592745
julia> 1/2
0.5
julia> 1+2*3
7
julia> (1+2)*3
9
```

Exercício 1.3.1 O que é o REPL?

Exercício 1.3.2 Converta as seguintes expressões da notação do Julia para a notação da aritmética:

1. `1/2*3`
2. `1/(2 - 3)`
3. `(1 + 2)/3`
4. `1/2/3`
5. `1/(2/3)`

1.3.3 Avaliação de Combinações

O avaliador determina o valor de uma combinação como o resultado de aplicar o procedimento especificado pelo operador ao valor dos operandos. O valor de cada operando é designado de argumento do procedimento. Assim, o valor da combinação `1+2*3` é o resultado de somar o valor de `1` com o valor de `2*3`. Como já se viu, `1` vale 1 e `2*3` é uma combinação cujo valor é o resultado de multiplicar o valor de `2` pelo valor de `3`, o que dá 6. Finalmente, somando 1 a 6 obtemos 7.

```
julia> 2*3
6
julia> 1 + 2*3
7
```

Exercício 1.3.3 Calcule o valor das seguintes expressões Julia:

1. `1/2*3`
2. `1*(2 - 3)`
3. `(1 + 2)/3`
4. `1 - 2 - 3`

1.3.4 Cadeias de Caracteres

As *cadeias de caracteres* (também denominadas *strings*) são outro tipo de dados primitivo. Um carácter é uma letra, um dígito ou qualquer outro símbolo gráfico, incluindo os símbolos gráficos não visíveis como o espaço, a tabulação e outros. Uma cadeia de caracteres é especificada através de uma sequência de caracteres delimitada por aspas. Tal como com os números, o valor de uma cadeia de caracteres é a própria cadeia de caracteres.

Sequência	Resultado
`\\`	o carácter \ (*backslash*)
`\"`	o carácter " (aspas)
`\e`	o carácter *escape*
`\n`	o carácter de mudança de linha (*newline*)
`\r`	o carácter de mudança de linha (*carriage return*)
`\t`	o carácter de tabulação (*tab*)

Tabela 1.1: Alguns caracteres de *escape* válidos em Julia

```
julia> "Olá"
"Olá"
```

Para o caso de ser necessário incluir caracteres especiais numa *string*, como mudanças de linha ou tabulações, existe um carácter que o Julia interpreta de forma distinta: quando, numa *string*, surge o carácter `\`, ele assinala que o próximo carácter tem de ser processado de forma especial. O carácter `\` é denominado um carácter de *escape* e permite a inclusão em *strings* de caracteres que, de outra forma, seriam difíceis de inserir. A Tabela 1.1 mostra algumas possibilidades.

Tal como para os números, existem inúmeros operadores para *strings*. Por exemplo, para concatenar várias *strings*, existe o operador `*`. A concatenação de várias *strings* produz uma só *string* com todos os caracteres dessas várias *strings* e pela mesma ordem:

```
julia> "1" * "2"
"12"
julia> "um" * "dois" * "tres" * "quatro"
"umdoistresquatro"
julia> "eu" * " " * "sou" * " " * "uma" * " " * "string"
"eu sou uma string"
julia> "E eu" * " sou " * "outra"
"E eu sou outra"
```

Para saber o número de caracteres de uma dada *string* existe o operador `length`:

```
julia> length("eu sou uma string")
17
julia> length("")
0
```

Note-se que as aspas são os delimitadores de *strings* e não contam como caracteres.

Para além dos números e das *strings*, o Julia possui ainda outros tipos de dados que iremos explicar mais tarde.

1.4 Definição de Funções

Para além das operações básicas aritméticas, a matemática disponibiliza-nos um vastíssimo conjunto de outras operações que se definem à custa daquelas. Por exemplo, o *quadrado* de um número é uma operação (também designada por *função*) que, dado um número, produz o resultado de multiplicar esse número por ele próprio. Matematicamente falando, define-se o quadrado de um número pela função $x^2 = x \cdot x$.

Tal como em matemática, pode-se definir numa linguagem de programação a função que obtém o quadrado de um número. Em Julia, para obtermos o quadrado de um número, por exemplo, `5`, escrevemos a combinação `5*5`. No caso geral, dado um número qualquer `x`, sabemos que obtemos o seu quadrado escrevendo `x*x`. Falta apenas associar um nome que indique que, dado um número `x`, obtemos o seu quadrado avaliando `x*x`. Julia permite-nos fazer isso através da definição de funções:

```
quadrado(x) =
  x*x
```

Como se pode ver pela definição da função `quadrado`, para se definirem funções em Julia é necessário indicar o nome da função, os parâmetros da função e, finalmente, a expressão que determina o valor da função para aqueles parâmetros. De modo genérico podemos indicar que a definição de funções é feita usando a seguinte forma:

```
nome(parâmetro_1, ..., parâmetro_n) =
  corpo
```

Os parâmetros de uma função são designados *parâmetros formais* e são os nomes usados no corpo para nos referirmos aos argumentos correspondentes. Quando escrevemos no avaliador `quadrado(5)`, `5` é o *argumento* da função. Durante o cálculo da função este argumento está associado ao parâmetro formal `x`. Os argumentos de uma função são também designados por *parâmetros actuais*.

No caso da função `quadrado`, a sua definição diz que para se determinar o quadrado de um número `x` devemos multiplicar esse número por ele próprio `x*x`. Esta definição associa a palavra `quadrado` a um procedimento, i.e., a uma descrição do modo de produzir o resultado pretendido. Note-se que este procedimento possui parâmetros, permitindo o seu uso com diferentes argumentos. A título de exemplo, podemos avaliar as seguintes expressões:

```
julia> quadrado(5)
25
julia> quadrado(6)
36
```

A regra de avaliação de combinações que descrevemos anteriormente é também válida para as funções por nós definidas. Assim, a avaliação da

expressão `quadrado(1 + 2)` passa pela avaliação do operando `1 + 2`. Este operando tem como valor 3, valor esse que é usado pela função no lugar do parâmetro `x`. O corpo da função é então avaliado mas substituindo todas as ocorrências de `x` pelo valor `3`, i.e., o valor final será o da combinação `3*3`.

Em termos formais, para se invocar uma função, é necessário construir uma combinação cujo primeiro elemento seja uma expressão que avalia para a função que se pretende invocar e cujos restantes elementos são expressões que avaliam para os argumentos que se pretende passar à função. O resultado da avaliação da combinação é o valor calculado pela função para aqueles argumentos.

A avaliação de uma combinação deste género processa-se nos seguintes passos:

1. Todos os elementos da combinação são avaliados, sendo que o valor do primeiro elemento é necessariamente uma função.

2. Associam-se os parâmetros formais dessa função aos argumentos, i.e., aos valores dos restantes elementos da combinação. Cada parâmetro é associado a um argumento, de acordo com a ordem dos parâmetros e argumentos. É gerado um erro sempre que o número de parâmetros não é igual ao número de argumentos.

3. Avalia-se o corpo da função tendo em conta estas associações entre os parâmetros e os argumentos.

Para se perceber este processo de avaliação, é útil decompô-lo nas suas etapas mais elementares. No seguinte exemplo mostramos o processo de avaliação para $((1+2)^2)^2$:

```
quadrado(quadrado(1 + 2))
            ↓
 quadrado(quadrado(3))
            ↓
     quadrado(3*3)
            ↓
      quadrado(9)
            ↓
          9*9
            ↓
          81
```

Todas as funções por nós definidas são consideradas pelo avaliador de Julia em pé de igualdade com todas as outras definições. Isto permite que elas possam ser usadas para definir ainda outras funções. Por exemplo, após termos definido a função `quadrado`, podemos definir a função que calcula a área de um círculo de raio r através da fórmula $\pi \cdot r^2$.

```
area_circulo(raio) =
  3.14159*quadrado(raio)
```

Naturalmente, durante a avaliação de uma expressão destinada a computar a àrea de um círculo, a função `quadrado` acabará por ser invocada. Isso é visível na seguinte sequência de passos de avaliação:

```
area_circulo(2)
      ↓
3.14159*quadrado(2)
      ↓
  3.14159*(2*2)
      ↓
    3.14159*4
      ↓
   12.5664
```

Uma vez que a definição de funções permite-nos associar um procedimento a um nome, isto implica que o Julia tem de possuir uma memória onde possa guardar esta associação. Esta memória do Julia designa-se *ambiente*.

Note-se que este ambiente apenas existe enquanto estamos a trabalhar com a linguagem. Quando terminamos, perde-se todo o ambiente. Para evitar perdermos as definições que tenhamos feito, convém registá-las num suporte persistente, como seja um ficheiro. Por este motivo, o processo usual de trabalho com Julia consiste em escrever as várias definições em ficheiros, embora se continue a usar o avaliador de Julia para experimentar e testar o correcto funcionamento das nossas definições.

Exercício 1.4.1 Defina a função `dobro` que, dado um número, calcula o seu dobro.

1.4.1 Nomes

A definição de funções em Julia passa pela utilização de nomes: nomes para as funções e nomes para os parâmetros das funções.

Em Julia, existem limitações para a escrita de nomes. Estes só podem ser compostos de letras, dígitos, e pelo carácter _, e não podem começar por um dígito. Na prática, a criação de nomes segue algumas regras que convém ter presentes:

- Deve-se evitar a utilização de dígitos.
- Por motivos de portabilidade, convém evitar o uso de caracteres acentuados.

- Se o nome da função é composto por várias palavras, deve-se separar as palavras com traços (_). Por exemplo, uma função que calcula a área de um círculo poderá ter como nome `area_circulo`.

A escolha de nomes apropriados pode ter um impacto significativo na legibilidade de um programa. Consideremos, por exemplo, a área A de um triângulo de base b e altura c que se define matematicamente por

$$A(b, c) = \frac{b \cdot c}{2}$$

Em Julia, teremos:

```
A(b, c) =
  b*c/2
```

Como se pode ver, a definição da função em Julia é idêntica à definição correspondente em Matemática. No entanto, se não soubermos de antemão para que serve aquela função, dificilmente a iremos compreender. Assim, e contrariamente ao que é usual ocorrer em Matemática, os nomes que empregamos em Julia devem ter um significado claro. Assim, ao invés de se escrever `A` é preferível escrever `area_triangulo`. Do mesmo modo, ao invés de se escrever `b` e `c`, é preferível escrever `base` e `altura`. Tendo estes aspectos em conta, podemos apresentar uma definição mais legível:

```
area_triangulo(base, altura) =
  base*altura/2
```

Quando o número de definições aumenta, torna-se particularmente importante para quem as lê que se perceba rapidamente o seu significado e, por isso, é de crucial importância que se faça uma boa escolha de nomes.

Exercício 1.4.2 Indique um nome apropriado para cada uma das seguintes funções:

1. Função que calcula o volume de uma esfera.
2. Função que indica se um número é primo.
3. Função que converte uma medida em centímetros para polegadas.

Exercício 1.4.3 Defina a função `radianos` que recebe uma quantidade angular em graus e calcula o valor correspondente em radianos. Note que 180 graus correspondem a π radianos.

Exercício 1.4.4 Defina a função `graus` que recebe uma quantidade angular em radianos e calcula o valor correspondente em graus.

Exercício 1.4.5 Defina a função que calcula o perímetro de uma circunferência de raio r.

Exercício 1.4.6 Defina uma função que calcula o volume de um paralelipípedo a partir do seu comprimento, altura e largura. Empregue nomes suficientemente claros.
Exercício 1.4.7 Defina a função que calcula o volume de um cilindro com um determinado raio e comprimento. Esse volume corresponde ao produto da área da base pelo comprimento do cilindro.
Exercício 1.4.8 Defina a função `media` que calcula o valor médio entre dois outros valores. Por exemplo `media(2, 3)`→`2.5`.

1.5 Funções Pré-Definidas

A possibilidade de se definirem novas funções é fundamental para aumentarmos a flexibilidade da linguagem e a sua capacidade de se adaptar aos problemas que pretendemos resolver. As novas funções, contudo, precisam de ser definidas à custa de outras que, ou foram também por nós definidas ou, no limite, já estavam pré-definidas na linguagem.

Como iremos ver, o Julia providencia um conjunto razoavelmente grande de funções pré-definidas. Em muitos casos, são suficientes para o que pretendemos, mas não nos devemos coibir de definir novas funções sempre que acharmos necessário.

A Tabela 1.2 apresenta uma selecção de funções matemáticas pré-definidas do Julia. Note-se que, devido às limitações sintáticas do Julia (e que são comuns a todas as outras linguagens de programação), há vários casos em que uma função em Julia emprega uma notação diferente daquela que é usual em matemática. Por exemplo, a função raiz quadrada $\sqrt{x}$ escreve-se como `sqrt(x)`. O nome `sqrt` é uma contracção das palavras *square root* e contracções semelhantes são empregues para várias outras funções. Por exemplo, a função valor absoluto $|x|$ escreve-se `abs(x)` (de *absolute value*) e a função potência x^y escreve-se `x^y`.
Exercício 1.5.1 Traduza as seguintes expressões matemáticas para Julia:

1. $\sqrt{\frac{1}{\log 2^{|(3-9\log 25)|}}}$
2. $\frac{\cos^4 \frac{2}{\sqrt{5}}}{\operatorname{atan} 3}$
3. $\frac{1}{2} + \sqrt{3} + \sin^{\frac{5}{2}} 2$

Exercício 1.5.2 Traduza as seguintes expressões Julia para a notação matemática:

1. `log(sin(2^4 + floor(atan(pi))/sqrt(5)))`
2. `cos(cos(cos(0.5)))^5`

Função	Argumentos	Resultado
`abs`	Um número	O valor absoluto do argumento.
`sin`	Um número	O seno do argumento (em radianos).
`cos`	Um número	O cosseno do argumento (em radianos).
`atan`	Um número	O arco tangente do argumento (em radianos).
`atan`	Dois números	Com dois argumentos, o arco tangente da divisão do primeiro pelo segundo (em radianos). O sinal dos argumentos é usado para determinar o quadrante.
`sqrt`	Um número	A raiz quadrada do argumento.
`exp`	Um número	A exponencial de base e.
`^`	Dois números	O primeiro argumento elevado ao segundo argumento.
`log`	Um número	O logaritmo natural do argumento.
`max`	Vários números	O maior dos argumentos.
`min`	Vários números	O menor dos argumentos.
`floor`	Um número	O argumento arredondado para baixo.
`ceil`	Um número	O argumento arredondado para cima.

Tabela 1.2: Algumas funções matemáticas pré-definidas no Julia.

3. `sin(cos(sin(pi/3)/3)/3)`

Exercício 1.5.3 Defina o predicado `impar` que, dado um número, testa se ele é ímpar, i.e., se o resto da divisão desse número por dois é um. Para calcular o resto da divisão de um número por outro, utilize a operação pré-definida `%`.

Exercício 1.5.4 A área A de um pentágono regular inscrito num círculo de raio r é dada pela fórmula

$$A = \frac{5}{8} r^2 \sqrt{10 + 2\sqrt{5}}$$

Defina uma função em Julia que calcule essa área. Teste-a na interacção do IDLE para valores à sua escolha.

Exercício 1.5.5 Defina uma função que calcula o volume de um elipsóide de semi-eixos a, b e c. Esse volume pode ser obtido pela fórmula $V = \frac{4}{3}\pi abc$.

1.6 Aritmética em Julia

Vimos anteriormente que o Julia é capaz de lidar com variados tipos de números, desde os inteiros aos complexos, passando pelas fracções. Alguns

destes números, como, por exemplo, $\sqrt{2}$ ou π, não possuem uma representação rigorosa baseada em numerais e, por este motivo, o Julia classifica-os como *inexactos*, para tornar claro que estamos apenas a lidar com uma aproximação. Quando um número inexacto participa numa operação aritmética, o resultado é também inexacto, dizendo-se, por isso, que a inexactidão é *contagiosa*.

A *finitude* é uma outra característica dos números inexactos. Contrariamente aos números exactos que, teoricamente, não são limitados, os números inexactos não podem ultrapassar um determinado limite, a partir do qual todos os números são representados pela infinidade, tal como se pode ver na seguinte interacção:

```
julia> 10.0^10
1.0e10
julia> 10.0^100
1e+100
julia> 10.0^1000
Inf
```

Note-se que o Julia não consegue representar números inexactos superiores a um determinado limite (`1.7976931348623157e+308`), considerando esses números como infinidades.

```
julia> Inf + 1
Inf
julia> Inf - 1
Inf
julia> -Inf - 1
-Inf
```

Há ainda outro problema importante relacionado com os números inexactos: erros de arredondamento. A título de exemplo, consideremos a óbvia igualdade matemática $(\frac{4}{3} - 1) \cdot 3 - 1 = 0$ e comparemos os resultados que se obtêm usando números inexactos:

```
julia> (4.0/3.0 - 1.0)*3.0 - 1.0
-2.220446049250313e-16
```

Como se pode ver, usando números inexactos, não se consegue obter o resultado correcto, sendo o problema causado por erros de arredondamento: $4/3$ não é representável com um número finito de dígitos. Este erro de arredondamento é então propagado às restantes operações, produzindo um valor que, embora não seja zero, está relativamente próximo.

Exercício 1.6.1 Traduza a seguinte definição para Julia:

$$f(x) = x - 0.1 \cdot (10 \cdot x - 10)$$

Exercício 1.6.2 Matematicamente falando, qualquer que seja o argumento usado na função anterior, o resultado deveria ser sempre 1 pois

$$f(x) = x - 0.1 \cdot (10 \cdot x - 10) = x - (x - 1) = 1$$

Usando a função definida na questão anterior, calcule em Julia o valor das seguintes expressões e explique os resultados:

```
f(5.1)
f(51367.7)
f(176498634.7)
f(1209983553611.9)
f(19843566622234755.9)
f(553774558711019983333.9)
```

Exercício 1.6.3 Pretende-se criar um lanço de escada com n espelhos capaz de vencer uma determinada altura a em metros. Admitindo que cada degrau tem uma altura do espelho h e uma largura do cobertor d que verificam a proporção

$$2h + d = 0.64$$

defina uma função que, a partir da altura a vencer e do número de espelhos, calcula o comprimento do lanço de escada.

1.7 Avaliação de Nomes

Vimos que todos os dados primitivos que apresentámos até agora, nomeadamente, os números e as cadeias de caracteres, avaliavam para eles próprios, i.e., o valor de uma expressão constituída apenas por um dado primitivo é o próprio dado primitivo. No caso dos nomes, isso já não é verdade.

Julia atribui um significado muito especial aos nomes. Reparemos que, quando definimos uma função, esta possui um nome. Os parâmetros formais da função também são nomes. Quando escrevemos uma combinação, o avaliador de Julia usa a definição de função que foi associada ao nome que constitui o primeiro elemento da combinação. Isto quer dizer que o valor do primeiro nome de uma combinação é a função que lhe está associada. Admitindo que tínhamos definido a função `quadrado` da forma que ilustrámos na secção 1.4, podemos verificar este comportamento experimentando as seguintes expressões:

```
julia> quadrado(3)
9
julia> quadrado
quadrado (generic function with 1 method)
```

Como se pode ver pelo exemplo anterior, o valor do nome `quadrado` é uma entidade que o Julia descreve usando uma notação especial. A entidade em questão é, como vimos, uma função. O mesmo comportamento ocorre para qualquer outra função pré-definida na linguagem:

```
julia> ^
^ (generic function with 63 methods)
julia> abs
abs (generic function with 13 methods)
```

Como vimos com o `^` e o `abs`, alguns dos nomes estão pré-definidos na linguagem. No entanto, quando o avaliador está a avaliar o corpo de uma função, o valor de um nome especificado nos parâmetros da função é o argumento correspondente na invocação da função. Assim, na combinação `quadrado(3)`, depois de o avaliador saber que o valor de `quadrado` é a função por nós definida e que o valor de `3` é 3, o avaliador passa a avaliar o corpo da função `quadrado` mas assumindo que, durante essa avaliação, o nome `x`, sempre que for necessário, irá ter como valor precisamente o mesmo 3 que foi associado ao parâmetro `x`.

1.8 Expressões Condicionais

Existem muitas operações cujo resultado depende da realização de um determinado teste. Por exemplo, a função matemática $|x|$, que calcula o valor absoluto do número x, equivale ao simétrico do número, se este for negativo, ou ao próprio número, caso contrário. Usando a moderna notação da matemática, temos:

$$|x| = \begin{cases} -x, & \text{se } x < 0 \\ x, & \text{caso contrário.} \end{cases}$$

Esta função terá, portanto, de *testar* se o seu argumento é negativo e escolher uma de duas alternativas: ou avalia para o próprio argumento, ou avalia para o seu simétrico.

Estas expressões, cujo valor depende de um ou mais testes a realizar previamente, são designadas *expressões condicionais*.

1.8.1 Expressões Lógicas

Uma expressão condicional assume a forma de "se *expressão* então ... caso contrário ...." A *expressão*, cujo valor é usado para decidir se devemos usar o ramo "então" ou o ramo "caso contrário," denomina-se *expressão lógica* e caracteriza-se por o seu valor ser interpretado como *verdade* ou *falso*. Por exemplo, a expressão lógica `x < 0` testa se o valor de `x` é menor que zero. Se for, a expressão avalia para verdade, caso contrário avalia para falso.

1.8.2 Valores Lógicos

Em Julia, as expressões condicionais consideram como verdade o valor `true` e como falso o valor `false`. Estes valores são os únicos elementos

de um tipo especial de dados denominado *lógico* ou *Booleano*.[1]

1.9 Predicados

No caso mais usual, uma expressão lógica envolve uma operação com determinados argumentos. Nesta situação, a operação é denominada *predicado* e, quando aplicada aos seus operandos, produz um valor que é interpretado como sendo verdadeiro ou falso. O predicado é, consequentemente, uma operação que produz verdade ou falso.

1.9.1 Predicados Aritméticos

Os *operadores relacionais* matemáticos $<$, $>$, $=$, $\leq$, $\geq$ e $\neq$ são um dos exemplos mais simples de predicados. Estes operadores comparam números entre si. No caso da linguagem Julia, escrevem-se, respectivamente, `<`, `>`, `==`, `<=`, `>=` e `!=`. Eis alguns exemplos:

```
julia> 4 > 3
true
julia> 4 < 3
false
julia> 2 + 3 <= 6 - 1
true
```

1.10 Operadores Lógicos

Para se poder combinar expressões lógicas entre si existem os operadores `&&` (conjunção), `||` (disjunção) e `!` (negação). O valor das expressões que empregam estes operadores lógicos é determinado do seguinte modo:

- O `&&` avalia os seus argumentos da esquerda para a direita até que um deles seja falso, devolvendo este valor. Se nenhum for falso, o `&&` devolve verdade.
- O `||` avalia os seus argumentos da esquerda para a direita até que um deles seja verdade, devolvendo este valor. Se nenhum for verdade, o `||` devolve falso.
- O `!` avalia para verdade se o seu argumento for falso e para falso em caso contrário.

Exercício 1.10.1 Qual o valor das seguintes expressões?

1. `(2 > 3 || !(2 == 3)) && 2 < 3`

[1] De George Boole, matemático inglês e inventor da álgebra da verdade e da falsidade.

2. `! (1 == 2 || 2 == 3)`

3. `1 < 2 || 1 == 2 || 1 > 2`

Exercício 1.10.2 O que é uma expressão condicional? O que é uma expressão lógica?
Exercício 1.10.3 O que é um valor lógico? Quais são os valores lógicos empregues em Julia?
Exercício 1.10.4 O que é um predicado? Dê exemplos de predicados em Julia.
Exercício 1.10.5 O que é um operador relacional? Dê exemplos de operadores relacionais em Julia.
Exercício 1.10.6 O que é um operador lógico? Quais são os operadores lógicos que conhece em Julia?
Exercício 1.10.7 Traduza para Julia as seguintes expressões matemáticas:

1. $x < y$
2. $x \leq y$
3. $x < y \wedge y < z$
4. $x < y \wedge x < z$
5. $x \leq y \leq z$
6. $x \leq y < z$
7. $x < y \leq z$

1.11 Selecção

Se observarmos a definição matemática da função valor absoluto

$$|x| = \begin{cases} -x, & \text{se } x < 0 \\ x, & \text{caso contrário.} \end{cases}$$

constatamos que ela emprega uma expressão condicional da forma

$$\begin{cases} \textit{expressão consequente}, & \text{se } \textit{expressão lógica} \\ \textit{expressão alternativa}, & \text{caso contrário.} \end{cases}$$

que, em linguagem natural, se traduz para "se *expressão lógica*, então *expressão consequente*, caso contrário, *expressão alternativa*."

A avaliação de uma expressão condicional é feita através da avaliação da *expressão lógica* que, se produzir verdade, implica a avaliação da *expressão consequente* e, se produzir falso, implica a avaliação da *expressão alternativa*.

No caso da linguagem Julia, existem duas notações possíveis para descrever expressões condicionais, sendo a segunda bastante mais compacta que a primeira. A primeira notação emprega as palavras `if`, `else` e `end` para delimitar e separar a expressão condicional. Nesta notação, a função valor absoluto poderá ser definida como:

```
abs(x) =
  if x < 0
    -x
  else
    x
  end
```

A segunda notação emprega limita-se a separar as partes da expressão condicional com os símbolos `?` e `:`. No caso da segunda notação, a função poderá ser definida como:

```
abs(x) =
  x < 0 ? -x : x
```

O valor de uma expressão condicional é obtido da seguinte forma:

1. A *expressão lógica* é avaliada.
2. Se o valor obtido da avaliação anterior é verdade, o valor da expressão condicional é o valor da *expressão consequente*.
3. Caso contrário (ou seja, o valor obtido da avaliação anterior é falso), o valor da expressão condicional é o valor da *expressão alternativa*.

O objectivo fundamental de uma expressão condicional é permitir definir funções cujo comportamento depende de uma ou mais condições. Por exemplo, consideremos a função `max` que recebe dois números como argumentos e devolve o maior deles. Para definirmos esta função apenas precisamos de testar se o primeiro argumento é maior que o segundo. Se for, a função devolve o primeiro argumento, caso contrário devolve o segundo. Com base neste raciocínio, podemos escrever:

```
max(x, y) =
  if x > y
    x
  else
    y
  end
```

Ou, alternativamente:

```
max(x, y) = x > y ? x : y
```

Um outro exemplo mais interessante ocorre com a função matemática *signum*, abreviadamente sgn, que calcula o sinal de um número. Esta função pode ser vista como a função *dual* da função valor absoluto pois tem-se sempre $x = \operatorname{sgn}(x)|x|$. A função sinal é definida por:

$$\operatorname{sgn} x = \begin{cases} -1 & \text{se } x < 0 \\ 0 & \text{se } x = 0 \\ 1 & \text{caso contrário} \end{cases}$$

Em linguagem natural, dizemos que se x for negativo, o valor de $\operatorname{sgn} x$ é -1, caso contrário, se x for 0, o valor é 0, caso contrário, o valor é 1. Isto mostra que, na verdade, a definição anterior emprega duas expressões condicionais encadeadas, da forma:

$$\operatorname{sgn} x = \begin{cases} -1 & \text{se } x < 0 \\ \begin{cases} 0 & \text{se } x = 0 \\ 1 & \text{caso contrário} \end{cases} & \text{caso contrário} \end{cases}$$

Assim sendo, para definirmos esta função em Julia, temos de empregar duas expressões condicionais. Uma possibilidade será:

```
signum(x) = x < 0 ? -1 : x == 0 ? 0 : 1
```

Neste caso, no entanto, a expressão fica excessivamente compacta, tornando a leitura difícil. É possível ajudar o leitor através do uso de parêntesis:

```
signum(x) = x < 0 ? -1 : (x == 0 ? 0 : 1)
```

mas uma alternativa mais legível será empregar a forma `if-else`:

```
signum(x) =
  if x < 0
    -1
  else
    if x == 0
      0
    else
      1
    end
  end
```

1.12 Selecção Múltipla

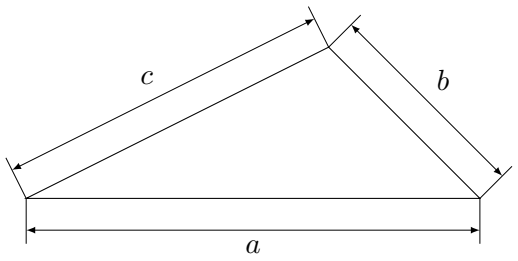
Quando uma expressão condicional necessita de vários `if`s encadeados, é possível que o programa comece a ficar mais difícil de ler. Neste caso, o Julia permite combinar os `if`s encadeados através do símbolo `elseif`. Explorando esta possibilidade, a função *sinal* pode ser escrita de forma mais simples:

```
signum(x) =
  if x < 0
    -1
  elseif x == 0
    0
  else
    1
  end
```

Exercício 1.12.1 Defina uma função `soma_maiores` que recebe três números como argumento e determina a soma dos dois maiores.
Exercício 1.12.2 Defina a função `max3` que recebe três números como argumento e calcula o maior entre eles.
Exercício 1.12.3 Defina a função `segundo_maior` que recebe três números como argumento e devolve o segundo maior número, i.e., que está entre o maior e o menor.

1.13 Nomes Locais

Consideremos o seguinte triângulo



e tentemos definir uma função Julia que calcula a área do triângulo a partir dos parâmetros a, b e c.

Uma das formas de calcular a área do triângulo baseia-se na famosa fórmula de Heron:[2]

[2]Heron de Alexandria foi um importante matemático e engenheiro Grego do primeiro século depois de Cristo a quem se atribuem inúmeras descobertas e invenções, incluindo a máquina a vapor e a seringa.

$$A = \sqrt{s\,(s-a)\,(s-b)\,(s-c)}$$

em que é s o semi-perímetro do triângulo:

$$s = \frac{a+b+c}{2}$$

Ao tentarmos usar a fórmula de Heron para definir a correspondente função Julia, somos confrontados com uma dificuldade: a fórmula está escrita (também) em termos do semi-perímetro s, mas s não é um parâmetro, é um valor *derivado* dos parâmetros do triângulo.

Uma das maneira de ultrapassarmos a dificuldade consiste em eliminar o nome s, substituindo-o pelo seu significado:

$$A = \sqrt{\frac{a+b+c}{2}\left(\frac{a+b+c}{2}-a\right)\left(\frac{a+b+c}{2}-b\right)\left(\frac{a+b+c}{2}-c\right)}$$

A partir desta fórmula já é possível definir a função pretendida:

```
area_triangulo(a, b, c) =
  sqrt((a + b + c)/2*
       ((a + b + c)/2 - a)*
       ((a + b + c)/2 - b)*
       ((a + b + c)/2 - c))
```

Infelizmente, esta definição tem dois problemas. O primeiro é que se perdeu a correspondência entre a fórmula original e a definição da função, tornando mais difícil reconhecer que a função está a implementar a fórmula de Heron. O segundo problema é que a função é obrigada a repetidamente empregar a expressão `(a + b + c)/2`, o que não só representa um desperdício do nosso esforço, pois tivemos de a escrever quatro vezes, como representa um desperdício de cálculo, pois a expressão é calculada quatro vezes, embora saibamos que ela produz sempre o mesmo resultado.

Para resolver estes problemas, o Julia permite a utilização de *nomes locais*. Um nome local é um nome que apenas tem significado no contexto de uma dada função e que é usado para calcular valores intermédios como o do semi-perímetro s. Os nomes locais são introduzidos pela forma `let`, que delimita o contexto desses nomes com um `end` no final. Empregando um nome local, podemos reescrever a função `area_triangulo` da seguinte forma:

```
area_triangulo(a, b, c) =
  let s = (a + b + c)/2
    sqrt(s*(s - a)*(s - b)*(s - c))
  end
```

A semântica da definição de um nome local é idêntica à das definições usuais, com a subtileza de o *âmbito* da definição, i.e., a região do programa

onde o nome definido pode ser usado, estar compreendida ao corpo da forma `let`, i.e., termina quando se atinge o `end`.

Ao invocarmos a função `area_triangulo`, passando-lhe os argumentos correspondentes aos parâmetros `a`, `b` e `c`, ela começa por introduzir um novo nome, `s`, associado ao valor que resultar da avaliação da expressão `(a + b + c)/2` e, no contexto desse novo nome, avalia as restantes expressões do corpo da função. Na prática, é como se a função dissesse: "Sendo $s = \frac{a+b+c}{2}$, calculemos $\sqrt{s\,(s-a)\,(s-b)\,(s-c)}$."

1.14 Nomes Globais

Contrariamente aos nomes locais, que possuem um âmbito de referência limitado, um *nome global* é um nome que é visivel de todos os pontos de um programa. O seu âmbito é, portanto, todo o programa. O nome `pi`, por exemplo, poderá representar a constante $\pi = 3.14159$ e, por isso, poderá ser usado em qualquer ponto dos nossos programas. Por esse motivo, o nome `pi` designa um nome global.

A definição de nomes globais é igual à dos nomes locais, com a diferença de serem feitos fora de qualquer função. No caso do nome `pi` poderemos fazer:

```
pi = 3.14159
```

A partir desse momento, o nome `pi` pode ser referenciado em qualquer ponto do programa.

É importante referir que o uso de nomes globais deve ser restrito, na medida do possível, à definição de *constantes*, tal como fizemos no caso do `pi`.

1.15 Módulos

Em Julia, é possível agrupar as definições em *módulos*. Cada módulo é uma unidade contendo um conjunto de definições, que podem ser usadas depois de pedirmos ao Julia para *usar* o módulo. Na verdade, a maioria da funcionalidade existente em Julia está contida em módulos a que só conseguimos aceder se o pedirmos explicitamente.

A título de exemplo, consideremos as seguintes definições:

```
module Areas

export area_circulo

pi = 3.14159

quadrado(x) =
  x*x

area_circulo(r) =
  pi*quadrado(r)

end
```

Estas definições constituem um módulo com o nome `Areas`, ele próprio visível no módulo pré-existente `Main`. Se um programa Julia pretender usar a funcionalidade implementada neste módulo deverá usar a instrução `using` e fazer:

```
using Main.Areas

volume_cilindro(r, h) =
  area_circulo(r)*h
```

Note-se como o nome *exportado* do módulo `Areas` é acedido. Naturalmente, quando se pretende usar outros módulos já existentes em Julia, o processo é idêntico, com uma pequena excepção: módulos que foram instalados dispensam o prefixo `Main.`.[3]

Exercício 1.15.1 Defina um módulo denominado `Hyperbolic` contendo as funções seno hiperbólico (`sinh`), cosseno hiperbólico (`cosh`) e tangente hiperbólica (`tanh`) a partir das definições matemáticas:

$$\sinh x = \frac{e^x - e^{-x}}{2}$$

$$\cosh x = \frac{e^x + e^{-x}}{2}$$

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Exercício 1.15.2 Defina, no módulo da pergunta anterior, as funções inversas do seno hiperbólico (`asinh`), cosseno hiperbólico (`acosh`) e tangente hiperbólica (`atanh`) cujas definições matemáticas são:

$$\operatorname{asinh} x = \sinh^{-1} x = \ln(x + \sqrt{x^2 + 1})$$

[3]Nesta obra não iremos discutir o processo de instalação de módulos. O leitor interessado em criar os seus próprios módulos deverá consultar a documentação oficial da linguagem Julia.

$$\operatorname{acosh} x = \cosh^{-1} x = \pm \ln(x + \sqrt{x^2 - 1})$$

$$\operatorname{atanh} x = \tanh^{-1} x = \begin{cases} \frac{1}{2} \ln(\frac{1+x}{1-x}), & \text{se } |x| < 1 \\ \frac{1}{2} \ln(\frac{x+1}{x-1}), & \text{se } |x| > 1 \end{cases}$$

Capítulo 2

Modelação

2.1 Introdução

Vimos, nas secções anteriores, alguns dos tipos de dados pré-definidos em Julia. Em muitos casos, esses tipos de dados são os necessários e suficientes para nos permitir criar os nossos programas. Noutros casos, será necessário introduzirmos novos tipos de dados. Nesta secção iremos estudar um novo tipo de dados que nos será particularmente útil para a modelação de entidades geométricas: coordenadas.

2.2 Coordenadas

A Arquitectura pressupõe a localização de elementos no espaço. Essa localização expressa-se em termos do que se designa por *coordenadas*: cada coordenada é um número e uma sequência de coordenadas identifica univocamente um ponto no espaço. A Figura 2.1 esquematiza uma possível sequência de coordenadas (x, y, z) que identificam o ponto P num espaço tridimensional. Diferentes sistemas de coordenadas são possíveis e, no caso da Figura 2.1, estamos a empregar aquele que se designa por sistema de coordenadas *rectangulares*, também conhecido por sistema de coordenadas *Cartesianas*, em honra do seu inventor: René Descartes.[1]

Existem várias operações úteis que podemos realizar com coordenadas. Por exemplo, podemos calcular a distância entre duas posições no espaço $P = (p_x, p_y, p_z)$ e $Q = (q_x, q_y, q_z)$. Essa distância é determinada pela fórmula

$$d = \sqrt{(q_x - p_x)^2 + (q_y - p_y)^2 + (q_z - p_z)^2}$$

e um primeiro esboço da sua tradução para Julia será:

[1] Descartes é um filósofo Francês do século XVII, autor da famosa frase "*Cogito ergo sum*" (penso, logo existo) e de inúmeras contribuições na Matemática e na Física.

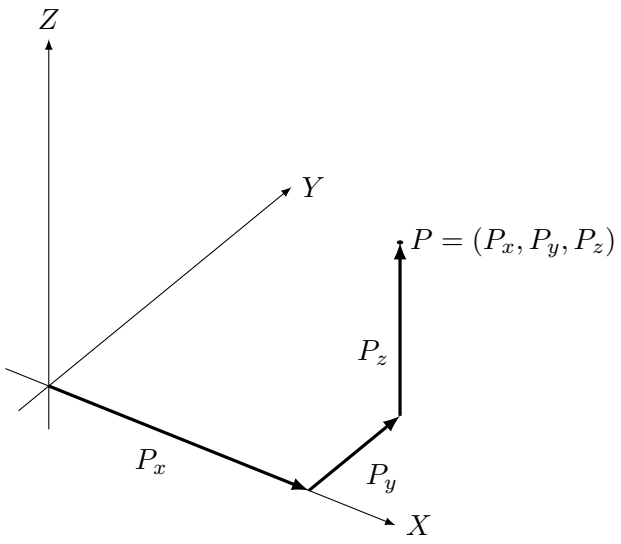


Figura 2.1: Coordenadas rectangulares de um ponto no espaço.

```
distancia(px, py, pz, qx, qy, qz) =
  sqrt((qx - px)^2 + (qy - py)^2 + (qz - pz)^2)
```

A distância entre as posições $(2, 1, 3)$ e $(5, 6, 4)$ será então

```
julia> distancia(2, 1, 3, 5, 6, 4)
5.916079783099616
```

Infelizmente, ao tratarmos as coordenadas como três números independentes, tornamos o uso das funções pouco claro. Isso já é visivel no exemplo anterior, em que a função `distancia` é invocada com seis argumentos, obrigando o leitor a perceber onde é que acabam as coordenadas de uma posição e começam as da outra. O problema torna-se mais grave quando uma função tem de produzir, como resultado, não um número, como acontece com a função `distancia`, mas sim uma posição no espaço, como acontece, por exemplo, com a função que computa a posição intermédia entre duas posições $P = (p_x, p_y, p_z)$ e $Q = (q_x, q_y, q_z)$. Essa posição intermédia é calculada pela fórmula

$$(\frac{p_x + q_x}{2}, \frac{p_y + q_y}{2}, \frac{p_z + q_z}{2})$$

mas é difícil conceber uma função que a implemente pois, aparentemente, teria de calcular simultâneamente três resultados diferentes, um para cada uma das coordenadas X, Y, e Z.

Para lidar com estes problemas, os matemáticos inventaram o conceito de *tuplo*. Informalmente, um tuplo é apenas um agrupamento de valores

mas, em inúmeros casos, este agrupamento acaba por ter um nome específico. Um número racional, por exemplo, é um tuplo que agrupa dois números inteiros, denominados o numerador e o denominador. Do mesmo modo, uma posição no espaço é um tuplo, pois agrupa três coordenadas.

Todas as linguagem de programação disponibilizam mecanismos para permitir a criação e manipulação de tuplos. No caso da linguagem Julia, para além dos tuplos já pré-definidos, foram definidos no módulo `Khepri` outros tuplos que nos serão úteis para a modelação geométrica.

Para os podermos usar, temos de requerer o módulo `Khepri`.

```
using Khepri
```

Para criarmos as coordenadas (x, y, z), o Khepri disponibiliza a função `xyz`:

```
julia> xyz(1, 2, 3)
xyz(1, 2, 3)
julia> xyz(2*3, 4 + 1, 6 - 2)
xyz(6, 5, 4)
```

Note-se que o resultado da avaliação da expressão `xyz(1, 2, 3)` é um valor que representa uma posição no espaço Cartesiano tridimensional.

Os valores produzidos pela função `xyz` pertencem a um novo tipo de dados, distinto dos outros tipos de dados que já tínhamos discutido anteriormente, tais como os números e as cadeias de caracteres. Tal como acontecia com estes últimos, existem operações específicas para manipular os valores deste tipo de dados.

2.3 Operações com Coordenadas

Agora que já somos capazes de criar coordenadas, podemos repensar as operações que manipulam coordenadas. Comecemos pela função que calcula a distância entre duas posições $P = (p_x, p_y, p_z)$ e $Q = (q_x, q_y, q_z)$. Como vimos, essa distância é determinada pela fórmula

$$\sqrt{(q_x - p_x)^2 + (q_y - p_y)^2 + (q_z - p_z)^2}$$

e um primeiro esboço da sua tradução para Julia será:

```
distancia(p, q) =
  sqrt((qx - px)^2 + (qy - py)^2 + (qz - pz)^2)
```

Para completarmos a função, temos de saber como obter as diferentes coordenadas x, y, e z de uma posição P. Para isso, o Khepri fornece as funções `cx`, `cy`, e `cz`, abreviaturas de c*oordinate* `x`, c*oordinate* `y`, e c*oordinate*

`z`, respectivamente. Cada uma destas funções recebe uma posição como argumento e calcula a coordenada respectiva.

As funções `xyz`, `cx`, `cy` e `cz`, são as operações fundamentais sobre coordenadas, sendo que a primeira nos permite construir uma posição a partir de três números e as restantes nos permitem saber quais são os números que determinam uma posição. Por este motivo, a primeira operação diz-se um *construtor* de coordenadas e as restantes dizem-se *selectores* de coordenadas.

Embora não saibamos como é que estas operações funcionam internamente, sabemos que são consistentes entre si, sendo essa consistência assegurada pelas seguintes equações:

$$\texttt{cx}(\texttt{xyz}(x, y, z)) = x$$

$$\texttt{cy}(\texttt{xyz}(x, y, z)) = y$$

$$\texttt{cz}(\texttt{xyz}(x, y, z)) = z$$

Usando estas funções, já podemos escrever:

```
distancia(p, q) =
  sqrt((cx(q) - cx(p))^2 + (cy(q) - cy(p))^2 + (cz(q) - cz(p))^2)
```

Podemos agora experimentar a função `distancia` com um caso concreto:

```
julia> distancia(xyz(2, 1, 3), xyz(5, 6, 4))
5.916079783099616
```

Para simplificar o uso das funções `cx`, `cy`, e `cz`, o Khepri disponibiliza uma sintaxe alternativa mais próxima da tradição matemática. Usando esta sintaxe, as expressões `cx(p)`, `cy(p)`, e `cz(p)` podem ser escritas como `p.x`, `p.y`, e `p.z`, respectivamente, o que permite simplificar a definição da função distância:

```
distancia(p, q) =
  sqrt((q.x - p.x)^2 + (q.y - p.y)^2 + (q.z - p.z)^2)
```

Vejamos agora um outro exemplo. Imaginemos que pretendemos definir uma operação que calcula a posição de um ponto após uma translação, expressa em termos das componentes ortogonais Δ_x, Δ_y, e Δ_z, tal como está apresentado na Figura 2.2. Como se pode ver nesta figura, sendo $P = (x, y, z)$, teremos $P' = (x + \Delta_x, y + \Delta_y, z + \Delta_z)$. Há no entanto uma outra possibilidade baseada no uso de *vectores*. Um vector é uma entidade matemática que representa um deslocamento aplicável a qualquer posição e que, por isso, não tem origem ou destino, sendo caracterizado por uma direcção e uma magnitude.

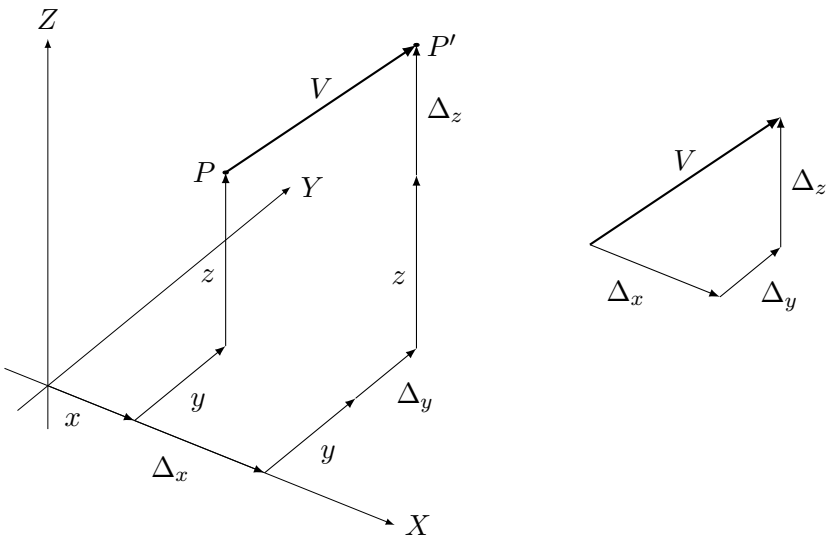


Figura 2.2: O ponto P' como resultado da translação do ponto $P = (x, y, z)$ de Δ_x no eixo X, Δ_y no eixo Y, e Δ_z no eixo Z.

Na Figura 2.2, à direita, o vector $V = (\Delta_x, \Delta_y, \Delta_z)$ representa o deslocamento que podemos aplicar à posição P para determinar a posição P'. Se definirmos a soma da posição $P = (x, y, z)$ com o vector $V = (\Delta_x, \Delta_y, \Delta_z)$ como $(x + \Delta_x, y + \Delta_y, z + \Delta_z)$, então torna-se evidente que $P' = P + V$. Do mesmo modo, é possível definir o vector V como a diferença entre as duas posições $V = P' - P$. Muitas outras operações fazem sentido no contexto dos vectores, incluindo a adição e subtracção de vectores e o produto de vectores por escalares ou por outros vectores. Esta álgebra vectorial, conhecida desde o século XVII, permite simplificar os cálculos envolvendo posições e deslocamentos.

Dada a utilidade dos vectores, estes são também providenciados pelo Khepri, sob a forma de uma operação denominada `vxyz` que, a partir de três deslocamentos ao longo dos eixos X, Y, e Z, produz o vector correspondente. À semelhança do que acontece com as posições, também os vectores podem ser usados com as operações `cx`, `cy`, e `cz`, bem como com a sintaxe alternativa `.x`, `.z`, e `.z`.

A seguinte interação mostra alguns exemplos da utilização de vectores:

```
julia> xyz(1,2,3) + vxyz(3,2,1)
xyz(4,4,4)
julia> xyz(4,5,6) - xyz(3,2,1)
vxyz(1,3,5)
julia> vxyz(4,5,6) - vxyz(3,2,1)
vxyz(1,3,5)
julia> xyz(1,2,3) + (xyz(4,5,6) - xyz(3,2,1))
xyz(2,5,8)
julia> xyz(1,2,3) + xyz(4,5,6) - xyz(3,2,1)
ERROR: MethodError: no method matching +(::XYZ, ::XYZ)
julia> xyz(1,2,3) - xyz(3,2,1) + xyz(4,5,6)
xyz(2,5,8)
julia> vxyz(1,2,3) + vxyz(4,5,6) - xyz(3,2,1)
xyz(2,5,8)
```

Note-se que nem todas as operações são possíveis. Em particular, a soma de posições não faz sentido.

Exercício 2.3.1 Defina a função `posicao_media` que calcula a posição intermédia entre duas outras posições P_0 e P_1.

Exercício 2.3.2 Defina a função `posicoes_iguais` que compara duas posições e devolve verdade apenas quando são coincidentes. Note que duas posições são coincidentes quando as coordenadas x, y e z forem iguais.

Exercício 2.3.3 Um vector *unitário* é um vector de magnitude unitária. Defina a operação `vector_unitario` que, dado um vector qualquer, calcula o vector unitário que tem a mesma direcção que o vector dado.

Exercício 2.3.4 O vector *simétrico* de um vector $\vec{v}$ é o vector $\vec{v}'$ tal que $\vec{v} + \vec{v}' = 0$. Dito de outro modo, é o vector de igual magnitude mas direcção oposta. Defina a operação `vector_simetrico` que, dado um vector qualquer, calcula o vector simétrico do vector dado.

2.4 Coordenadas Bidimensionais

Tal como as coordenadas tridimensionais localizam posições no espaço, as coordenadas bidimensionais localizam posições no plano. A questão que se coloca é: qual plano? Do ponto de vista matemático, a questão não é relevante pois é perfeitamente possível pensar em geometria no plano sem pensar no plano propriamente dito, mas quando tentamos visualizar essa geometria num programa de modelação tridimensional, inevitavelmente temos de pensar na localização desse plano. Se nada for dito em contrário, os programas de CAD tridimensionais consideram que o plano bidimensional corresponde ao plano XY, localizado na cota Z igual a zero. Assim sendo, vamos considerar a coordenada bidimensional (x, y) como uma notação simplificada da coordenada tridimensional $(x, y, 0)$.

Usando esta simplificação, podemos definir um construtor de coordenadas bidimensionais à custa do construtor de coordenadas tridimensionais:

```
xy(x, y) =
  xyz(x, y, 0)
```

Do mesmo modo, podemos também definir vectores no plano à custa dos vectores tridimensionais:

```
vxy(x, y) =
  vxyz(x, y, 0)
```

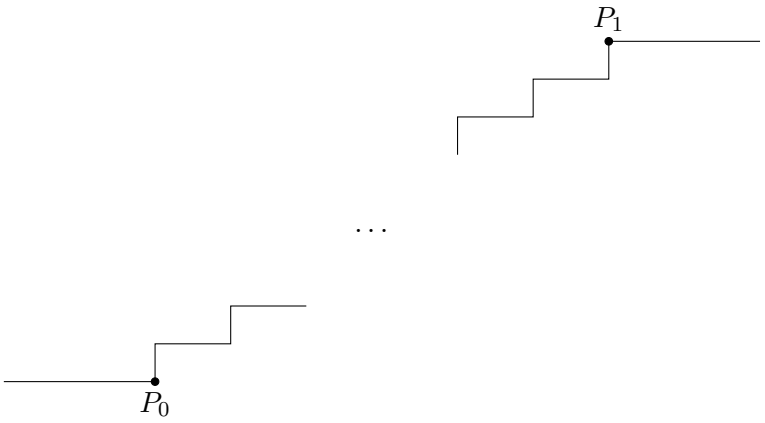
Uma das vantagens da definição de coordenadas bidimensionais como um caso particular das coordenadas tridimensionais é o facto de os selectores `cx`, `cy`, e `cz` serem automaticamente aplicáveis a coordenadas bidimensionais e, por arrasto, também o são as operações de coordenadas que definimos anteriormente, como a `distancia` e a soma de posições com vectores.

Exercício 2.4.1 Dado um ponto $P_0 = (x_0, y_0)$ e uma recta definida por dois pontos $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$, a distância mínima d do ponto P_0 à recta obtém-se pela fórmula:

$$d = \frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

Defina uma função denominada `distancia_ponto_recta` que, dadas as coordenadas dos pontos P_0, P_1 e P_2, devolve a distância mínima de P_0 à recta definida por P_1 e P_2.

Exercício 2.4.2 Sabendo que o espelho máximo admissível para um degrau é de 0.18m, defina uma função que calcula o número mínimo de espelhos que a escada representada no seguinte esquema necessita para fazer a ligação entre os pontos P_0 e P_1.



2.5 Coordenadas Polares

Embora o sistema de coordenadas rectangulares seja muito usado, existem outros sistemas de coordenadas que podem ser mais úteis em determinadas situações. A título de exemplo, imaginemos que queremos dispor n elementos igualmente espaçados a uma distância d da origem, tal como está parcialmente ilustrado na Figura 2.3. Como é lógico, os elementos acabarão por formar um círculo e o ângulo entre cada dois elementos terá de ser $\frac{2\pi}{n}$.

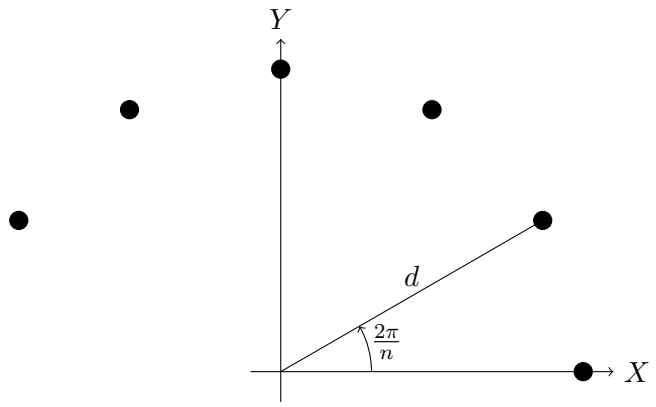


Figura 2.3: Posições ao longo de um círculo.

Usando o eixo X como referência, podemos dizer que o primeiro elemento ficará posicionado nesse eixo, à distância d da origem, o segundo elemento ficará à mesma distância mas posicionado numa linha que faz um ângulo $\frac{2\pi}{n}$ com o eixo X, o terceiro ficará à mesma distância mas posicionado numa linha que faz um ângulo $2\frac{2\pi}{n}$ com o eixo X, e assim sucessivamente. Contudo, se tentarmos descrever estas mesmas posições no sistema de coordenadas rectangulares, veremos que se perde imediatamente a regularidade expressa em "e assim sucessivamente." É esta necessidade de regularidade que nos leva a considerar outros sistemas de coordenadas e, em particular, o sistema de coordenadas polares.

Tal como representado da Figura 2.4, uma posição no plano bidimensional é descrita, em coordenadas rectangulares, pelos números x e y—significando, respectivamente, a *abcissa* e a *ordenada*—enquanto que a mesma posição em coordenadas polares é descrita pelos números ρ e ϕ—significando, respectivamente, o *raio vector* (também chamado *módulo*) e o *ângulo polar* (também chamado *argumento*). Com a ajuda da trigonometria e do teorema de Pitágoras conseguimos facilmente converter de coordenadas pola-

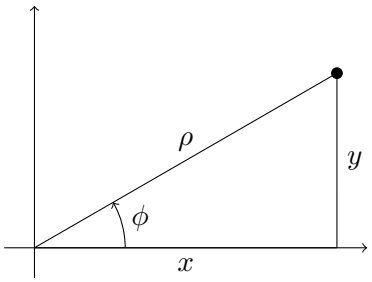


Figura 2.4: Coordenadas rectangulares e polares.

res para coordenadas rectangulares:

$$\begin{cases} x = \rho \cos \phi \\ y = \rho \sin \phi \end{cases}$$

ou de coordenadas rectangulares para coordenadas polares:

$$\begin{cases} \rho = \sqrt{x^2 + y^2} \\ \phi = \arctan \dfrac{y}{x} \end{cases}$$

Com base nas equações anteriores, podemos definir o construtor de coordenadas polares `pol` (abreviatura de "**pol**ar") que contrói coordenadas a partir da sua representação polar simplesmente convertendo-a para a representação rectangular equivalente.

```
pol(ro, fi) =
  xy(ro*cos(fi), ro*sin(fi))
```

Assim sendo, o tipo coordenadas polares fica implementado em termos do tipo coordenadas rectangulares. Por este motivo, os selectores do tipo coordenadas polares—a função `pol_ro` que nos permite obter o módulo ρ e a função `pol_fi` que nos permite obter o argumento ϕ—terão de usar os selectores de coordenadas rectangulares, i.e., `cx` e `cy`:[2]

```
pol_ro(c) =
  sqrt(cx(c)^2 + cy(c)^2)


pol_fi(c) =
  atan2(cy(c), cx(c))
```

Eis alguns exemplos do uso destas funções:[3]

[2]Estas funções encontram-se pré-definidas em Khepri, com os nomes `pol_rho` e `pol_phi`.

[3]Note-se, nestes exemplos, que alguns valores das coordenadas não são zero ou um, como seria expectável, mas sim valores muito próximos desses números, que resultam de erros de arredondamento. Note-se também que, como estamos a usar coordenadas bidimensionais, a coordenada z é sempre igual a zero.

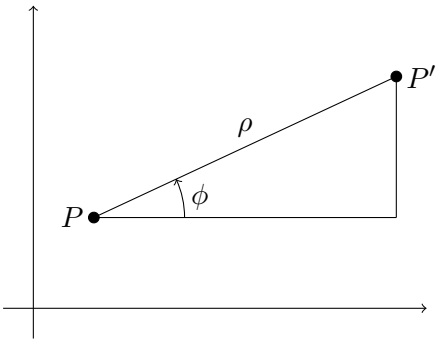


Figura 2.5: O deslocamento de um ponto em coordenadas polares.

```
julia> pol(1, 0)
xyz(1, 0, 0)
julia> pol(sqrt(2), pi/4)
xyz(1.0000000000000002, 1.0, 0)
julia> pol(1, pi/2)
xyz(6.123233995736766e-017, 1.0, 0)
julia> pol(1, pi)
xyz(-1.0, 1.2246467991473532e-016, 0)
```

No caso de querermos especificar vectores em coordenadas polares, tal como ilustramos na Figura 2.5, podemos usar uma abordagem em tudo idêntica, definindo o construtor `vpol`:

```
vpol(ro, fi) =
  vxy(ro*cos(fi), ro*sin(fi))
```

Como exemplos de utilização, temos:

```
julia> xy(1, 2) + vpol(sqrt(2), pi/4)
xyz(2.0, 3.0, 0.0)
julia> xy(1, 2) + vpol(1, 0)
xyz(2.0, 2.0, 0.0)
julia> xy(1, 2) + vpol(1, pi/2)
xyz(1.0, 3.0, 0.0)
```

Exercício 2.5.1 A função `posicoes_iguais` definida no exercício 2.3.2 compara as coordenadas de duas posições e devolve verdade apenas quando são coincidentes. No entanto, se tivermos em conta que as operações numéricas podem produzir erros de arredondamento, é perfeitamente possível que duas posições que, em teoria, deveriam ser iguais, na prática, sejam consideradas diferentes. Por exemplo, a posição $(-1, 0)$ em coordenadas rectangulares pode ser expressa através das coordenadas polares $\rho = 1, \phi = \pi$ mas o Julia não as considera iguais, tal como é perfeitamente visível na seguinte interação:

```
julia> posicoes_iguais(xy(-1, 0), pol(1, pi))
false
julia> xy(-1, 0)
xyz(-1, 0, 0)
julia> pol(1, pi)
xyz(-1.0, 1.2246467991473532e-016, 0)
```

Como se pode ver, embora as coordenadas não sejam iguais, elas são bastante próximas, ou seja, a distância entre elas é muito próxima de zero. Proponha uma nova definição para a função `posicoes_iguais` baseada no conceito de distância entre as coordenadas.

2.6 Modelação Geométrica Bidimensional

Nesta secção vamos introduzir algumas operações de modelação geométrica bidimensional.

Para visualizarmos as formas que criamos é necessário dispormos de uma aplicação de CAD, como o AutoCAD ou o Rhino. A escolha de qual a aplicação de CAD que pretendemos usar é feita usando a função `backend`. Assim, um programa que use o Khepri começa geralmente com

```
using Khepri
backend(autocad)
```

ou com

```
using Khepri
backend(rhino)
```

dependendo se queremos usar, respectivamente, o AutoCAD ou o Rhino.

Comecemos por considerar a criação de círculos. Para isso, vamos usar a função `circle`, que recebe o centro e o raio do círculo. A Figura 2.6 mostra o resultado da avaliação do seguinte programa em AutoCAD:[4]

```
using Khepri
backend(autocad)

circle(pol(0, 0), 4)
circle(pol(4, pi/4), 2)
circle(pol(6, pi/4), 1)
```

Uma outra operação de modelação muito usada é a que cria segmentos de recta: `line`. Na sua versão mais simples, esta função recebe duas posições correspondentes às extremidades do segmento de recta. No entanto, é

[4]De futuro, iremos omitir o cabeçalho do programa que requere o Khepri e escolhe a ferramenta de CAD e iremos focar a nossa atenção apenas nas operações de modelação geométrica.

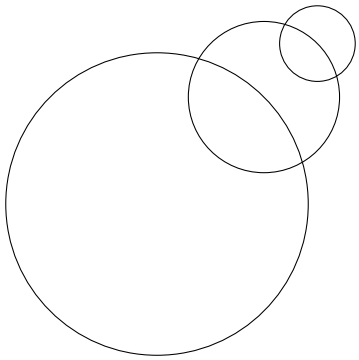


Figura 2.6: Um conjunto de círculos.

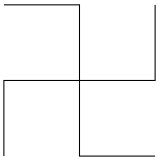


Figura 2.7: Um conjunto de linhas.

possível invocar a função com qualquer número de posições, caso em que a função cria sucessivos segmentos de recta a unir cada posição à posição seguinte, formando uma linha poligonal. A Figura 2.7 ilustra uma suástica[5] produzida pelas seguintes invocações:

```
line(xy(-1, -1), xy(-1, 0), xy(1, 0), xy(1, 1))
line(xy(-1, 1), xy(0, 1), xy(0, -1), xy(1, -1))
```

No caso de pretendermos desenhar linhas poligonais *fechadas*, é preferível usarmos a função `polygon`, em tudo idêntica à função `line` mas que cria um segmento de recta adicional a unir a última posição à primeira. A Figura 2.8 mostra um exemplo resultante da avaliação da seguinte expressão:

```
polygon(pol(1, 2*pi*0/5),
        pol(1, 2*pi*1/5),
        pol(1, 2*pi*2/5),
        pol(1, 2*pi*3/5),
        pol(1, 2*pi*4/5))
```

Para o caso de polígonos *regulares*, i.e., polígonos com todos os lados e todos os ângulos iguais, como o apresentado na Figura 2.8, é preferível usar a função `regular_polygon`. Esta função tem, como argumentos, o

[5] A suástica é um símbolo místico usado em diferentes culturas e cujas primeiras ocorrências datam do período neolítico.

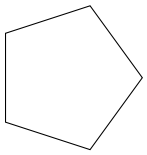


Figura 2.8: Um polígono.

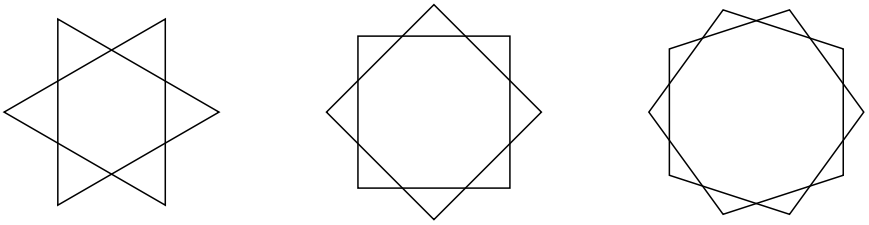


Figura 2.9: Triângulos, quadrados e pentágonos sobrepostos com diferentes ângulos de rotação.

número de lados do polígono, o centro, um raio, um ângulo de rotação e, finalmente, um boleano que indica se o raio é de um círculo inscrito ao polígono (ou seja, o raio é a distância dos lados ao centro) ou de um círculo circunscrito ao polígono (ou seja, o raio é a distância dos vértices ao centro). Por omissão, o centro é a origem, o raio é unitário, o ângulo é zero e o círculo é circunscrito.

Usando a função `regular_polygon`, a Figura 2.8 pode ser obtida através de:

```
regular_polygon(5)
```

Exemplos mais interessantes podem ser obtidos através de diferentes ângulos de rotação. Por exemplo, a seguinte sequência de expressões produz a imagem representada na Figura 5.6:

```
regular_polygon(3, xy(0, 0), 1, 0, true)
regular_polygon(3, xy(0, 0), 1, pi/3, true)

regular_polygon(4, xy(3, 0), 1, 0, true)
regular_polygon(4, xy(3, 0), 1, pi/4, true)

regular_polygon(5, xy(6, 0), 1, 0, true)
regular_polygon(5, xy(6, 0), 1, pi/5, true)
```

Para o caso de polígonos de quatro lados alinhados com os eixos X e Y, existe uma função muito simples: `rectangle`. Esta função tanto pode ser usada com as posições do canto inferior esquerdo e superior direito do rectângulo, como pode ser usada com a posição do canto inferior esquerdo e as duas dimensões do rectângulo, tal como podemos ver no seguinte exemplo cujo resultado está representado na Figura 2.10:

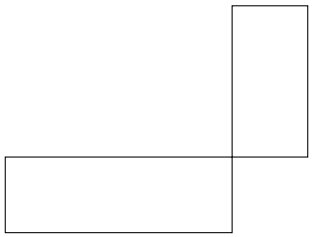


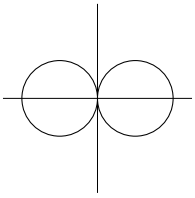
Figura 2.10: Um conjunto de rectângulos.

```
rectangle(xy(0, 1), xy(3, 2))
rectangle(xy(3, 2), 1, 2)
```

Nas secções seguintes iremos introduzir as restantes funções de modelação geométricas disponíveis em Khepri.

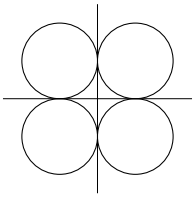
Exercício 2.6.1 Refaça o desenho apresentado na Figura 2.6 mas utilizando apenas coordenadas rectangulares.

Exercício 2.6.2 Pretendemos colocar duas circunferências de raio unitário em torno da origem de modo a que fiquem encostadas uma à outra, tal como se ilustra no seguinte desenho:



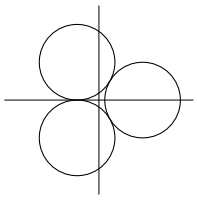
Escreva uma sequência de expressões que, quando avaliadas, produzem a figura anterior.

Exercício 2.6.3 Pretendemos colocar quatro circunferências de raio unitário em torno da origem de modo a que fiquem encostadas umas às outras, tal como se ilustra no seguinte desenho:



Escreva uma sequência de expressões que, quando avaliadas, produzem a figura anterior.

Exercício 2.6.4 Pretendemos colocar três circunferências de raio unitário em torno da origem de modo a que fiquem encostadas umas às outras, tal como se ilustra no seguinte desenho:



Escreva uma sequência de expressões que, quando avaliadas, produzem a figura anterior.

2.7 Efeitos Secundários

Vimos anteriormente que qualquer expressão Julia tem um valor. Isso é visivel quando avaliamos uma expressão aritmética, mas também o é quando avaliamos uma expressão geométrica:

```
julia> 1 + 2
3
julia> circle(xy(1, 2), 3)
Circle(...)
```

No exemplo anterior, o resultado da avaliação da segunda expressão é uma entidade geométrica. Quando o Julia escreve um resultado que é uma entidade geométrica, emprega uma notação baseada no nome dessa entidade. No entanto, na maioria dos casos, não estamos apenas interessados em ver uma entidade geométrica como um fragmento de texto, estamos também interessados em *visualizar* a entidade no espaço. Para esse propósito, a avaliação de expressões geométricas tem também um *efeito secundário*: as formas geométricas produzidas são automaticamente adicionadas à aplicação de CAD que tiver sido escolhida.

Assim, a avaliação do seguinte programa:

```
using Khepri
backend(autocad)

circle(xy(1, 2), 3)
```

tem, como resultado, um valor abstracto que representa um círculo de raio 3 com centro no ponto $(1, 2)$ e, como efeito, esse círculo torna-se imediatamente visivel no AutoCAD.

Este comportamento das funções geométricas, como `circle`, `line`, `rectangle`, etc, é fundamentalmente diferente do comportamento das restantes funções que vimos até agora pois, anteriormente, as funções eram usadas para computar algo, i.e., para produzir um valor a partir da sua invocação com determinados argumentos e, agora, não é o valor que mais interessa mas sim o efeito secundário (também chamado *efeito colateral*) que nos permite visualizar a forma geométrica na aplicação de CAD.

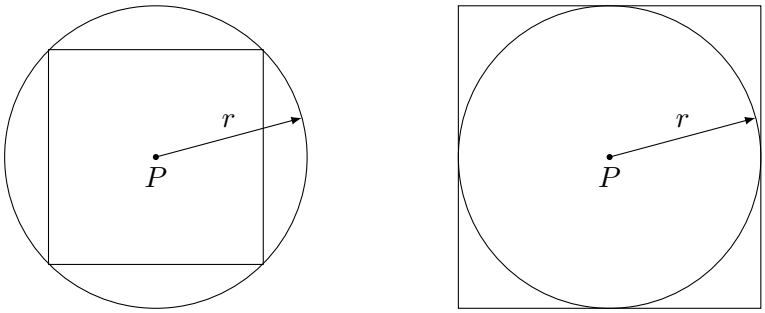


Figura 2.11: Um círculo com um quadrado inscrito (à esquerda) e circunscrito (à direita).

Um dos aspectos importantes da utilização de efeitos secundários está na possibilidade da sua *composição*. A composição de efeitos secundários faz-se através da sua *sequenciação*, i.e., da realização sequencial dos vários efeitos. Na secção seguinte iremos discutir a sequenciação de efeitos secundários.

2.8 Sequenciação

Até agora, temos combinado expressões matemáticas usando operadores matemáticos. Por exemplo, a partir das expressões `sin(x)` e `cos(x)`, podemos calcular a sua divisão através de `sin(x)/cos(x)`. Isto é possível porque a avaliação das subexpressões `sin(x)` e `cos(x)` irá produzir dois valores que podem então ser usados para fazer a divisão.

Já no caso de expressões envolvendo as funções geométricas, a sua combinação tem de ser realizada de forma diferente pois, como vimos, a sua avaliação também produz efeitos secundários. Como é precisamente a realização desses efeitos que nos interessa, o Julia providencia uma forma de os realizarmos sequencialmente, i.e., uns a seguir aos outros. A título de exemplo, consideremos uma função que desenha um círculo de raio r centrado em P com um quadrado inscrito ou circunscrito, dependendo de uma escolha do utilizador, tal como apresentamos na Figura 2.11.

A definição desta função poderá começar por algo do género:

```
circulo_quadrado(p, r, inscrito) =
  ...
```

O desenho produzido pela função depende, naturalmente, do valor lógico do parâmetro `inscrito`, ou seja, a definição da função deverá ser algo da forma:

```
circulo_quadrado(p, r, inscrito) =
  if inscrito
    cria um círculo e um quadrado inscrito no círculo
  else
    cria um círculo e um quadrado circunscrito ao círculo
  end
```

Note-se que, em cada alternativa do `if`, a função necessita de realizar *dois* efeitos secundários, nomeadamente, criar um círculo e criar um rectângulo. Para este fim, o Julia disponibiliza a forma `begin`-`end`. Esta forma recebe várias expressões que avalia sequencialmente, i.e., uma a seguir à outra, devolvendo o valor da última. Logicamente, se apenas o valor da última expressão é utilizado, então os valores de todas as outras avaliações dessa sequência são descartados e estas apenas são relevantes pelos efeitos secundários que possam provocar, como seja a criação de formas geométricas na aplicação de CAD que estiver a ser usada.

Usando a forma `begin`-`end` já podemos detalhar um pouco mais a função:

```
circulo_quadrado(p, r, inscrito) =
  if inscrito
    begin
      cria um círculo
      cria um quadrado inscrito no círculo
    end
  else
    begin
      cria um círculo
      cria um quadrado circunscrito ao círculo
    end
```

Felizmente, a forma `if` também permite ter várias expressões em cada alternativa, que serão avaliadas sequencialmente, i.e., uma a seguir à outra. Usando esta capacidade da forma `if` podemos simplificar a função:

```
circulo_quadrado(p, r, inscrito) =
  if inscrito
    cria um círculo
    cria um quadrado inscrito no círculo
  else
    cria um círculo
    cria um quadrado circunscrito ao círculo
  end
```

Basta-nos agora traduzir cada um dos desenhos elementares em invocações das funções correspondentes. No caso do quadrado inscrito, vamos utilizar coordenadas polares pois sabemos que os vértices terão de ficar sobre o círculo, um a $\pi/4$ e ou outro a $\pi + \pi/4 = 5\pi/4$. Assim, temos:

```
circulo_quadrado(p, r, inscrito) =
  if inscrito
    circle(p, r)
    rectangle(p + vpol(r, 5/4*pi), p + vpol(r, 1/4*pi))
  else
    circle(p, r)
    rectangle(p + vxy(-r, -r), p + vxy(r, r))
  end
```

Uma observação atenta da função `circulo_quadrado` mostra que há alguma repetição, uma vez que a criação do círculo é sempre feita, independentemente do rectângulo ser inscrito ou circunscrito. Isto sugere reescrever a definição da função de modo a que a criação do círculo seja realizada fora do `if`:

```
circulo_quadrado(p, r, inscrito) =
  begin
    circle(p, r)
    if inscrito
      rectangle(p + vpol(r, 5/4*pi), p + vpol(r, 1/4*pi))
    else
      rectangle(p + vxy(-r, -r), p + vxy(r, r))
    end
  end
```

Reparemos que, mais uma vez, estamos a empregar sequenciação, agora da criação do círculo, a que se segue a forma `if` para decidir o que fazer a seguir.

Exercício 2.8.1 Defina uma função denominada `circulo_e_raio` que, dadas as coordenadas do centro do círculo e o raio desse círculo, cria o círculo especificado e, à semelhança do que se vê na Figura 2.6, coloca o texto a descrever o raio do círculo à direita do círculo. O texto deverá ter um tamanho proporcional ao raio do círculo. Sugestão: empregue a função Julia `string`, que transforma os seus argumentos numa *string*, e a função Khepri `text` que desenha um texto, a partir de uma *string* a descrever o texto, da posição do texto e do tamanho da letra a empregar.

Exercício 2.8.2 Utilize a função `circulo_e_raio` definida na pergunta anterior para reconstituir a imagem apresentada na Figura 2.6.

2.9 A Ordem Dórica

Na Figura 2.12 apresentamos uma imagem do templo grego de Segesta. Este templo, que nunca chegou a ser acabado, foi construído no século quinto antes de Cristo e representa um excelente exemplo da *Ordem Dórica*, a mais antiga das três ordens da arquitectura Grega clássica. Nesta ordem, uma coluna caracteriza-se por ter um fuste, um coxim e um ábaco. O ábaco tem a forma de uma placa quadrada que assenta sobre o coxim, o

Figura 2.12: O Templo Grego de Segesta, exemplificando alguns aspectos da Ordem Dórica. Este templo nunca chegou a ser terminado, sendo visível, por exemplo, a falta das caneluras nas colunas. Fotografia de Enzo De Martino.

coxim assemelha-se a um tronco de cone invertido e assenta sobre o fuste, e o fuste assemelha-se a um tronco de cone com vinte *caneluras* em seu redor. Estas caneluras assemelham-se a uns canais semi-circulares escavados ao longo da coluna.[6] Quando os Romanos copiaram a Ordem Dórica introduziram-lhe um conjunto de alterações, em particular, nas caneluras que, muitas vezes, são simplesmente eliminadas.

Embora estejamos particularmente interessados na modelação tridimensional, por motivos pedagógicos vamos começar por esquematizar o alçado de uma coluna Dórica (sem caneluras). Nas secções seguintes iremos estender este processo para a criação de um modelo tridimensional.

Do mesmo modo que uma coluna Dórica se pode decompor nos seus componentes fundamentais – fuste, coxim e ábaco – também o desenho da coluna se poderá decompor no desenho dos seus componentes. Assim, vamos definir funções para desenhar o fuste, o coxim e o ábaco. A Figura 2.13 apresenta um modelo de referência. Comecemos por definir uma função para o desenho do fuste:

[6] Estas colunas apresentam ainda uma deformação intencional denominada *entasis*. A entasis consiste em dar uma ligeira curvatura à coluna e, segundo alguns autores, destina-se a corrigir uma ilusão de óptica que faz as colunas direitas parecerem encurvadas.

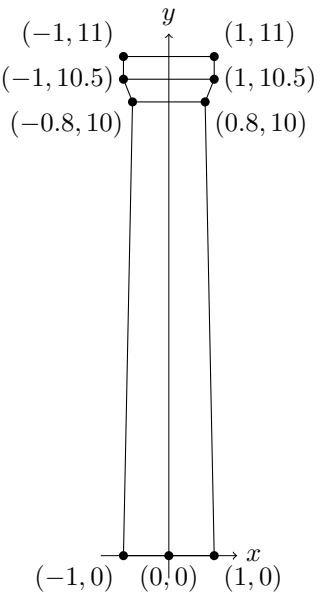


Figura 2.13: Uma coluna Dórica de referência.

```
fuste() =
  line(xy(-0.8, 10),
       xy(-1, 0),
       xy(1, 0),
       xy(0.8, 10),
       xy(-0.8, 10))
```

Neste exemplo, usámos a função `line` que, dada uma sequência de posições, constrói uma linha com vértices nesses pontos. Uma outra possibilidade, eventualmente mais correcta, seria a criação de uma linha poligonal *fechada*, algo que podemos fazer com a função `polygon`, omitindo a posição repetida, i.e.:

```
fuste() =
  polygon(xy(-0.8, 10),
          xy(-1, 0),
          xy(1, 0),
          xy(0.8, 10))
```

Para completar a figura, é ainda necessário definir uma função para o coxim e outra para o ábaco. No caso do coxim, o raciocínio é semelhante:

```
coxim() =
  polygon(xy(-0.8, 10),
          xy(-1, 10.5),
          xy(1, 10.5),
          xy(0.8, 10))
```

No caso do ábaco, podemos empregar uma abordagem idêntica ou po-

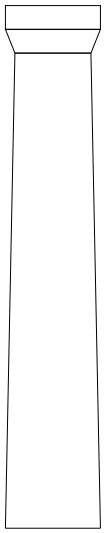


Figura 2.14: Uma coluna dórica.

demos explorar uma função do Khepri destinada à construção de rectângulos. Esta função apenas necessita de dois pontos para definir completamente o rectângulo:

```
abaco() =
  rectangle(xy(-1, 10.5), xy(1, 11))
```

Finalmente, vamos definir uma função que desenha as três partes da coluna:

```
coluna() =
  begin
    fuste()
    coxim()
    abaco()
  end
```

Repare-se, na função `coluna`, que ela invoca sequencialmente as funções `fuste`, `coxim` e, finalmente, `abaco`.

A Figura 2.14 mostra o resultado da invocação da função `coluna`.

2.10 Parametrização de Figuras Geométricas

Infelizmente, a coluna que criámos na secção anterior tem a sua posição e dimensão fixas, pelo que será difícil reutilizar a função noutros contextos. Naturalmente, esta função seria mais útil se a criação da coluna fosse *parametrizável*, i.e., se a criação dependesse dos vários parâmetros que caracterizam a coluna como, por exemplo, as coordenadas da base da coluna,

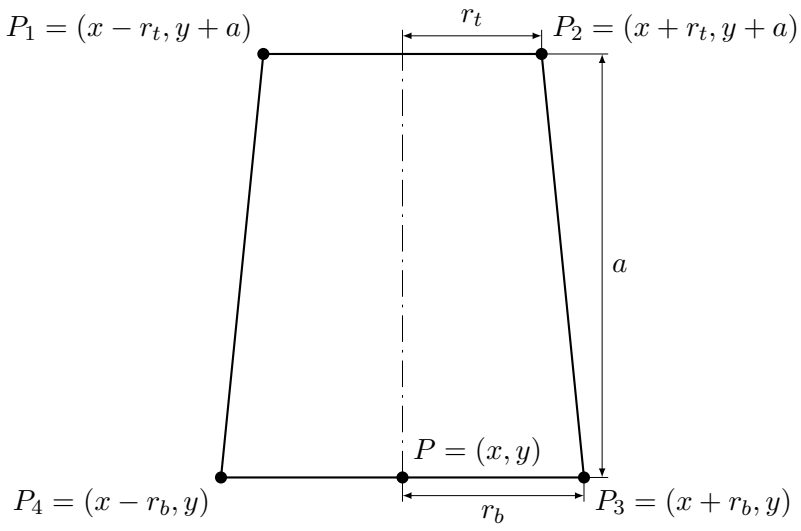


Figura 2.15: Esquema do desenho do fuste de uma coluna.

a altura do fuste, do coxim e do ábaco, os raios da base e do topo do fuste, etc.

Para se compreender a parametrização destas funções vamos começar por considerar o fuste representado esquematicamente na Figura 2.15.

O primeiro passo para parametrizarmos um desenho geométrico consiste na identificação dos parâmetros relevantes. No caso do fuste, um dos parâmetros óbvios é a localização espacial desse fuste, i.e., as coordenadas de um ponto de referência em relação ao qual fazemos o desenho do fuste. Assim, comecemos por imaginar que o fuste irá ser colocado com o centro da base num imaginário ponto P de coordenadas arbitrárias (x, y). Para além deste parâmetro, temos ainda de conhecer a altura do fuste a e os raios da base r_b e do topo r_t do fuste.

Para mais facilmente idealizarmos um processo de desenho, é conveniente assinalarmos no esquema alguns pontos de referência adicionais. No caso do fuste, uma vez que o seu desenho é, essencialmente, um trapézio, basta-nos idealizar o desenho deste trapézio através de uma sucessão de linhas rectas dispostas ao longo de uma sequência de pontos P_1, P_2, P_3 e P_4, pontos esses que conseguimos calcular facilmente a partir do ponto P.

Desta forma, estamos em condições de definir a função que desenha o fuste. Para tornar mais claro o programa, vamos empregar os nomes `a_fuste`, `r_base` e `r_topo` para caracterizar a altura a, o raio da base r_b e o raio do topo r_t, respectivamente. A definição fica então:

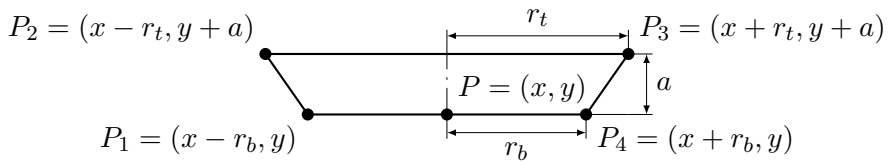


Figura 2.16: Esquema do desenho do coxim de uma coluna.

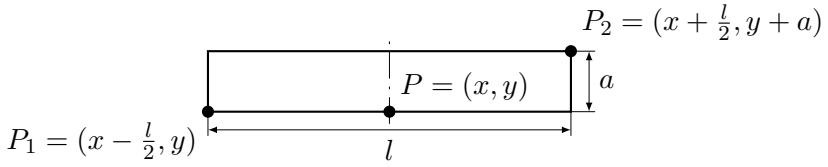


Figura 2.17: Esquema do desenho do ábaco de uma coluna.

```
fuste(p, a_fuste, r_base, r_topo) =
  polygon(p + vxy(-r_topo, a_fuste),
          p + vxy(-r_base, 0),
          p + vxy(r_base, 0),
          p + vxy(r_topo, a_fuste))
```

Em seguida, temos de especificar o desenho do coxim. Mais uma vez, convém pensarmos num esquema geométrico, tal como apresentamos na Figura 2.16.

Tal como no caso do fuste, a partir de um ponto P correspondente ao centro da base do coxim, podemos computar as coordenadas dos pontos que estão nas extremidades dos segmentos de recta que delimitam o desenho do coxim. Usando estes pontos, a definição da função fica com a seguinte forma:

```
coxim(p, a_coxim, r_base, r_topo) =
  polygon(p + vxy(-r_base, 0),
          p + vxy(-r_topo, a_coxim),
          p + vxy(r_topo, a_coxim),
          p + vxy(r_base, 0))
```

Terminado o fuste e o coxim, é preciso definir o desenho do ábaco. Para isso, fazemos um novo esquema que apresentamos na Figura 2.17.

Mais uma vez, vamos considerar como ponto de partida o ponto P no centro da base do ábaco. A partir deste ponto, podemos facilmente calcular os pontos P_1 e P_2 que constituem os dois extremos do rectângulo que representa o alçado do ábaco. Assim, temos:

```
abaco(p, a_abaco, l_abaco) =
  rectangle(p + vxy(l_abaco/-2, 0),
            p + vxy(l_abaco/2, a_abaco))
```

Finalmente, para desenhar a coluna completa, temos de combinar os

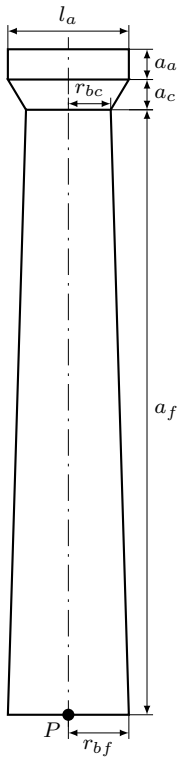


Figura 2.18: A composição do fuste, coxim e ábaco.

desenhos do fuste, do coxim e do ábaco. Apenas precisamos de ter em conta que, tal como a Figura 2.18 demonstra, o raio do topo do fuste coincide com o raio da base do coxim e o raio do topo do coxim é metade da largura do ábaco. A mesma figura mostra também que o centro da base do coxim corresponde a uma translação do centro da base do fuste até à altura do fuste e o centro da base do ábaco corresponde a uma translação da base do fuste até a altura do fuste mais a altura do coxim.

Tal como fizemos anteriormente, vamos dar nomes mais claros aos parâmetros da Figura 2.18. Usando os nomes `p`, `a_fuste`, `r_base_fuste`, `a_coxim`, `r_base_coxim`, `a_abaco` e `l_abaco` no lugar de, respectivamente, P, a_f, r_{bf}, a_c, r_{bc}, a_a e l_a, temos:

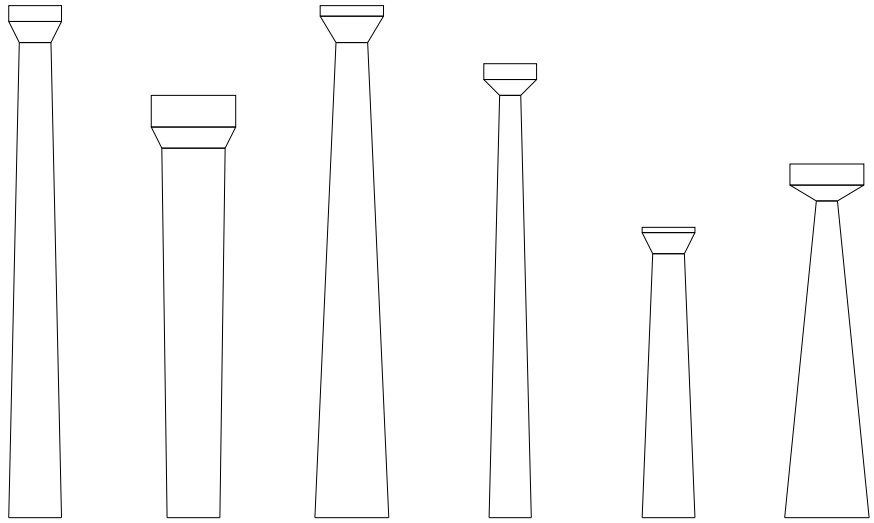


Figura 2.19: Variações de colunas dóricas.

```
coluna(p,
       a_fuste, r_base_fuste,
       a_coxim, r_base_coxim,
       a_abaco, l_abaco) =
  begin
    fuste(p, a_fuste, r_base_fuste, r_base_coxim)
    coxim(p + vxy(0, a_fuste), a_coxim, r_base_coxim, l_abaco/2)
    abaco(p + vxy(0, a_fuste + a_coxim), a_abaco, l_abaco)
  end
```

Com base nestas funções, podemos agora facilmente experimentar variações de colunas. As seguintes invocações produzem o desenho apresentado na Figura 2.19.

```
coluna(xy(0, 0), 9, 0.5, 0.4, 0.3, 0.3, 1.0)
coluna(xy(3, 0), 7, 0.5, 0.4, 0.6, 0.6, 1.6)
coluna(xy(6, 0), 9, 0.7, 0.5, 0.3, 0.2, 1.2)
coluna(xy(9, 0), 8, 0.4, 0.3, 0.2, 0.3, 1.0)
coluna(xy(12, 0), 5, 0.5, 0.4, 0.3, 0.1, 1.0)
coluna(xy(15, 0), 6, 0.8, 0.3, 0.2, 0.4, 1.4)
```

Como é óbvio pela análise desta figura, nem todas as colunas desenhadas obedecem aos *cânones* da ordem Dórica. Mais à frente iremos ver que modificações serão necessárias para evitar este problema.

2.11 Documentação

Na função `coluna`, `a_fuste` é a altura do fuste, `r_base_fuste` é o raio da base do fuste, `r_topo_fuste` é o raio do topo do fuste, `a_coxim` é a

altura do coxim, `a_abaco` é a altura do ábaco e, finalmente, `r_abaco` é o raio do ábaco. Uma vez que a função já tem vários parâmetros e o seu significado poderá não ser óbvio para quem lê a definição da função pela primeira vez, é conveniente *documentar* a função. Para isso, a linguagem Julia providencia uma sintaxe especial: sempre que surge o carácter `#`, o Julia ignora tudo o que vem a seguir até ao fim da linha. Isto permite-nos escrever texto nos nossos programas sem correr o risco de o Julia tentar perceber o que lá está escrito.

Usando documentação, o nosso programa completo para desenhar colunas dóricas fica com o aspecto ilustrado na Figura 2.20. Note-se que o exemplo destina-se a mostrar as diferentes formas de documentação usadas em Julia e não a mostrar um exemplo típico de programa documentado.[7]

Quando um programa está documentado, a sua leitura permite ficar com uma ideia muito mais clara do que cada função faz sem obrigar ao estudo do corpo das funções. Como se vê pelo exemplo, é pragmática usual em Julia usar um diferente número de caracteres "`#`" para indicar a relevância do comentário.

É importante que nos habituemos a documentar as nossas definições, mas convém salientar que a documentação em excesso também tem desvantagens:

- O programa Julia deve ser suficientemente claro para que um ser humano o consiga perceber. É sempre preferível perder mais tempo a tornar o programa claro do que a escrever documentação que o explique.
- Documentação que não está de acordo com o programa é pior que não ter documentação.
- É frequente termos de modificar os nossos programas para os adaptar a novos fins. Quanto mais documentação existir, mais documentação é necessário alterar para a pôr de acordo com as alterações que tivermos feito ao programa.

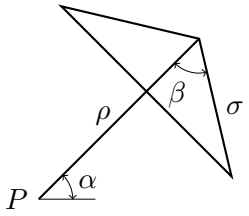
Por estes motivos, devemos esforçar-nos por escrever programas da forma mais clara que nos for possível e, ao mesmo tempo, providenciar documentação sucinta e útil: a documentação não deve dizer aquilo que é óbvio a partir da leitura do programa.

Exercício 2.11.1 Considere o desenho de uma seta com origem no ponto P, comprimento ρ, inclinação α, ângulo de abertura β e comprimento da "farpa" σ, tal como se representa em seguida:

[7] Na verdade, o programa é tão simples que não deveria necessitar de tanta documentação.

```
#### Desenho de colunas doricas

### O desenho de uma coluna dorica divide-se no desenho do
### fuste, do coxim e do abaco. A cada uma destas partes
### corresponde uma funcao independente.

"
Desenha o fuste de uma coluna dorica.

    p: coordenadas do centro da base da coluna,
    a_fuste: altura do fuste,
    r_base: raio da base do fuste,
    r_topo: raio do topo do fuste.
"
fuste(p, a_fuste, r_base, r_topo) =
  polygon(p + vxy(-r_topo, a_fuste),
          p + vxy(-r_base, 0),
          p + vxy(r_base, 0),
          p + vxy(r_topo, a_fuste))

"
Desenha o coxim de uma coluna dorica.

    p: coordenadas do centro da base do coxim,
    a_coxim: altura do coxim,
    r_base: raio da base do coxim,
    r_topo: raio do topo do coxim.
"
coxim(p, a_coxim, r_base, r_topo) =
  polygon(p + vxy(-r_base, 0),
          p + vxy(-r_topo, a_coxim),
          p + vxy(r_topo, a_coxim),
          p + vxy(r_base, 0))

"
Desenha o abaco de uma coluna dorica.

    p: coordenadas do centro da base da coluna,
    a_abaco: altura do abaco,
    l_abaco: largura do abaco.
"
abaco(p, a_abaco, l_abaco) =
  rectangle(p + vxy(l_abaco/-2, 0),
            p + vxy(l_abaco/2, a_abaco))

"
Desenha uma coluna dorica composta por fuste, coxim e abaco.

    p: coordenadas do centro da base da coluna,
    a_fuste: altura do fuste,
    r_base_fuste: raio da base do fuste,
    r_base_coxim: raio da base do coxim = raio do topo do fuste,
    a_coxim: altura do coxim,
    a_abaco: altura do abaco,
    l_abaco: largura do abaco = 2*raio do topo do coxim.
"
coluna(p, a_fuste, r_base_fuste, a_coxim, r_base_coxim, a_abaco, l_abaco) =
  begin
    fuste(p, a_fuste, r_base_fuste, r_base_coxim)
    coxim(p + vxy(0, a_fuste), a_coxim, r_base_coxim, l_abaco/2)
    abaco(p + vxy(0, a_fuste + a_coxim), a_abaco, l_abaco)
  end
```

Figura 2.20: Um programa Julia documentado.

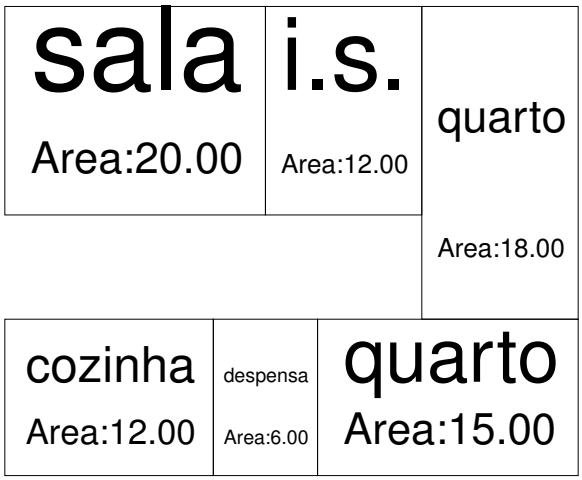


Defina uma função denominada `seta` que, a partir dos parâmetros P, ρ, α, σ e β, constrói a seta correspondente.

Exercício 2.11.2 Considere o desenho de uma habitação composta apenas por divisões rectangulares. Pretende-se que defina a função `divisao_rectangular` que recebe como parâmetros a posição do canto inferior esquerdo da divisão, o comprimento e a largura da divisão e um texto a descrever a função dessa divisão na habitação. Com esses valores a função deverá construir o rectângulo correspondente e deve colocar no interior desse rectângulo duas linhas de texto, a primeira com a função da divisão e a segunda com a área da divisão. Por exemplo, a sequência de invocações

```
divisao_rectangular(xy(0, 0), 4, 3, "cozinha")
divisao_rectangular(xy(4, 0), 2, 3, "despensa")
divisao_rectangular(xy(6, 0), 5, 3, "quarto")
divisao_rectangular(xy(0, 5), 5, 4, "sala")
divisao_rectangular(xy(5, 5), 3, 4, "i.s.")
divisao_rectangular(xy(8, 3), 3, 6, "quarto")
```

produz, como resultado, a habitação que se apresenta de seguida:



2.12 Depuração

Como sabemos, *errare humanum est*. O erro faz parte do nosso dia-a-dia e, por isso, em geral, sabemos lidar com ele. Já o mesmo não se passa com

as linguagens de programação. Qualquer erro num programa tem como consequência que o programa tem um comportamento diferente daquele que era esperado.

Uma vez que é fácil cometer erros, deve também ser fácil detectá-los e corrigi-los. A actividade de detecção e correcção de erros denomina-se *depuração* e diferentes linguagens de programação providenciam diferentes mecanismos para a realização dessa actividade. Neste domínio, como iremos ver, o Julia está particularmente bem apetrechado.

Em termos gerais, os erros num programa podem classificar-se em *erros sintáticos* e *erros semânticos*.

2.12.1 Erros Sintáticos

Os erros sintáticos ocorrem quando escrevemos frases que não obedecem à gramática da linguagem. Como exemplo prático, imaginemos que pretendíamos definir uma função que criava uma única coluna dórica, que iremos designar de coluna *standard* e que tem sempre as mesmas dimensões, não necessitando de quaisquer outros parâmetros. Uma possibilidade para a definição desta função será:

```
coluna_standard()
  coluna(xy(0, 0), 9, 0.5, 0.4, 0.3, 0.3, 0.5)
```

No entanto, se avaliarmos aquela definição, o Julia irá apresentar um erro, avisando-nos de que algo está errado. O erro de que o Julia nos está a avisar é de que a forma que lhe demos para avaliar não obedece à sintaxe esperada e, de facto, uma observação atenta da forma anterior mostra que não seguimos a sintaxe exigida para uma definição que, tal como discutimos na secção 1.4, era a seguinte:

```
nome(parâmetro_1, ..., parâmetro_n) =
  corpo
```

Como podemos verificar, esquecemo-nos de terminar a primeira linha com o carácter = e é desse erro que o Julia se está a queixar.

Há vários outros tipos de erros sintáticos que o Julia é capaz de detectar e que serão apresentados à medida que os formos discutindo. O importante, no entanto, não é saber quais são os erros sintáticos detectáveis pelo Julia, mas antes saber que o Julia é capaz de verificar o que escrevemos e fazer a detecção de erros sintáticos.

2.12.2 Erros Semânticos

Os erros semânticos são muito diferentes dos sintáticos. Um erro semântico não é um erro na escrita de uma "frase" da linguagem, mas sim um erro no significado dessa frase. Dito de outra forma, um erro semântico

ocorre quando escrevemos uma frase que julgamos ter um significado e, na verdade, ela tem outro.

Em geral, os erros semânticos apenas são detectáveis durante a invocação das funções que os contêm. Parte dos erros semânticos é detectável pelo avaliador de Julia, mas há inúmeros erros cuja detecção só pode ser feita pelo próprio programador.

Como exemplo de erro semântico consideremos uma operação sem significado, como seja a soma de um número com uma *string*:

```
julia> 1 + "dois"
ERROR: MethodError: no method matching +(::Int64, ::String)
```

Como se pode ver, o erro é explicado na mensagem que indica que o segundo argumento é do tipo *string*, quando devia ser um número. Neste exemplo, o erro é suficientemente óbvio para o conseguirmos detectar imediatamente. No entanto, no caso de programas mais complexos, convém ter presente que o processo de identificação de erros pode ser moroso e frustrante. É também um facto que quanto mais experiência temos na detecção de erros, mais rapidamente o conseguimos fazer.

2.13 Modelação Tridimensional

Como vimos na secção anterior, o Khepri disponibiliza um conjunto de operações de desenho (linhas, rectângulos, círculos, etc) que nos permitem facilmente criar representações bidimensionais de objectos, como sejam plantas, alçados e cortes.

Embora até este momento apenas tenhamos utilizado essencialmente as capacidades de desenho bi-dimensional do Khepri , é possível irmos mais longe, entrando no que se denomina por *modelação tridimensional*. Esta modelação visa a representação gráfica de linhas, superfícies e volumes no espaço tridimensional.

Nesta secção iremos estudar as operações do Khepri que nos permitem modelar directamente os objectos tridimensionais.

2.13.1 Sólidos Pré-Definidos

O Khepri disponibiliza um conjunto de operações pré-definidas que constroem um sólido a partir das suas coordenadas tridimensionais. Embora as operações pré-definidas apenas permitam construir um conjunto limitado de sólidos, esse conjunto é suficiente para a elaboração de modelos sofisticados.

As operações pré-definidas para criação de sólidos permitem construir paralelipípedos (função `box`), cilindros (função `cylinder`), cones (função `cone`), esferas (função `sphere`), toros (função `torus`) e pirâmides (função

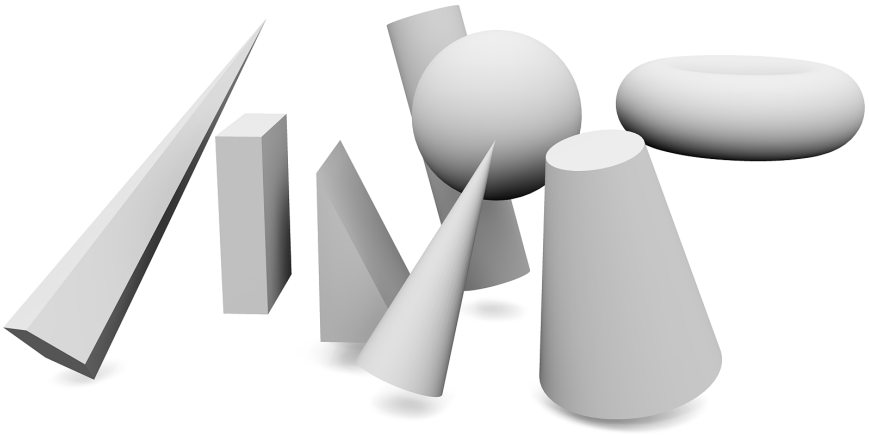


Figura 2.21: Sólidos primitivos em Khepri.

`regular_pyramid`). Cada uma destas funções aceita vários argumentos opcionais que permitem construir sólidos de diferentes maneiras. A Figura 2.21 mostra um conjunto de sólidos construídos pela avaliação das seguintes expressões:

```
box(xyz(2, 1, 1), xyz(3, 4, 5))
cone(xyz(6, 0, 0), 1, xyz(8, 1, 5))
cone_frustum(xyz(11, 1, 0), 2, xyz(10, 0, 5), 1)
sphere(xyz(8, 4, 5), 2)
cylinder(xyz(8, 7, 0), 1, xyz(6, 8, 7))
regular_pyramid(5, xyz(-2, 1, 0), 1, 0, xyz(2, 7, 7))
torus(xyz(14, 6, 5), 2, 1)
```

É interessante vermos que algumas das obras mais importantes da história da Arquitectura podem ser modeladas por aplicações directas destas operações. A Grande Pirâmide de Gizé, ilustrada na Figura 2.22, foi construída há mais de quarenta e cinco séculos e é um exemplo quase perfeito de uma pirâmide quadrangular. Na sua forma original a Grande Pirâmide tinha uma base quadrada com 230 metros de lado e uma altura de 147 metros e foi o edifício mais alto do mundo até à construção da torre Eiffel. Muitas outras pirâmides Egípcias possuem proporções semelhantes, o que nos permite modelar muitas destas construções através da seguinte função:

```
piramide_egipcia(p, lado, altura) =
  regular_pyramid(4, p, lado/2, 0, altura, false)
```

O caso particular da Grande Pirâmide será então:

```
piramide_egipcia(xyz(0, 0, 0), 230, 147)
```

Ao contrário do que acontece com a Grande Pirâmide, não há muitas obras que possam ser modeladas com uma só primitiva geométrica. No en-

Figura 2.22: A Grande Pirâmide de Gizé. Fotografia de Nina Aldin Thune.

tanto, é possível criar estruturas bastante mais complexas através da combinação destas primitivas.

Como exemplo, consideremos a cruzeta representada na Figura 2.23, constituida por seis troncos de cone idênticos, com a base centrada num mesmo ponto e com os topos posicionados ao longo dos eixos coordenados.

Para a modelação deste sólido vamos parametrizar o posicionamento p da base dos troncos de cone, bem como as dimensões de cada tronco de cone em termos dos raios da base r_b e do topo r_t e ainda do comprimento c. Assim, temos:

```
cruzeta(p, rb, rt, c) =
  begin
    cone_frustum(p, rb, p + vx(c), rt)
    cone_frustum(p, rb, p + vy(c), rt)
    cone_frustum(p, rb, p + vz(c), rt)
    cone_frustum(p, rb, p - vx(c), rt)
    cone_frustum(p, rb, p - vy(c), rt)
    cone_frustum(p, rb, p - vz(c), rt)
  end
```

Exercício 2.13.1 Empregando a função `cruzeta` responsável pela criação do modelo apresentado na Figura 2.23, determine valores aproximados para os parâmetros de modo a conseguir reproduzir os seguintes modelos:

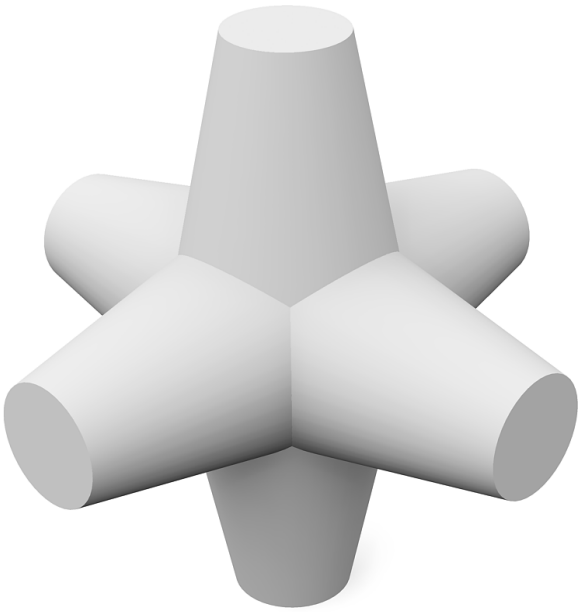
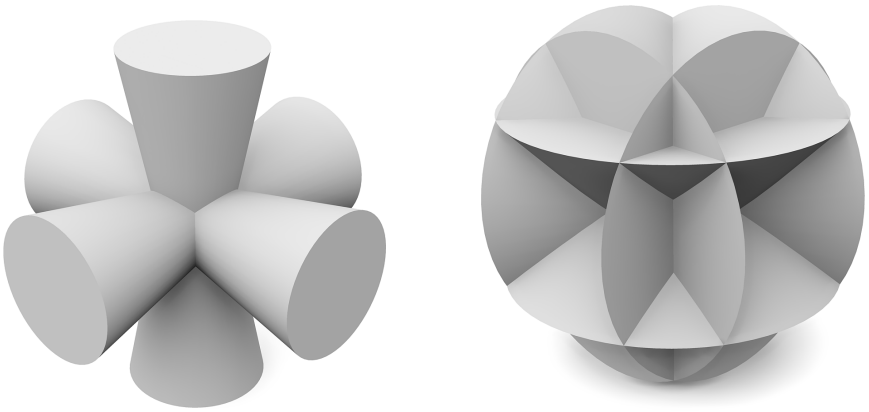
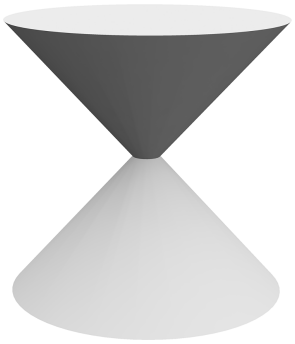


Figura 2.23: Uma sobreposição ortogonal de troncos de cone.



Exercício 2.13.2 Usando a função `cone_frustum`, defina a função `ampulheta` que, a partir do centro da base da ampulheta, do raio da base da ampulheta, do raio do estrangulamento da ampulheta e da altura da ampulheta, constrói um modelo de uma ampulheta semelhante ao que se apresenta em seguida:

Figura 2.24: Três troncos de pirâmide construídos com diferentes ângulos de rotação da base. Da esquerda para a direita o ângulo de rotação é 0, $\frac{\pi}{4}$ e $\frac{\pi}{2}$.



Semelhante ao tronco de cone, mas com uma base poligonal, temos o tronco de pirâmide. O tronco de pirâmide pode ser criado usando a função `regular_pyramid_frustum`. Para isso, temos de especificar o número de lados, o centro da base, o raio da circunferência que circunscreve a base, o ângulo de rotação da base, o centro do topo e o raio da circunferência que circunscreve o topo:

A Figura 2.24 apresenta o efeito do ângulo de rotação e foi gerada pela avaliação das seguintes expressões:

```
regular_pyramid_frustum(3, xyz(0, 0, 0), 2, 0, xyz(0, 0, 1), 1)
regular_pyramid_frustum(3, xyz(8, 0, 0), 2, pi/4, xyz(8, 0, 1), 1)
regular_pyramid_frustum(3, xyz(16, 0, 0), 2, pi/2, xyz(16, 0, 1), 1)
```

Para vermos um exemplo mais arquitectónico, consideremos a pirâmide *romboidal* de Dashur, ilustrada na Figura 2.25, que se caracteriza por as suas faces terem um ângulo de inclinação que varia abruptamente de $55°$ para $43°$, presumindo-se que tal tenha acontecido para evitar o desmoronamento da mesma devido à elevada inclinação inicial. Esta pirâmide tem uma base com 188.6 metros de largura e uma altura de 101.1 metros e é decomponível em dois sólidos geométricos, o primeiro um tronco de pirâmide quadrangular, em cima do qual assenta uma pirâmide quadrangular.

Para generalizarmos a modelação deste tipo de construções podemos considerar o esquema ilustrado na Figura 2.26, onde mostramos a pirâmide

Figura 2.25: A pirâmide romboidal de Dashur. Fotografia de Ivrienen.

em corte. A partir deste esquema é fácil vermos que $h_0 + h_1 = h$ e que $l_0 + l_1 = l$. Por outro lado, temos as definições trigonométricas

$$\tan \alpha_0 = \frac{h_0}{l_0} \qquad \tan \alpha_1 = \frac{h_1}{l_1}$$

Substituindo, obtemos

$$l_0 = \frac{h - l \tan \alpha_1}{\tan \alpha_0 - \tan \alpha_1}$$

Transcrevendo estes cálculos para Julia, temos:

```
piramide_romboide(p, l, h, a0, a1) =
  let l0 = (h - l*tan(a1))/(tan(a0) - tan(a1)),
      l1 = l - l0,
      h0 = l0*tan(a0),
      h1 = h - h0
    regular_pyramid_frustum(4, p, l, 0, h0, l1)
    regular_pyramid(4, p + vz(h0), l1, 0, h1)
  end
```

A Figura 2.27 mostra, à direita, um modelo da pirâmide de Dashur, criada através da expressão:

```
piramide_romboide(xyz(0, 0, 0), 186.6/2, 101.1, radianos(55), radianos(43))
```

Nessa imagem, as pirâmides mais à esquerda foram criadas com as seguintes variações:

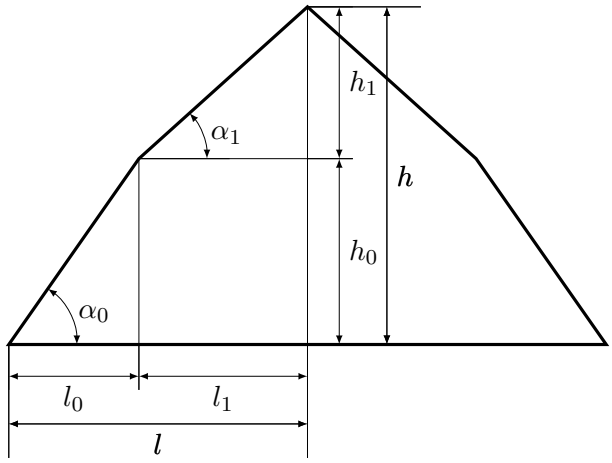


Figura 2.26: Esquematização da pirâmide romboidal.

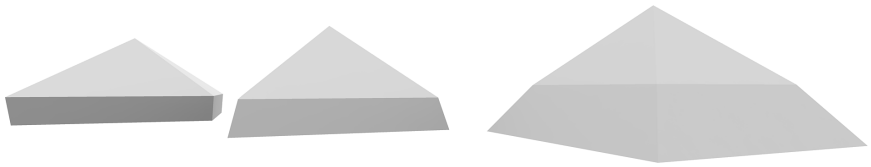


Figura 2.27: Três pirâmides rombóides usando diferentes parâmetros.

```
piramide_romboide(xyz(300, 0, 0), 186.6/2, 101.1, radianos(75), radianos(40))
piramide_romboide(xyz(600, 0, 0), 186.6/2, 101.1, radianos(95), radianos(37))
```

Exercício 2.13.3 Um obelisco é um monumento caracterizado por ter a forma de uma pirâmide rombóide. O Monumento a Washington, ilustrado na Figura 2.28, é um exemplo moderno de um obelisco de enormes dimensões, cujos parâmetros definidores (relativamente ao esquema representado na Figura 2.26) são um tronco de pirâmide com uma largura de $2l = 16.8$ metros na base e $2l_1 = 10.5$ metros no topo, uma altura total $h = 169.3$ metros e uma altura da pirâmide superior de $h_1 = 16.9$ metros.

Defina a função `obelisco` que, a partir do centro da base do obelisco, da largura da base, da altura total, da largura do topo e da altura da pirâmide, cria um obelisco.

Exercício 2.13.4 Um obelisco *perfeito* obedece a um conjunto de proporções em que a altura da pirâmide é igual à largura da base que, por sua vez, é um décimo da altura total. Finalmente, a largura do topo tem de ser dois terços da largura da base. Tendo estas proporções em conta, defina a função `obelisco_perfeito` que, a partir apenas do centro da base e da altura total do obelisco, cria um obelisco perfeito.

Exercício 2.13.5 Usando a função `regular_pyramid_frustum`, defina uma função denominada `prisma` que cria um sólido prismático regular. A fun-

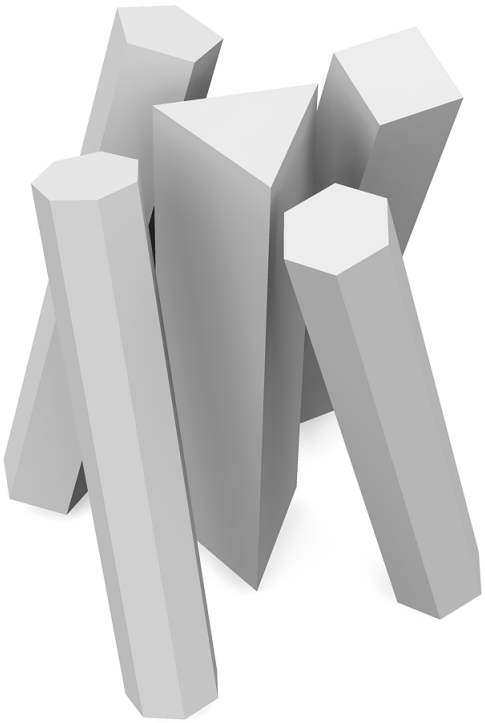
Figura 2.28: O Monumento a Washington. Fotografia de David Iliff.

ção deverá receber o número de lados do prisma, as coordenadas tridimensionais do centro da base do prisma, a distância do centro da base a cada vértice, o ângulo de rotação da base do prisma e as coordenadas tridimensionais do centro do topo do prisma.

A título de exemplo, considere as expressões

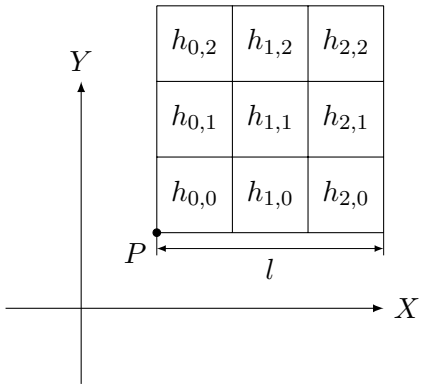
```
prisma(3, xyz(0, 0, 0), 0.4, 0, xyz(0, 0, 5))
prisma(5, xyz(-2, 0, 0), 0.4, 0, xyz(-1, 1, 5))
prisma(4, xyz(0, 2, 0), 0.4, 0, xyz(1, 1, 5))
prisma(6, xyz(2, 0, 0), 0.4, 0, xyz(1, -1, 5))
prisma(7, xyz(0, -2, 0), 0.4, 0, xyz(-1, -1, 5))
```

cuja avaliação produz a imagem seguinte:



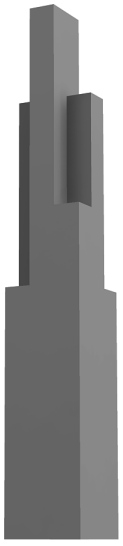
Exercício 2.13.6 A *Sears Tower* (actualmente denominada *Willis Tower*) apresentada na Figura 2.29 foi, durante muitos anos, o mais alto edifício do mundo. Esta torre é constituida por nove prismas de base quadrada de diferentes alturas $h_{i,j}$ interligados entre si. Em planta, estes nove blocos definem um quadrado de lado l, tal como se apresenta no esquema seguinte:

Figura 2.29: A *Sears Tower*, em Chicago.



Usando a função `prisma` definida no exercício anterior, defina uma função denominada `torre_sears` capaz de criar edifícios semelhantes à *Sears Tower*. A função deverá receber as coordenadas tridimensionais do canto P da base da torre, a largura da base l e ainda os nove parâmetros $h_{0,0}, h_{0,1}, \ldots, h_{2,2}$ que definem a altura de cada prisma.

A *Sears Tower* real possui, como parâmetros, $l = 68.7m$, $h_{0,1} = h_{1,1} = 442m$, $h_{1,0} = h_{2,1} = h_{1,2} = 368m$, $h_{0,0} = h_{2,2} = 270m$, e $h_{0,2} = h_{2,0} = 205m$, tal como se apresenta na seguinte imagem:



Para além das primitivas geométricas já discutidas, existe ainda uma outra que permite criar *cubóides*, i.e., sólidos constituídos por seis faces mas cuja forma não é necessariamente cúbica. Um cubóide define-se pelos seus oito vértices, divididos em dois grupos de quatro, para as faces da base e do topo do cubóide (descritos no sentido anti-horário). As seguintes expressões criam os três cubóides apresentados na Figura 2.30:

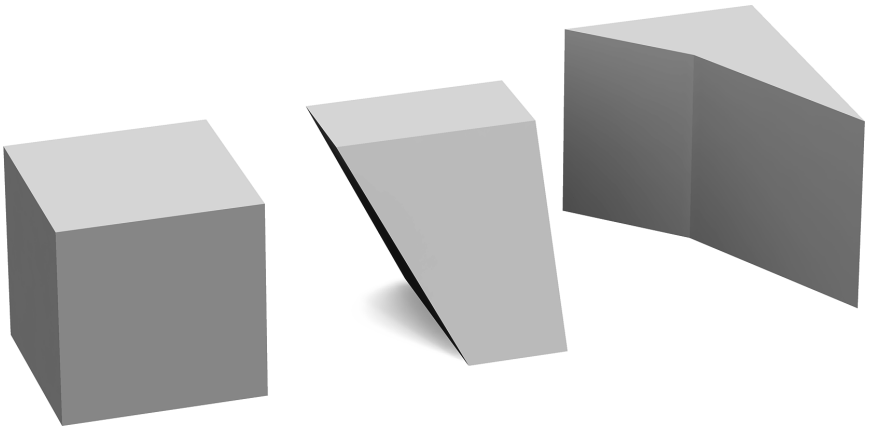


Figura 2.30: Três cubóides criados a partir de diferentes vértices.

```
cuboid(xyz(0, 0, 0), xyz(2, 0, 0), xyz(2, 2, 0), xyz(0, 2, 0),
       xyz(0, 0, 2), xyz(2, 0, 2), xyz(2, 2, 2), xyz(0, 2, 2))

cuboid(xyz(4, 0, 0), xyz(5, 0, 0), xyz(5, 2, 0), xyz(4, 2, 0),
       xyz(3, 1, 2), xyz(5, 1, 2), xyz(5, 2, 2), xyz(3, 2, 2))

cuboid(xyz(7, 2, 0), xyz(8, 0, 0), xyz(8, 3, 0), xyz(6, 3, 0),
       xyz(7, 2, 2), xyz(8, 0, 2), xyz(8, 3, 2), xyz(6, 3, 2))
```

O *John Hancock Center*, ilustrado na Figura 2.31 é um bom exemplo de um edifício cuja geometria se pode definir usando um cubóide. De facto, este edifício tem a forma de um tronco, com uma base e topo rectangulares. Para o modelarmos, podemos começar por definir a função `tronco_rectangular`, parameterizada pelo centro da base do edifício P, pelo comprimento c_b e largura l_b da base, pelo comprimento c_t e largura l_t do topo, e, finalmente, pela altura h. Para a criação do sólido propriamente dito, a operação `cuboid` torna-se particularmente útil, bastando-nos determinar as posições dos vértices de sólido relativamente à posição P.

```
tronco_rectangular(p, cb, lb, ct, lt, h) =
  cuboid(p + vxyz(-cb/2, -lb/2, 0),
         p + vxyz(+cb/2, -lb/2, 0),
         p + vxyz(+cb/2, +lb/2, 0),
         p + vxyz(-cb/2, +lb/2, 0),
         p + vxyz(-ct/2, -lt/2, h),
         p + vxyz(+ct/2, -lt/2, h),
         p + vxyz(+ct/2, +lt/2, h),
         p + vxyz(-ct/2, +lt/2, h))
```

Usando esta função podemos criar edifícios inspirados na forma do *John Hancock Center*, tal como ilustramos na Figura 2.32.

Exercício 2.13.7 O *pilone* é um elemento característico da arquitectura egípcia, de que podemos ver uma ilustração na Figura 2.33. Trata-se de um pór-

Figura 2.31: O *John Hancock Center*, em Chicaco.

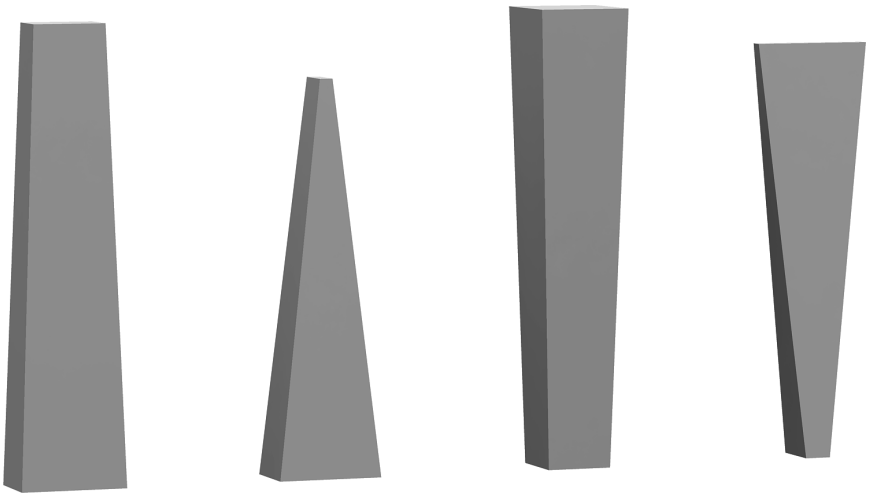


Figura 2.32: Edifícios cubóides inspirados na forma do *John Hancock Center*.

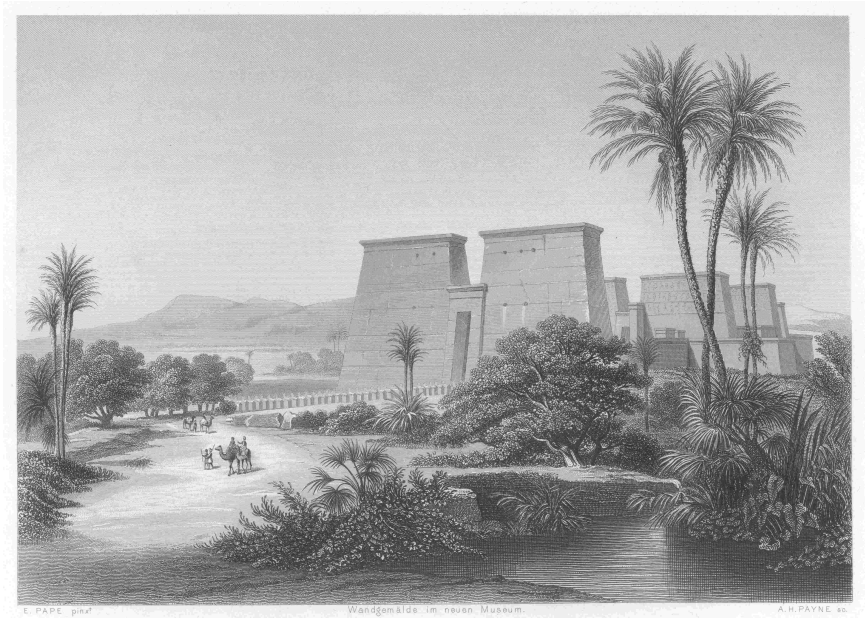
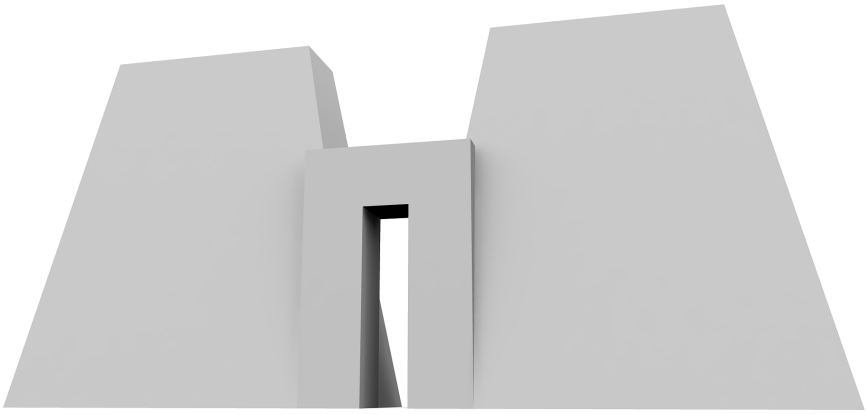


Figura 2.33: Pilones do templo de Karnak. Ilustração de Albert Henry Payne.

tico monumental ladeado por duas torres idênticas. Cada torre é de forma cubóide, com a base e o topo rectangulares e as restantes quatro faces trapezoidais. Defina uma função convenientemente parametrizada capaz de criar uma representação simplificada de um pilone, semelhante ao que se apresenta na seguinte imagem:



2.14 Coordenadas Cilíndricas

Vimos nas secções anteriores alguns exemplos da utilização dos sistemas de coordenadas rectangulares e polares. Ficou também claro que uma escolha

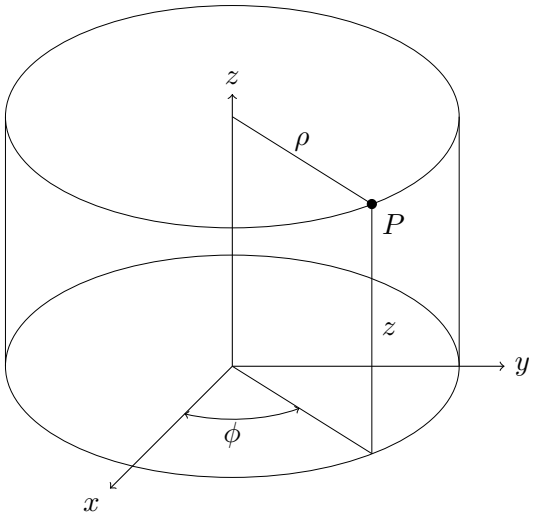


Figura 2.34: Coordenadas cilíndricas.

judiciosa do sistema de coordenadas pode simplificar bastante a solução de um problema geométrico.

Para a modelação tridimensional, para além das coordenadas rectangulares e polares, é ainda usual empregarem-se dois outros sistemas de coordenadas que iremos ver de seguida: as *coordenadas cilíndricas* e as *coordenadas esféricas*.

Tal como podemos verificar na Figura 2.34, um ponto, em coordenadas cilíndricas, caracteriza-se por um raio ρ assente no plano $z = 0$, um ângulo ϕ que esse raio faz com o eixo X, e por uma cota z. É fácil de ver que o raio e o ângulo correspondem às coordenadas polares da projecção do ponto no plano $z = 0$.

A partir da Figura 2.34 é fácil vermos que, dado um ponto (ρ, ϕ, z) em coordenadas cilíndricas, o mesmo ponto em coordenadas rectangulares é

$$(\rho \cos \phi, \rho \sin \phi, z)$$

De igual modo, dado um ponto (x, y, z) em coordenadas rectangulares, o mesmo ponto em coordenadas cilíndricas é

$$(\sqrt{x^2 + y^2}, \operatorname{atan} \frac{y}{x}, z)$$

Estas equivalências são asseguradas pelo constructor de coordenadas cilíndricas `cyl`. Embora esta função esteja pré-definida em Khepri, não é difícil imaginar que esteja definida como

```
cyl(ro, fi, z) =
  xyz(ro*cos(fi), ro*sin(fi), z)
```

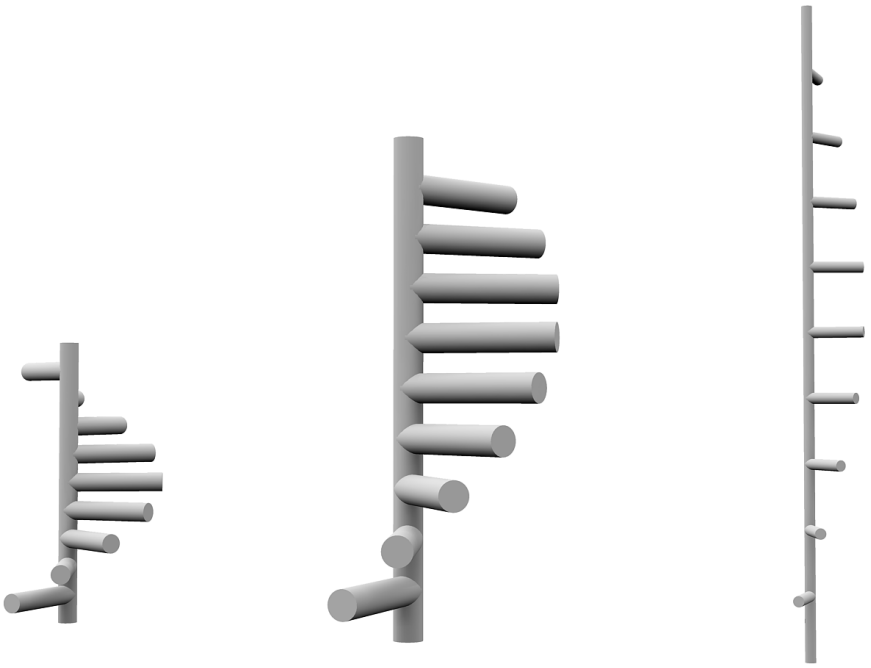
No caso de pretendermos simplesmente somar a um ponto um deslo-

camento em coordenadas cilíndricas, podemos definir a função `vcyl` para criar o vector de deslocamento:

```
vcyl(ro, fi, z) =
  vxyz(ro*cos(fi), ro*sin(fi), z)
```

Exercício 2.14.1 Defina os selectores `cyl_rho`, `cyl_phi` e `cyl_z` que devolvem, respectivamente, os componentes ρ, ϕ e z de uma posição construída pelo construtor `cyl`.

Exercício 2.14.2 Defina uma função `escadas` capaz de construir uma escada em hélice. A imagem seguinte mostra três exemplos destas escadas.



Como se pode ver pela imagem anterior, a escada é constituída por um cilindro central no qual se apoiam 10 degraus cilíndricos. Para facilitar a experimentação considere que o cilindro central é de raio r. Os degraus são iguais ao cilindro central, possuem 10 raios de comprimento e estão dispostos a alturas progressivamente crescentes. Cada degrau está a uma altura h em relação ao degrau anterior e, visto em planta, faz um ângulo α com o degrau anterior.

A função `escada` deverá receber as coordenadas do centro da base do cilindro central, o raio r, a altura h e o ângulo α. A título de exemplo, considere que as três escadas anteriores foram construídas pelas seguintes invocações:

```
escada(xyz(0, 0, 0), 1.0, 3, pi/6)
escada(xyz(0, 40, 0), 1.5, 5, pi/9)
escada(xyz(0, 80, 0), 0.5, 6, pi/8)
```

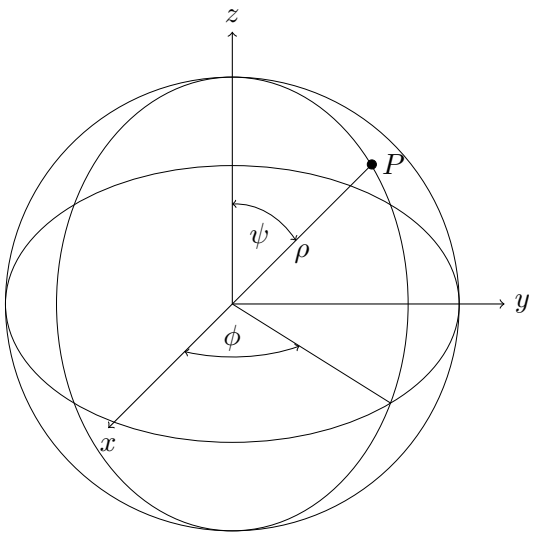


Figura 2.35: Coordenadas Esféricas

2.15 Coordenadas Esféricas

Tal como podemos ver na Figura 2.35, um ponto, em coordenadas esféricas, caracteriza-se pelo comprimento ρ do raio vector, por um ângulo ϕ (denominado *longitude* ou *azimute*) que a projecção desse vector no plano $z = 0$ faz com o eixo X e por um ângulo ψ (denominado *colatitude*[8], *zénite* ou *ângulo polar*) que o vector faz com o eixo Z.

Dado um ponto (ρ, ϕ, ψ) em coordenadas esféricas, o mesmo ponto em coordenadas rectangulares é

$$(\rho \sin\psi \cos\phi, \rho \sin\psi \sin\phi, \rho \cos\psi)$$

De igual modo, dado um ponto (x, y, z) em coordenadas rectangulares, o mesmo ponto em coordenadas esféricas é

$$(\sqrt{x^2+y^2+z^2}, \operatorname{atan}\frac{y}{x}, \operatorname{atan}\frac{\sqrt{x^2+y^2}}{z})$$

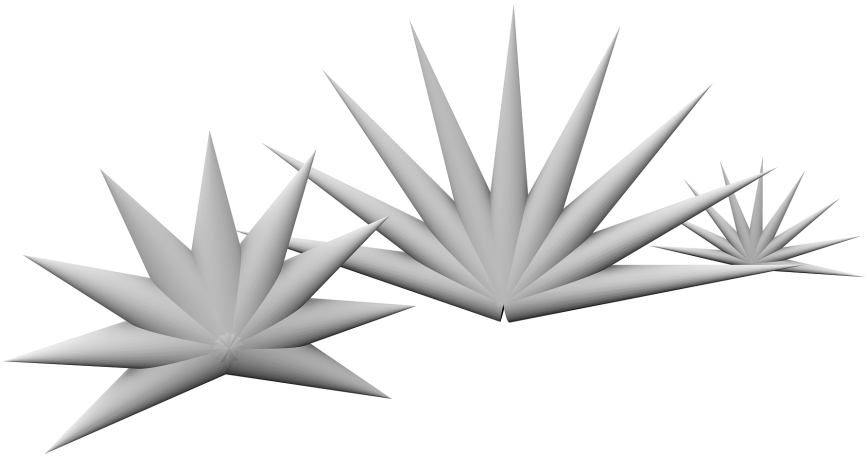
Tal como acontece com as coordenadas cilíndricas, os construtores de coordenadas esféricas `sph` e `vsph` encontram-se pré-definidos mas não é difícil imaginar que essas funções tenham as seguintes definições:

[8] A colatitude é o ângulo complementar à latitude, i.e., a diferença entre o pólo ($\frac{\pi}{2}$) e a latitude.

```
sph(ro, fi, psi) =
  xyz(ro*sin(psi)*cos(fi), ro*sin(psi)*sin(fi), ro*cos(psi))

vsph(ro, fi, psi) =
  vxyz(ro*sin(psi)*cos(fi), ro*sin(psi)*sin(fi), ro*cos(psi))
```

Exercício 2.15.1 Defina os selectores `sph_rho`, `sph_phi` e `sph_psi` que devolvem, respectivamente, os componentes ρ, ϕ e ψ de uma posição construída pelo construtor `sph`.

Exercício 2.15.2 O corte de cabelo de estilo *Moicano* foi muito utilizado no período *punk*. Ele consiste em fixar os cabelos em bicos que se dispoem em forma de leque ou crista, tal como se esquematiza na seguinte imagem.



Defina a função `moicano`, de parâmetros P, r, c, ϕ e Δ_ψ, que constrói 9 cones de comprimento c e raio da base r, todos com a base centrada num mesmo ponto P e com os eixos inclinados uns em relação aos outros por um ângulo Δ_ψ, e dispostos ao longo de um plano que faz um ângulo ϕ com o plano XZ.

A título de exemplo, considere que a figura anterior foi produzida pelas seguintes invocações:

```
moicano(xyz(30, 0, 0), 1.5, 10, pi/2, pi/6)
moicano(xyz(30, 15, 0), 1.0, 15, pi/3, pi/9)
moicano(xyz(30, 30, 0), 0.5, 6, pi/4, pi/8)
```

2.16 Modelação de Colunas Dóricas

A modelação tri-dimensional tem a virtude de nos permitir criar entidades geométricas muito mais realistas do que meros aglomerados de linhas a representarem vistas dessas entidades. A título de exemplo, reconsideremos a coluna Dórica que apresentámos na secção 2.9. Nessa secção desenvolvemos um conjunto de funções cuja invocação criava uma vista frontal

dos componentes da coluna Dórica. Apesar dessas vistas serem úteis, é ainda mais útil poder modelar directamente a coluna como uma entidade tri-dimensional.

Nesta secção vamos empregar algumas das operações mais relevantes para a modelação tri-dimensional de colunas, em particular, a criação de troncos de cone para modelar o fuste e o coxim e a criação de paralelipípedos para modelar o ábaco.

Anteriormente, as nossas "colunas" estavam dispostas no plano XY, com as colunas a "crescer" ao longo do eixo Y. Agora, será apenas a base das colunas que ficará assente no plano XY: o corpo das colunas irá desenvolver-se ao longo do eixo Z. Embora fosse possível empregar outro arranjo dos eixos do sistema de coordenadas, este é aquele que é mais próximo da realidade.

À semelhança de inúmeras outras operações do Khepri, cada uma das operações de modelação de sólidos do Khepri permite alguns modos diferentes de invocação. No caso da operação de modelação de troncos de cone—`cone_frustum`—o modo que nos é mais conveniente é aquele em que a operação recebe as coordenadas do centro da base e o raio dessa base, a altura e, finalmente, o raio do topo.

Tendo isto em conta, podemos redefinir a operação que constrói o fuste tal como se segue:

```
fuste(p, a_fuste, r_base, r_topo) =
  cone_frustum(p, r_base, a_fuste, r_topo)
```

Do mesmo modo, a operação que constrói o coxim ficará com a forma:

```
coxim(p, a_coxim, r_base, r_topo) =
  cone_frustum(p, r_base, a_coxim, r_topo)
```

Finalmente, no que diz respeito ao ábaco—o paralelipípedo que é colocado no topo da coluna—temos várias maneiras de o especificarmos. Uma, consiste em indicar os dois cantos do paralelipípedo. Outra, consiste em indicar apenas um dos cantos, seguido das dimensões do paralelipípedo. Para este exemplo, vamos seguir segunda alternativa:

```
abaco(p, a_abaco, l_abaco) =
  box(p + vxyz(-l_abaco/2, -l_abaco/2, 0), l_abaco, l_abaco, a_abaco)
```

Exercício 2.16.1 Implemente a função `abaco` mas empregando a criação de um paralelipípedo baseada nos dois cantos.

Finalmente, falta-nos implementar a função `coluna` que, à semelhança do que fazia no caso bi-dimensional, invoca sucessivamente as funções `fuste`, `coxim` e `abaco` mas, agora, elevando progressivamente a coordenada z:

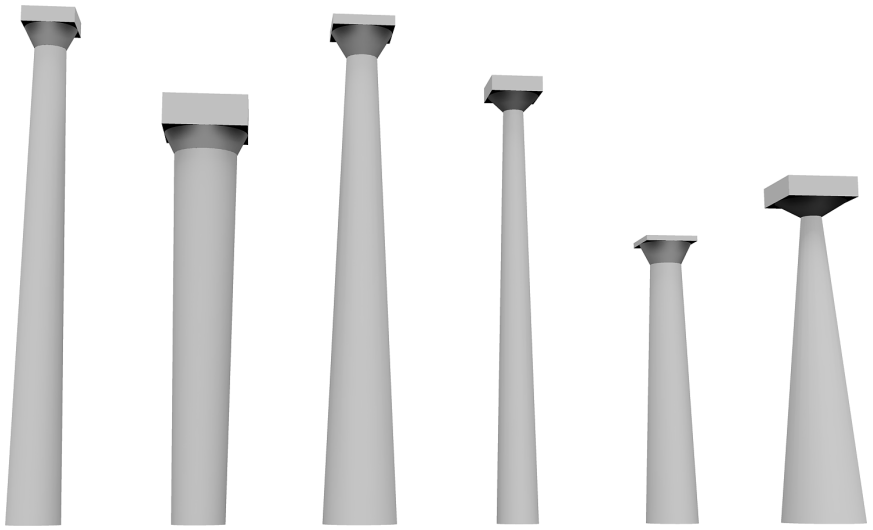


Figura 2.36: Modelação tri-dimensional das variações de colunas dóricas.

```
coluna(p,
       a_fuste, r_base_fuste,
       a_coxim, r_base_coxim,
       a_abaco, l_abaco) =
  begin
    fuste(p, a_fuste, r_base_fuste, r_base_coxim)
    coxim(p + vz(a_fuste), a_coxim, r_base_coxim, l_abaco/2)
    abaco(p + vz(a_fuste + a_coxim), a_abaco, l_abaco)
  end
```

Com estas redefinições, podemos agora repetir as colunas que desenhámos na secção 2.10 e que apresentámos na Figura 2.19, mas, agora, gerando uma imagem tridimensional dessas mesmas colunas, tal como apresentamos na Figura 2.36:

```
coluna(xyz(0, 0, 0), 9, 0.5, 0.4, 0.3, 0.3, 1.0)
coluna(xyz(3, 0, 0), 7, 0.5, 0.4, 0.6, 0.6, 1.6)
coluna(xyz(6, 0, 0), 9, 0.7, 0.5, 0.3, 0.2, 1.2)
coluna(xyz(9, 0, 0), 8, 0.4, 0.3, 0.2, 0.3, 1.0)
coluna(xyz(12, 0, 0), 5, 0.5, 0.4, 0.3, 0.1, 1.0)
coluna(xyz(15, 0, 0), 6, 0.8, 0.3, 0.2, 0.4, 1.4)
```

2.17 Proporções de Vitrúvio

A modelação de colunas dóricas que desenvolvemos na secção 2.16 permitenos facilmente construir colunas, bastando para isso indicarmos os valores dos parâmetros relevantes, como a altura e o raio da base do fuste, a altura e raio da base do coxim e a altura e largura do ábaco. Cada um destes

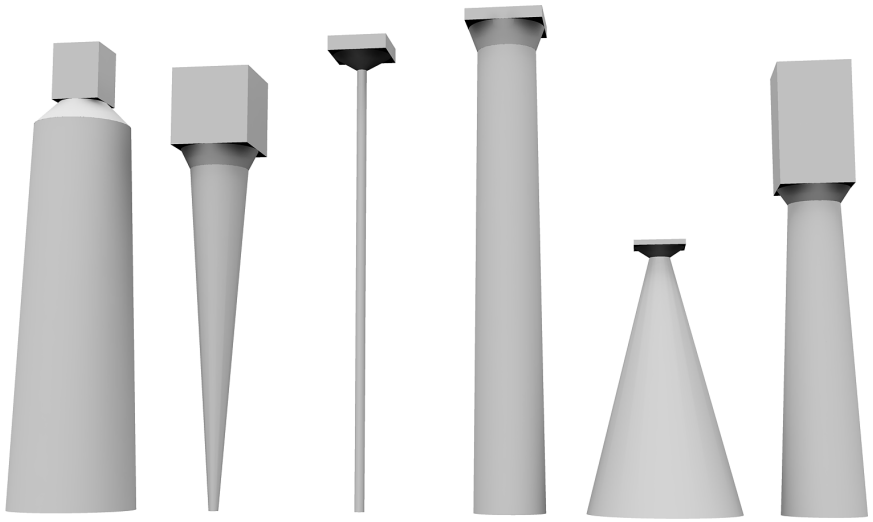


Figura 2.37: Perspectiva da modelação tri-dimensional de colunas cujos parâmetros foram escolhidos aleatoriamente. Apenas uma das colunas obedece aos *cânones* da Ordem Dórica.

parâmetros constitui um grau de liberdade que podemos fazer variar livremente.

Embora seja lógico pensar que quantos mais graus de liberdade tivermos mais flexível é a modelação, a verdade é que um número excessivo de parâmetros pode conduzir a modelos pouco realistas. Esse fenómeno é evidente na Figura 2.37 onde mostramos uma perspectiva de um conjunto de colunas cujos parâmetros foram escolhidos aleatoriamente.

Na verdade, de acordo com os *cânones* da Ordem Dórica, os diversos parâmetros que regulam a forma de uma coluna devem relacionar-se entre si segundo um conjunto de proporções bem definidas. Vitrúvio,[9] no seu famoso tratado de arquitectura, considera que essas proporções derivam das proporções do próprio ser humano:

> Uma vez que pretendiam erguer um templo com colunas mas não tinham conhecimento das proporções adequadas, [...] mediram o comprimento dum pé de um homem e viram que era um sexto da sua altura e deram à coluna uma proporção semelhante, i.e., fizeram a sua altura, incluindo o capitel, seis vezes a largura da coluna medida na base. Assim, a ordem Dórica obteve a sua proporção e a sua beleza, da figura masculina.

Mais concretamente, Vitrúvio caracteriza as colunas da Ordem Dórica em termos do conceito de *módulo*:

[9] Vitrúvio foi um escritor, arquitecto e engenheiro romano que viveu no século um antes de Cristo e autor do único tratado de arquitectura que sobreviveu a antiguidade.

- *A largura das colunas, na base, será de dois modulos e a sua altura, incluindo os capitéis, será de catorze.*

  Daqui se deduz que um módulo iguala o raio da base da coluna e que a altura da coluna deverá ser 14 vezes esse raio. Dito de outra forma, o raio da base da coluna deverá ser $\frac{1}{14}$ da altura da coluna.

- *A altura do capitel será de um módulo e a sua largura de dois módulos e um sexto.*

  Isto implica que a altura do coxim somada à do ábaco será um módulo, ou seja, igual ao raio da base da coluna e a largura do ábaco será de $2\frac{1}{6}$ módulos ou $\frac{13}{6}$ do raio. Juntamente com o facto de a altura da coluna ser de 14 módulos, implica ainda que a altura do fuste será de 13 vezes o raio.

- *Seja a altura do capitel dividida em três partes, das quais uma formará o ábaco com o seu cimáteo, o segundo o équino (coxim) com os seus aneis e o terceiro o pescoço.*

  Isto quer dizer que o ábaco tem uma altura de um terço de um modulo, ou seja $\frac{1}{3}$ do raio da base, e o coxim terá os restantes dois terços, ou seja, $\frac{2}{3}$ do raio da base.

Estas considerações levam-nos a poder determinar o valor de alguns dos parâmetros de desenho das colunas dóricas em termos do raio da base do fuste. Em termos de implementação, isso quer dizer que os parâmetros da função passam a ser nomes locais cuja definição é feita aplicando as proporções estabelecidas por Vitrúvio ao parâmetro `r_base_fuste`. A definição da função fica então:

```
coluna_dorica(p, r_base_fuste, r_base_coxim) =
  let a_fuste = 13*r_base_fuste,
      a_coxim = 2/3*r_base_fuste,
      a_abaco = 1/3*r_base_fuste,
      l_abaco = 13/6*r_base_fuste
    fuste(p, a_fuste, r_base_fuste, r_base_coxim)
    coxim(p + vz(a_fuste), a_coxim, r_base_coxim, l_abaco/2)
    abaco(p + vz(a_fuste + a_coxim), a_abaco, l_abaco)
  end
```

Usando esta função já é possível desenhar colunas que se aproximam mais do padrão dórico (tal como estabelecido por Vitrúvio). A Figura 2.38 apresenta as colunas desenhadas pelas seguintes invocações:

```
coluna_dorica(xyz(0, 0, 0), 0.3, 0.2)
coluna_dorica(xyz(3, 0, 0), 0.5, 0.3)
coluna_dorica(xyz(6, 0, 0), 0.4, 0.2)
coluna_dorica(xyz(9, 0, 0), 0.5, 0.4)
coluna_dorica(xyz(12, 0, 0), 0.5, 0.5)
coluna_dorica(xyz(15, 0, 0), 0.4, 0.7)
```

Figura 2.38: Variações de colunas dóricas segundo as proporções de Vitrúvio.

As proporções de Vitrúvio permitiram-nos reduzir o número de parâmetros independentes de uma coluna Dórica a apenas dois: o raio da base do fuste e o raio da base do coxim. Contudo, não parece correcto que estes parâmetros sejam totalmente independentes pois isso permite construir colunas aberrantes, em que o topo do fuste é mais largo do que a base, tal como acontece com a coluna mais à direita na Figura 2.38.

Na verdade, a caracterização da Ordem Dórica que apresentámos encontra-se incompleta pois, acerca das proporções das colunas, Vitrúvio afirmou ainda que:

> A diminuição no topo de uma coluna parece ser regulada segundo os seguintes princípios: se uma coluna tem menos de quinze pés, divida-se a largura na base em seis partes e usem-se cinco dessas partes para formar a largura no topo. Se a coluna tem entre quinze e vinte pés, divida-se a largura na base em seis partes e meio e usem-se cinco e meio dessas partes para a largura no topo da coluna. Se a coluna tem entre vinte e trinta pés, divida-se a largura na base em sete partes e faça-se o topo diminuido medir seis delas. Uma coluna de trinta a quarenta pés deve ser dividida na base em sete partes e meia e, no princípio da diminuição, deve ter seis partes e meia no topo. Colunas de quarenta a cinquenta pés devem ser divididas em oito partes e diminuidas para sete delas no topo da coluna debaixo do capitel. No caso de colunas mais altas, determine-se proporcionalmente a diminuição com base nos mesmos princípios. [Vitrúvio, Os Dez Livros da Arquitectura, Livro III, Cap. 3.1]

Estas considerações de Vitrúvio permitem-nos determinar a razão entre

o topo e a base de uma coluna em função da sua altura em pés.[10]

Consideremos então a definição de uma função, que iremos denominar de `raio_topo_fuste`, que recebe como parâmetros a largura da base da coluna e a altura da coluna e devolve como resultado a largura do topo da coluna.

Uma tradução literal das considerações de Vitrúvio para Julia permite-nos começar por escrever:

```
raio_topo_fuste(raio_base, altura) =
  if altura < 15
    5.0/6.0*raio_base
  else
    ...
  end
```

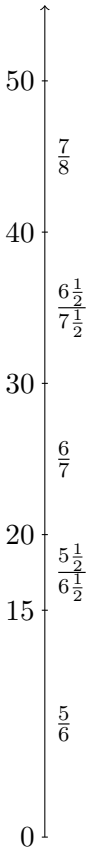
O fragmento anterior corresponde, obviamente, à afirmação: *se uma coluna tem menos de quinze pés, divida-se a largura na base em seis partes e usem-se cinco dessas partes para formar a largura no topo.* No caso de a coluna não ter menos de quinze pés, então passamos ao caso seguinte: *se a coluna tem entre quinze e vinte pés, divida-se a largura na base em seis partes e meio e usem-se cinco e meio dessas partes para a largura no topo da coluna.* A tradução deste segundo caso permite-nos escrever:

```
raio_topo_fuste(raio_base, altura) =
  if altura < 15
    5.0/6.0*raio_base
  elseif altura >= 15 && altura < 20
    5.5/6.5*raio_base
  else
    ...
  end
```

Uma análise cuidadosa das duas cláusulas anteriores mostra que, na realidade, estamos a fazer testes a mais na segunda cláusula. De facto, se conseguimos chegar à segunda cláusula é porque a primeira é falsa, i.e., a altura não é menor que 15 e, portanto, é maior ou igual a 15. Nesse caso, é inútil estar a testar novamente se a altura é maior ou igual a 15. Assim, podemos simplificar a função e escrever:

[10] O *pé* foi a unidade fundamental de medida durante inúmeros séculos, mas a sua real dimensão variou ao longo do tempo. O comprimento do pé internacional é de 304.8 milímetros e foi estabelecido, por acordo, em 1958. Antes disso, vários outros comprimentos foram usados, como o pé Dórico de 324 milímetros, os pés Jónico e Romano de 296 milímetros, o pé Ateniense de 315 milímetros, os pés Egípcio e Fenício de 300 milímetros, etc.

```
raio_topo_fuste(raio_base, altura) =
  if altura < 15
    5.0/6.0*raio_base
  elseif altura < 20
    5.5/6.5*raio_base
  else
    ...
  end
```



A continuação da tradução levar-nos-á, então, a:

```
raio_topo_fuste(raio_base, altura) =
  if altura < 15
    5.0/6.0*raio_base
  elseif altura < 20
    5.5/6.5*raio_base
  elseif altura < 30
    6.0/7.0*raio_base
  elseif altura < 40
    6.5/7.5*raio_base
  elseif altura < 50
    7.0/8.0*raio_base
  else
    ...
  end
```

O problema agora é que Vitrúvio deixou a porta aberta para colunas arbitrariamente altas, dizendo simplesmente que, *no caso de colunas mais altas, determine-se proporcionalmente a diminuição com base nos mesmos princípios*. Para percebermos claramente de que princípios estamos a falar, consideremos a evolução da relação entre o topo e a base das colunas que é visível na imagem lateral.

A razão entre o raio do topo da coluna e o raio da base da coluna é, tal como se pode ver na margem (e já era possível deduzir da função `raio_topo_fuste`), uma sucessão da forma

$$\frac{5}{6}, \frac{5\frac{1}{2}}{6\frac{1}{2}}, \frac{6}{7}, \frac{6\frac{1}{2}}{7\frac{1}{2}}, \frac{7}{8}, \cdots$$

Torna-se agora óbvio que, para colunas mais altas, "os mesmos princípios" de que Vitrúvio fala se resumem a, para cada 10 pés adicionais, somar $\frac{1}{2}$ quer ao numerador, quer ao denominador. No entanto, é importante reparar que este princípio só pode ser aplicado a partir dos 15 pés de altura, pois o primeiro intervalo é maior que os restantes. Assim, temos de tratar de forma diferente as colunas até aos 15 pés e, daí para a frente, basta subtrair 20 pés à altura e determinar a divisão inteira por 10 para saber o número de vezes que precisamos de somar $\frac{1}{2}$ quer ao numerador quer ao denominador de $\frac{6}{7}$.

É este "tratar de forma diferente" um caso e outro que, mais uma vez, sugere a necessidade de um mecanismo de selecção: é necessário distinguir dois casos e reagir em conformidade para cada um. No caso da coluna de Vitrúvio, se a coluna tem uma altura a até 15 pés, a razão entre o topo e a base é $r = \frac{5}{6}$; se a altura a não é inferior a 15 pés, a razão r entre o topo e a base será:

$$r = \frac{6 + \lfloor \frac{a-20}{10} \rfloor \cdot \frac{1}{2}}{7 + \lfloor \frac{a-20}{10} \rfloor \cdot \frac{1}{2}}$$

A título de exemplo, consideremos uma coluna com 43 pés de altura. A divisão inteira de $43 - 20$ por 10 é 2 portanto temos de somar $2 \cdot \frac{1}{2} = 1$ ao numerador e denominador de $\frac{6}{7}$, obtendo $\frac{7}{8} = 0.875$.

Para um segundo exemplo, consideremos a proposta de Adolf Loos para a sede do jornal americano Chicago Tribune, um edifício de 122 metros com a forma de coluna dórica assente numa base. A coluna propriamente dita teria cerca de 85 metros de altura. Tendo em conta que um pé, na ordem Dórica, media 324 milímetros, em 85 metros existem $85/0.324 \approx 262$ pés. A divisão inteira de $262 - 20$ por 10 é 24. A razão entre o topo e a base desta coluna será então de $\frac{6+24/2}{7+24/2} = \frac{18}{19} = 0.95$. Este valor, por ser próximo da unidade, mostra que a coluna seria praticamente cilíndrica.

Com base nestas considerações, podemos agora definir uma função que, dado um número inteiro representando a altura da coluna em pés, calcula a razão entre o topo e a base da coluna. Antes, contudo, convém simplificar a fórmula para as colunas com altura não inferior a 15 pés. Assim,

$$r = \frac{6 + \lfloor \frac{a-20}{10} \rfloor \cdot \frac{1}{2}}{7 + \lfloor \frac{a-20}{10} \rfloor \cdot \frac{1}{2}} = \frac{12 + \lfloor \frac{a-20}{10} \rfloor}{14 + \lfloor \frac{a-20}{10} \rfloor} = \frac{12 + \lfloor \frac{a}{10} \rfloor - 2}{14 + \lfloor \frac{a}{10} \rfloor - 2} = \frac{10 + \lfloor \frac{a}{10} \rfloor}{12 + \lfloor \frac{a}{10} \rfloor}$$

A definição da função fica então:

```
raio_topo_fuste(raio_base, altura) =
  if altura < 15
    5/6*raio_base
  else
    let divisoes = floor(altura/10)
      (10 + divisoes)/(12 + divisoes)*raio_base
    end
  end
```

Esta é a última relação que nos falta para especificarmos completamente o desenho de uma coluna dórica de acordo com as proporções referidas por Vitrúvio no seu tratado de arquitectura. Vamos considerar, para este desenho, que vamos fornecer as coordenadas do centro da base da coluna e a sua altura. Todos os restantes parâmetros serão calculados em termos destes. Eis a definição da função:

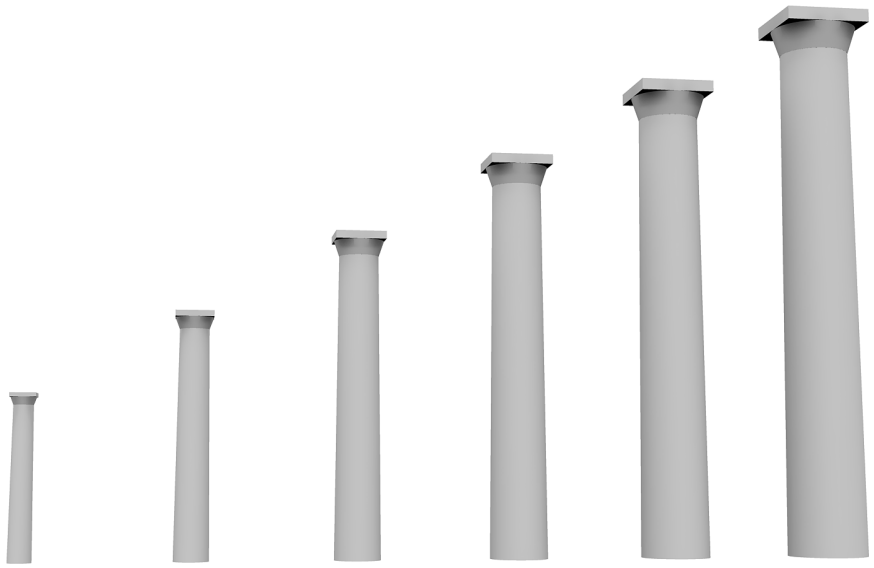


Figura 2.39: Variações de colunas dóricas segundo as proporções de Vitrúvio.

```
coluna_dorica(p, altura) =
  let r_base_fuste = altura/14,
      r_base_coxim = raio_topo_fuste(r_base_fuste, altura),
      a_fuste = 13*r_base_fuste,
      a_coxim = 2/3*r_base_fuste,
      a_abaco = 1/3*r_base_fuste,
      l_abaco = 13/6*r_base_fuste
    fuste(p, a_fuste, r_base_fuste, r_base_coxim)
    coxim(p + vz(a_fuste), a_coxim, r_base_coxim, l_abaco/2)
    abaco(p + vz(a_fuste + a_coxim), a_abaco, l_abaco)
  end
```

O seguinte exemplo de utilização da função produz a sequência de colunas apresentadas na Figura 2.39:[11]

```
coluna_dorica(xy(0, 0), 10)
coluna_dorica(xy(10, 0), 15)
coluna_dorica(xy(20, 0), 20)
coluna_dorica(xy(30, 0), 25)
coluna_dorica(xy(40, 0), 30)
coluna_dorica(xy(50, 0), 35)
```

Finalmente, vale a pena referir que as funções `coluna` e `coluna_dorica` representam dois extremos: a primeira modela colunas com um grande número de graus de liberdade, desde a localização às medidas do fuste,

[11] Note-se que, agora, a altura da coluna tem de ser especificada em pés dóricos.

coxim, e ábaco, enquanto a segunda apenas admite variações na localização e altura da coluna. A função `coluna_dorica` é, na realidade, um caso particular da função `coluna` e, por isso, pode ser definida à custa dela:

```
coluna_dorica(p, altura) =
  let r_base_fuste = altura/14,
      r_base_coxim = raio_topo_fuste(r_base_fuste, altura),
      a_fuste = 13*r_base_fuste,
      a_coxim = 2/3*r_base_fuste,
      a_abaco = 1/3*r_base_fuste,
      l_abaco = 13/6*r_base_fuste
    coluna(p, a_fuste, r_base_fuste, a_coxim, r_base_coxim, a_abaco, l_abaco)
  end
```

As funções `coluna` e `coluna_dorica` são também um bom exemplo de uma estratégia de modelação. Sempre que possível, devemos começar por modelar de forma genérica, contemplando o maior número de graus de liberdade que for razoável, e só então consideramos os casos particulares dessa modelação, que definiremos com funções próprias que, naturalmente, poderão recorrer à definição do caso geral.

Capítulo 3

Recursão

3.1 Introdução

Vimos que as nossas funções, para fazerem algo útil, precisam de invocar outras funções. Por exemplo, se já tivermos a função que calcula o quadrado de um número e pretendermos definir a função que calcula o cubo de um número, podemos facilmente fazê-lo à custa do quadrado e de uma multiplicação adicional, i.e.:

```
cubo(x) = quadrado(x) * x
```

Do mesmo modo, podemos definir a função `quarta_potencia` à custa do cubo e de uma multiplicação adicional, i.e.:

```
quarta_potencia(x) = cubo(x) * x
```

Como é óbvio, podemos continuar a definir sucessivamente novas funções para calcular potências crescentes, mas isso não só é moroso como será sempre limitado. Seria muito mais útil podermos generalizar o processo e definir simplesmente a função potência que, a partir de dois números (a *base* e o *expoente*), calcula o primeiro elevado ao segundo.

No entanto, aquilo que fizemos para a `quarta_potencia`, o `cubo` e o `quadrado` dão-nos uma pista muito importante: *se tivermos uma função que calcula a potência de expoente imediatamente inferior, então basta-nos uma multiplicação adicional para calcular a potência seguinte*.

Dito de outra forma, temos:

```
potencia(x, n) = potencia_inferior(x, n) * x
```

Embora tenhamos conseguido simplificar o problema do cálculo de potência, sobrou uma questão por responder: como podemos calcular a potência imediatamente inferior? A resposta poderá não ser óbvia mas, uma vez percebida, é trivial: *a potência imediatamente inferior à potência de expoente*

n é a potência de expoente $(n-1)$. Isto implica que `potencia_inferior(x, n)` é exactamente o mesmo que `potencia(x, n - 1)`. Com base nesta ideia, podemos reescrever a definição anterior:

```
potencia(x, n) = potencia(x, n - 1) * x
```

Apesar da nossa solução engenhosa, esta definição tem um problema: qualquer que seja a potência que tentemos calcular, nunca conseguiremos obter o resultado final. Para percebermos este problema, é mais simples usar um caso real: tentemos calcular a terceira potência do número 4, i.e., `potencia(4, 3)`.

Para isso, de acordo com a definição da função `potencia`, será preciso avaliar a expressão
`potencia(4, 2)*4`
que, por sua vez, implica avaliar
`potencia(4, 1)*4*4`
que, por sua vez, implica avaliar
`potencia(4, 0)*4*4*4`
que, por sua vez, implica avaliar
`potencia(4, -1)*4*4*4*4`
que, por sua vez, implica avaliar
`potencia(4, -2)*4*4*4*4*4`
que, por sua vez, implica avaliar
`potencia(4, -3)*4*4*4*4*4*4`
que, por sua vez, implica avaliar ...

É fácil vermos que este processo nunca termina. O problema está no facto de termos reduzido o cálculo da potência de um número elevado a um expoente ao cálculo da potência desse número elevado ao expoente imediatamente inferior, mas não dissemos em que situação é que já temos um expoente suficientemente simples cuja solução seja imediata. Quais são as situações em que isso acontece? Já vimos que quando o expoente é 2, a função `quadrado` devolve o resultado correcto, pelo que o caso $n = 2$ é já suficientemente simples. No entanto, é possível ter um caso ainda mais simples: quando o expoente é 1, o resultado é simplesmente a base. Finalmente, o caso mais simples de todos: quando o expoente é zero, o resultado é 1, independentemente da base. Este último caso é fácil de perceber quando vemos que a avaliação de `potencia(4, 2)` (i.e., do quadrado de quatro) se reduz, em última análise, a `potencia(4, 0)*4*4`. Para que esta expressão seja equivalente a `4*4` é necessário que a avaliação de `potencia(4, 0)` produza `1`.

Estamos então em condições de definir correctamente a função `potencia`:

1. Quando o expoente é zero, o resultado é um.
2. Caso contrário, calculamos a potência de expoente imediatamente inferior e multiplicamo-la pela base.

```
potencia(x, n) =
  if n == 0
    1
  else
    potencia(x, n - 1) * x
  end
```

A função anterior é um exemplo de uma função *recursiva*, i.e., uma função que está definida em termos de si própria. Dito de outra forma, uma função recursiva é uma função que se usa a si própria na sua definição. Esse uso é óbvio quando "desenrolamos" a avaliação de `potencia(4, 3)`:

```
potencia(4, 3)
      ↓
potencia(4, 2)*4
      ↓
potencia(4, 1)*4*4
      ↓
potencia(4, 0)*4*4*4
      ↓
1*4*4*4
   ↓
4*4*4
  ↓
16*4
  ↓
64
```

A *recursão* é o mecanismo que permite que uma função se possa invocar a si própria durante a sua própria avaliação. A recursão é uma das mais importantes ferramentas de programação, pelo que é fundamental que a percebamos bem. Muitos problemas aparentemente complexos possuem soluções recursivas surpreendentemente simples.

Existem inúmeros exemplos de funções recursivas. Uma das mais simples é a função factorial que se define matematicamente como:

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n \cdot (n-1)!, & \text{caso contrário.} \end{cases}$$

A tradução desta fórmula para Julia é directa:

```
factorial(n) =
  if n == 0
    1
  else
    n * factorial(n - 1)
  end
```

É importante repararmos que em todas as funções recursivas existe:

- Um caso básico (também chamado *caso de paragem*) cujo resultado é imediatamente conhecido.
- Um caso não básico (também chamado *caso recursivo*) em que se transforma o problema original num sub-problema mais simples.

Se analisarmos a função factorial, o caso básico é o teste de igualdade a zero `n == 0`, o resultado imediato é `1`, e o caso recursivo é, obviamente, `n*factorial(n - 1)`.

Geralmente, uma função recursiva só está correcta se tiver uma expressão condicional que identifique o caso básico, mas não é obrigatório que assim seja. A invocação de uma função recursiva consiste em ir resolvendo subproblemas sucessivamente mais simples até se atingir o caso mais simples de todos, cujo resultado é imediato. Desta forma, o padrão mais comum para escrever uma função recursiva é:

- Começar por testar os casos básicos.
- Fazer uma invocação recursiva com um subproblema cada vez mais próximo de um caso básico.
- Usar o resultado da invocação recursiva para produzir o resultado da invocação original.

Dado este padrão, os erros mais comuns associados às funções recursivas são, naturalmente:

- Não detectar um caso básico.
- A recursão não diminuir a complexidade do problema, i.e., não passar para um problema mais simples.
- Não usar correctamente o resultado da recursão para produzir o resultado originalmente pretendido.

Repare-se que uma função recursiva que funciona perfeitamente para os casos para que foi prevista pode estar completamente errada para outros casos. A função `factorial` é um exemplo: quando o argumento é negativo, o problema torna-se cada vez mais complexo, cada vez mais longe do caso simples:

```
factorial(-1)
      ↓
-1*factorial(-2)
      ↓
-1*-2*factorial(-3)
```

```
                    ↓
  -1*-2*-3*factorial(-4)
                    ↓
 -1*-2*-3*-4*factorial(-5))
                    ↓
-1*-2*-3*-4*-5*factorial(-6)
                    ↓
                   ...
```

O caso mais frequente de erro numa função recursiva é a recursão nunca parar, ou porque não se detecta correctamente o caso básico, ou por a recursão não diminuir a complexidade do problema. Neste caso, o número de invocações recursivas cresce indefinidamente até esgotar a memória do computador, altura em que o programa gera um erro. Eis um exemplo:

```
julia> factorial(3)
6
julia> factorial(-1)
RuntimeError: maximum recursion depth exceeded
```

É muito importante compreendermos bem o conceito de recursão. Embora a princípio possa ser difícil abarcar por completo as implicações deste conceito, a recursão permite resolver, com enorme simplicidade, problemas aparentemente muito complexos.

Exercício 3.1.1 A função de Ackermann é definida para números não negativos da seguinte forma:

$$A(m,n) = \begin{cases} n+1 & \text{se } m = 0 \\ A(m-1,1) & \text{se } m > 0 \text{ e } n = 0 \\ A(m-1, A(m,n-1)) & \text{se } m > 0 \text{ e } n > 0 \end{cases}$$

Defina, em Julia, a função de Ackermann.

Exercício 3.1.2 Indique o valor de

1. `ackermann(0, 8)`
2. `ackermann(1, 8)`
3. `ackermann(2, 8)`
4. `ackermann(3, 8)`
5. `ackermann(4, 8)`

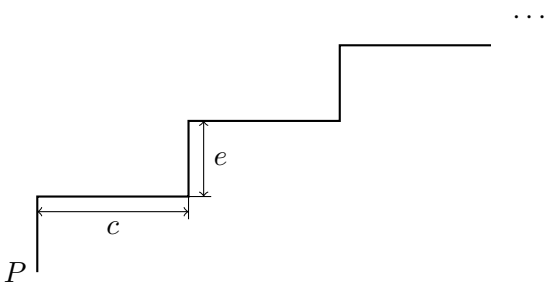


Figura 3.1: Perfil de uma escada com n degraus cujo primeiro degrau começa no ponto P e cujos degraus possuem um cobertor c e um espelho e.

3.2 Recursão em Arquitectura

Como iremos ver, também em arquitectura a recursão é um conceito fundamental. A título de exemplo, consideremos o perfil de uma escada, tal como esquematizado na Figura 3.1 e suponhamos que pretendemos definir uma função denominada `escada` que, dado o ponto P, dados o comprimento do cobertor c e o comprimento do espelho e de cada degrau e, finalmente, dado o número n de degraus, desenha a escada com o primeiro espelho a começar a partir do ponto P. Dados estes parâmetros, a definição da função deverá começar por:

```
escada(p, c, e, n) =
  ...
```

Para implementarmos esta função temos de ser capazes de decompor o problema em subproblemas menos complexos e é aqui que a recursão dá uma enorme ajuda: ela permite-nos decompor o desenho de uma escada com n degraus no desenho de um degrau seguido do desenho de uma escada com $n - 1$ degraus, tal como se apresenta no esquema da Figura 3.2.

Isto quer dizer que a função ficará com a forma:

```
escada(p, c, e, n) =
  ...
  degrau(p, c, e)
  escada(p + vxy(c, e), c, e, n - 1)
```

Para desenharmos um degrau, podemos definir a seguinte função que cria os segmentos do espelho e do cobertor:

```
degrau(p, c, e) =
  line(p, p + vy(e), p + vxy(c, e))
```

O problema que se coloca agora é que a função `escada` precisa de parar de desenhar degraus num determinado momento. É fácil vermos que

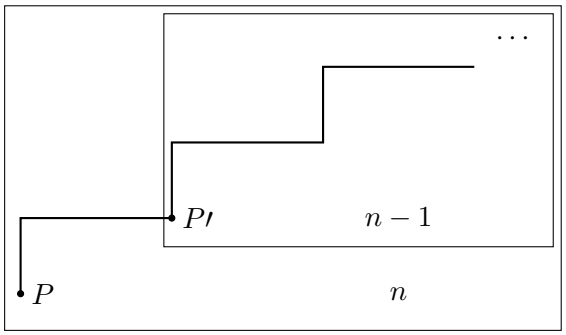


Figura 3.2: Decomposição do desenho de uma escada de n degraus no desenho de um degrau seguido do desenho de uma escada de $n - 1$ degraus.

esse momento chega quando, ao reduzirmos sucessivamente o números de degraus, atingimos um ponto em que esse número é zero. Assim, quando pedimos para desenhar uma escada com zero degraus, a função `escada` já não precisa de fazer nada. Ora o Julia disponibiliza uma expressão precisamente para indicarmos que não é preciso fazer nada: `nothing`. Isso quer dizer que a função tem de ter a seguinte forma:

```
escada(p, c, e, n) =
  if n == 0
    nothing
  else
    degrau(p, c, e)
    escada(p + vxy(c, e), c, e, n - 1)
  end
```

Para vermos um exemplo mais interessante de recursão em Arquitectura, consideremos a pirâmide de Saqqara ilustrada na Figura 3.3, construída pelo arquitecto Imhotep durante o século XXVII a.C. Esta pirâmide de degraus é considerada a primeira pirâmide do Egito e a mais antiga construção monumental em pedra do mundo, sendo composta por seis mastabas progressivamente mais pequenas, empilhadas umas sobre as outras. Curiosamente, a construção destas mastabas foi, ela própria, feita recursivamente. Reza a história que o faraó Djoser encomendou a Imhotep um túmulo monumental para guardar os seus restos mortais, tendo Imhotep construído uma mastaba, que não agradou ao faraó por ser insuficientemente monumental. Em resposta, Imhotep alargou a base e construiu outra mastaba por cima. Como o faraó continuava insatisfeito, Imhotep alargou as duas primeiras mastabas e construiu uma terceira em cima, depois alargou essas três e construiu uma quarta, repetindo este processo até o faraó ficar satisfeito, o que só aconteceu à sexta mastaba.

Se Imhotep soubesse desde o início qual era a forma final da pirâmide, poderia empregar uma abordagem, também recursiva, mas que consistia

Figura 3.3: A pirâmide de degraus de Saqqara. Fotografia de Charles J. Sharp.

em pensar a pirâmide de degraus como uma mastaba em cima da qual assenta outra pirâmide de degraus mais pequena. É esta abordagem que iremos agora empregar.

Formalmente, podemos definir uma pirâmide de n degraus como uma mastaba em cima da qual assenta uma pirâmide de $n-1$ degraus. Para completar a definição temos de referir que quando criamos a última mastaba, a pirâmide de 0 degraus que lhe está em cima é, na realidade, inexistente.

Assim, considerando a ilustração da Figura 3.4, se admitirmos que o centro da base da pirâmide está na posição p e que as várias mastabas são troncos de pirâmides, podemos escrever:

```
piramide_degraus(p, b, t, h, d, n) =
  if n == 0
    nothing
  else
    regular_pyramid_frustum(4, p, b, 0, h, t)
    piramide_degraus(p + vz(h), b - d, t - d, h, d, n - 1)
  end
```

Um exemplo aproximado da pirâmide de Saqqara será então:

```
piramide_degraus(xyz(0, 0, 0), 120, 115, 20, 15, 6)
```

Exercício 3.2.1 A definição anterior não reflecte rigorosamente a geometria da pirâmide de degraus de Saqqara pois esta possui umas rampas de ligação entre as mastabas, tal como é visivel no seguinte esquema onde

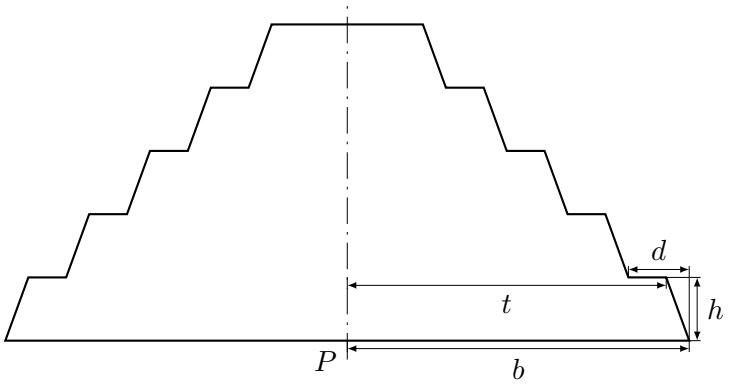
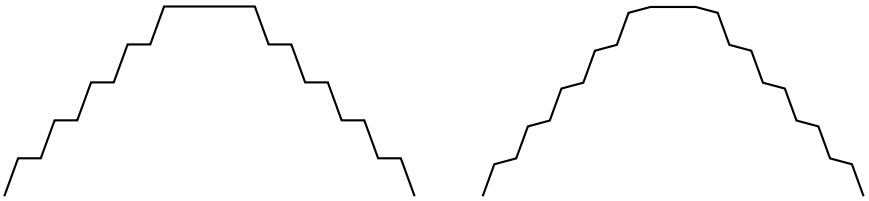
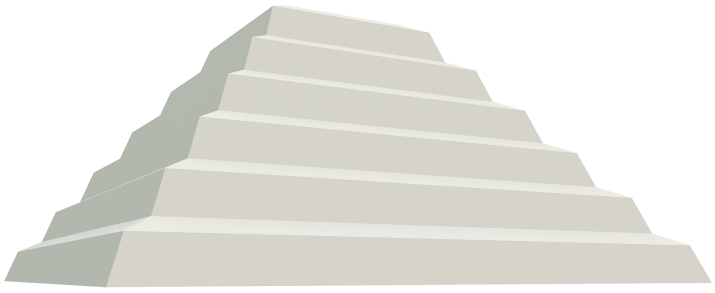


Figura 3.4: Esquema de uma pirâmide de degraus.

comparamos as secções da pirâmide que definimos (à esquerda) e a da verdadeira pirâmide de Saqqara (à direita):

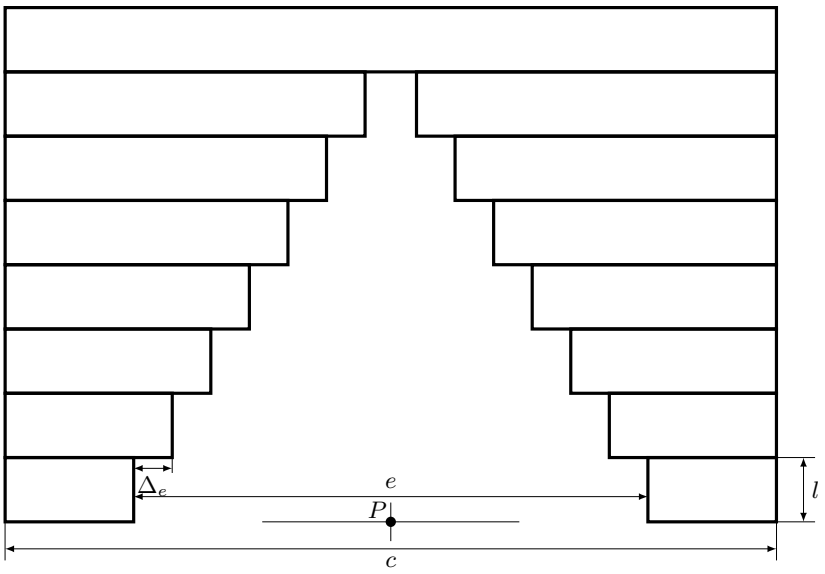


Defina uma versão mais rigorosa da função `piramide_degraus`, que receba, para além dos parâmetros anteriores, a altura das rampas de ligação. Experimente valores para os parâmetros que lhe permitem gerar um modelo semelhante ao seguinte:



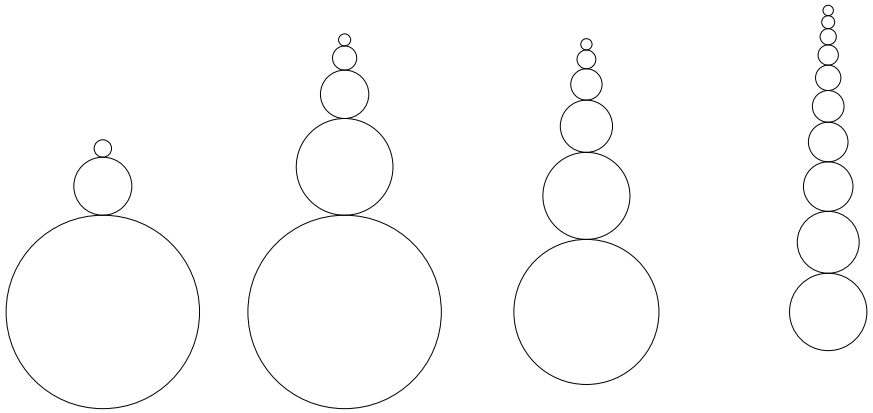
Exercício 3.2.2 O arco *falso* é a mais antiga forma de arco, sendo formado por paralelipípedos dispostos horizontalmente em degraus formando uma

abertura que se vai estreitando até ao topo, terminando com uma viga horizontal, tal como se apresenta no seguinte esquema:



Partindo do pressuposto que os paralelipípedos possuem secção quadrada de lado l, que a redução de abertura Δ_e é igual para todos os degraus e, finalmente, que a posição P está no centro da base do arco, defina a função `arco_falso` que, a partir dos parâmetros P, c, e, Δ_e, e l, constrói um arco falso.

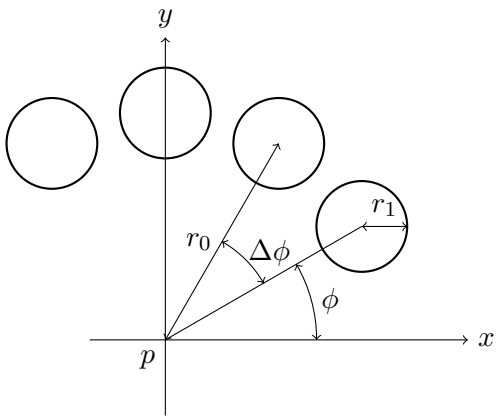
Exercício 3.2.3 Defina uma função `equilibrio_circulos` capaz de criar qualquer uma das figuras apresentadas em seguida:



Note que os círculos possuem raios que estão em progressão geométrica de razão f, com $0 < f < 1$. Assim, cada círculo (excepto o primeiro) tem um raio que é o produto de f pelo raio do círculo maior em que está apoiado. O círculo mais pequeno de todos tem raio maior ou igual a 1. A

sua função deverá ter como parâmetros o centro e o raio do círculo maior e, ainda, o factor de redução f.

Exercício 3.2.4 Considere o desenho de círculos tal como apresentado na seguinte imagem:

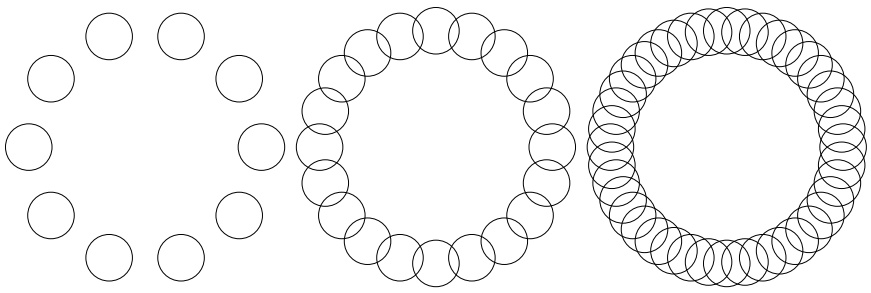


Escreva uma função denominada `circulos_radiais` que, a partir das coordenadas p do centro de rotação, do número de círculos n, do raio de translação r_0, do raio de circunferência r_1, do ângulo inicial ϕ e do incremento de ângulo $\Delta\phi$, desenha os círculos tal como apresentados na figura anterior.

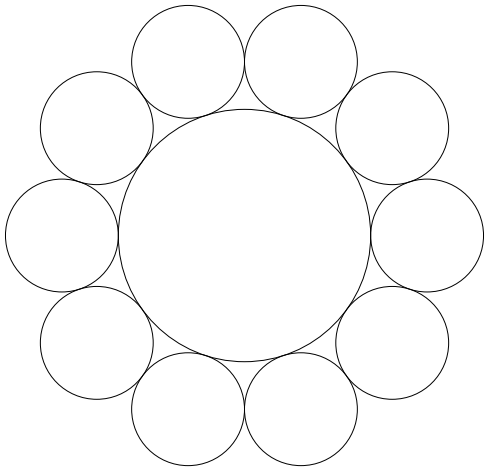
Teste a sua função com os seguintes expressões:

```
circulos_radiais(xy(0, 0), 10, 10, 2, 0, pi/5)
circulos_radiais(xy(25, 0), 20, 10, 2, 0, pi/10)
circulos_radiais(xy(50, 0), 40, 10, 2, 0, pi/20)
```

cuja avaliação deverá gerar a imagem seguinte:



Exercício 3.2.5 Considere o desenho de flores simbólicas compostas por um círculo interior em torno da qual estão dispostos círculos radiais correspondentes a pétalas. Estes círculos deverão ser tangentes uns aos outros e ao círculo interior, tal como se apresenta na seguinte imagem:

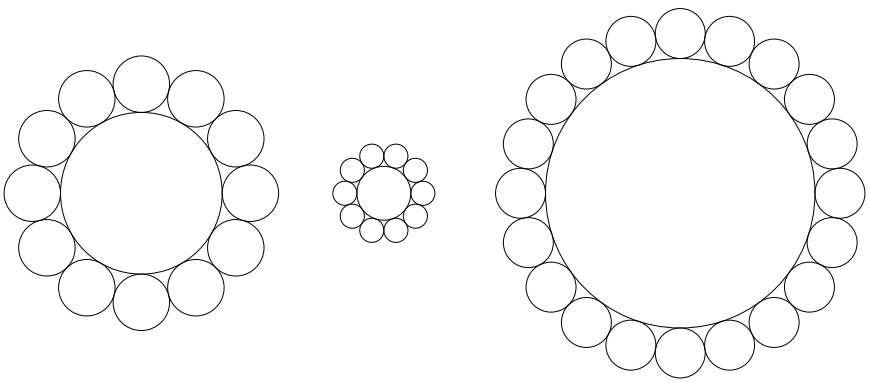


Defina a função `flor` que recebe apenas o ponto correspondente ao centro da flor, o raio do círculo interior e o número de pétalas.

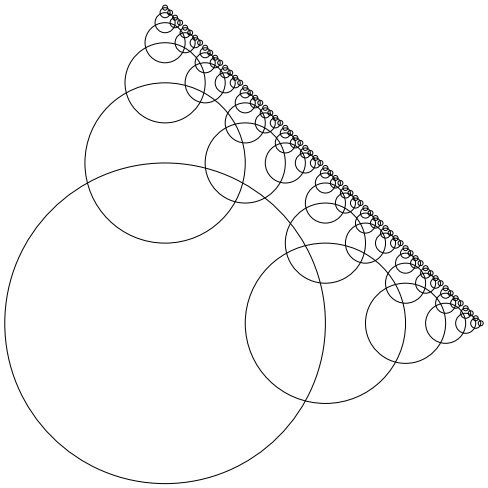
Teste a sua função com as expressões

```
flor(xy(0, 0), 5, 12)
flor(xy(18, 0), 2, 10)
flor(xy(40, 0), 10, 20)
```

que deverão gerar a imagem seguinte:

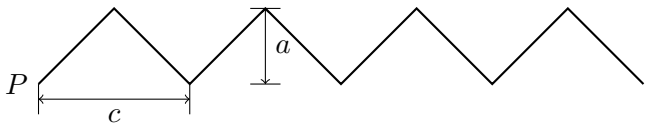


Exercício 3.2.6 Defina uma função `circulos` capaz de criar a figura apresentada em seguida:

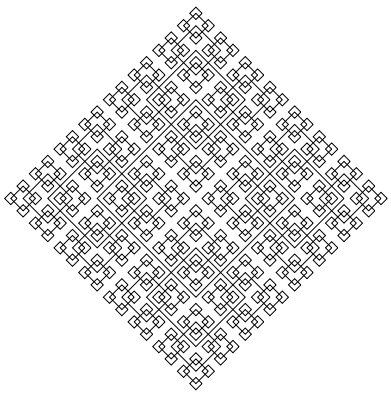


Note que os círculos possuem raios que estão em progressão geométrica de razão $\frac{1}{2}$. Dito de outra forma, os círculos mais pequenos têm metade do raio do círculo maior que lhes é adjacente. Os círculos mais pequenos de todos têm raio maior ou igual a 1. A sua função deverá ter como parâmetros apenas o centro e o raio do círculo maior.

Exercício 3.2.7 Defina uma função denominada `serra` que, dado um ponto P, um número de dentes, o comprimento c de cada dente e a altura a de cada dente, desenha uma serra com o primeiro dente a começar a partir do ponto P, tal como se vê na imagem seguinte:

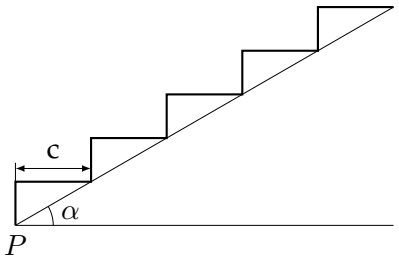


Exercício 3.2.8 Defina uma função `losangos` capaz de criar a figura apresentada em seguida:



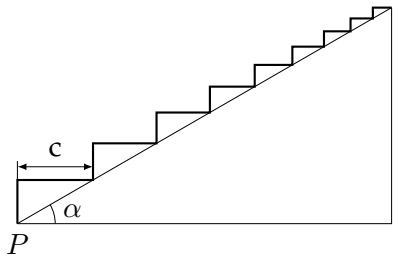
Note que os losangos possuem dimensões que estão em progressão geométrica de razão $\frac{1}{2}$. Dito de outra forma, os losangos mais pequenos têm metade do tamanho do losango maior em cujas extremidades estão centrados. Os losangos mais pequenos de todos têm largura maior ou igual a 1. A sua função deverá ter como parâmetros apenas o centro e a largura do losango maior.

Exercício 3.2.9 Considere a escada esquematizada na seguinte figura e destinada a vencer uma rampa de inclinação α.



Defina a função `escada_rampa` que recebe o ponto P, o ângulo α, o cobertor c e o número de degraus n e constrói a escada descrita no esquema anterior.

Exercício 3.2.10 Considere a escada esquematizada na seguinte figura e destinada a vencer uma rampa de inclinação α.



Note que os degraus da escada possuem dimensões que estão em progressão geométrica de razão f, i.e., dado um degrau cujo cobertor é de dimensão c, o degrau imediatamente acima tem um cobertor de dimensão $f \cdot c$. Defina a função `escada_progressao_geometrica` que recebe o ponto P, o ângulo α, o cobertor c, o número de degraus n e o factor f e constrói a escada descrita no esquema anterior.

3.3 Templos Dóricos

Vimos, pelas descrições de Vitrúvio, que os Gregos criaram um elaborado sistema de proporções para colunas. Estas colunas eram usadas para a criação de *pórticos*, em que uma sucessão de colunas encimadas por um

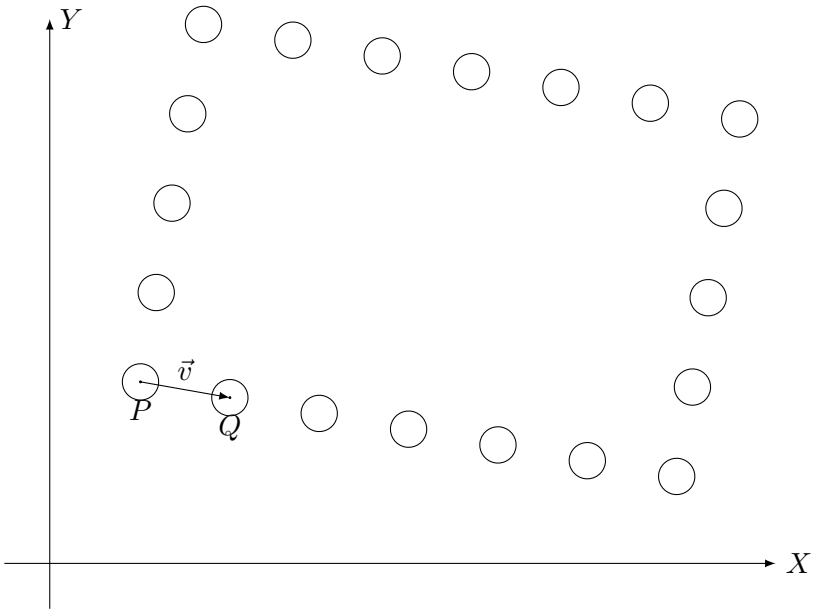


Figura 3.5: Planta de um templo com uma orientação arbitrária.

telhado servia de entrada para os edifícios e, em particular, para os templos. Quando esse arranjo de colunas era avançado em relação ao edifício, denominava-se o mesmo de próstilo, classificando-se este pelo número de colunas que possuem em Distilo, Tristilo, Tetrastilo, Pentastilo, Hexastilo, etc. Quando o próstilo se alargava a todo o edifício, colocando colunas a toda a sua volta, denominava-se de peristilo.

Para além de descrever as proporções das colunas, Vitrúvio também explicou no seu famoso tratado as regras a que devia obedecer a construção dos templos, em particular, no que diz respeito à sua orientação, que devia ser de este para oeste, e no que diz respeito à separação entre colunas, distinguindo vários casos de templos, desde aqueles em que o espaço entre colunas era muito reduzido (*picnostilo*) até aos templos com espaçamento excessivamente alargado (*araeostilo*), passando pelo seu favorito (*eustilo*) em que o espaço entre colunas é variável, sendo maior nas colunas centrais.

Para simplificar a nossa implementação, vamos ignorar estes detalhes e, ao invés de distinguir vários *stilo*, vamos simplesmente considerar o posicionamento de colunas distribuídas linearmente segundo uma determinada orientação, tal como esquematizamos na Figura 3.5.

Para procedermos ao posicionamento das colunas no templo, ilustrado na Figura 3.5, dado o vector $\vec{v}$ de orientação e separação de colunas, a partir da posição de qualquer coluna P, determinamos a posição da coluna seguinte através de $P + \vec{v}$. Este raciocínio permite-nos definir uma primeira função capaz de criar uma fileira de colunas. Esta função irá usar,

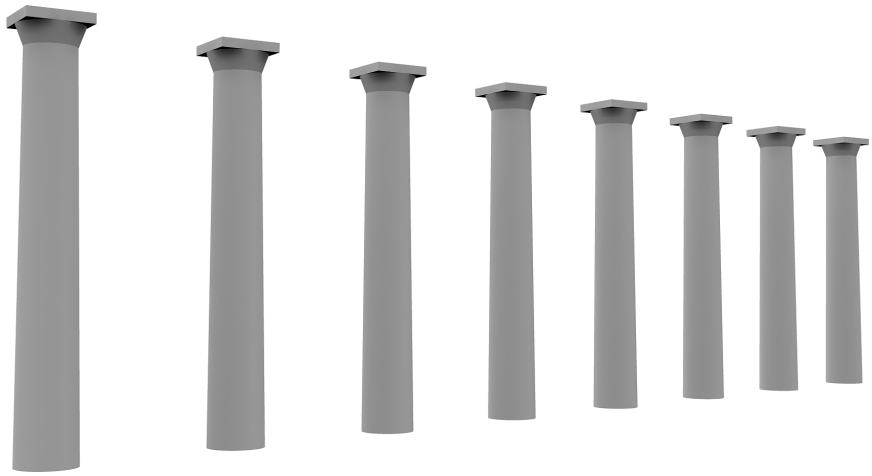


Figura 3.6: Uma perspectiva de um conjunto de oito colunas dóricas com 10 unidades de altura e 5 unidades de separação entre os eixos da colunas ao longo do eixo X.

como parâmetros, as coordenadas P da base do eixo da primeira coluna, a altura h da coluna, o vector $\vec{v}$ de separação entre os eixos das colunas e, finalmente, o número n de colunas que pretendemos colocar. O raciocínio para a definição desta função é, mais uma vez, recursivo:

- Se o número de colunas a colocar for zero, então não é preciso fazer nada.
- Caso contrário, colocamos uma coluna no ponto P e, recursivamente, colocamos as restantes colunas a partir do ponto que resulta de somarmos o vector de separação v ao ponto P.

Traduzindo este raciocínio para Julia, temos:

```
colunas_doricas(p, h, v, n) =
  if n == 0
    nothing
  else
    coluna_dorica(p, h)
    colunas_doricas(p + v, h, v, n - 1)
  end
```

Podemos testar a criação das colunas usando, por exemplo:

```
colunas_doricas(xy(0, 0), 10, vxy(5, 0), 8)
```

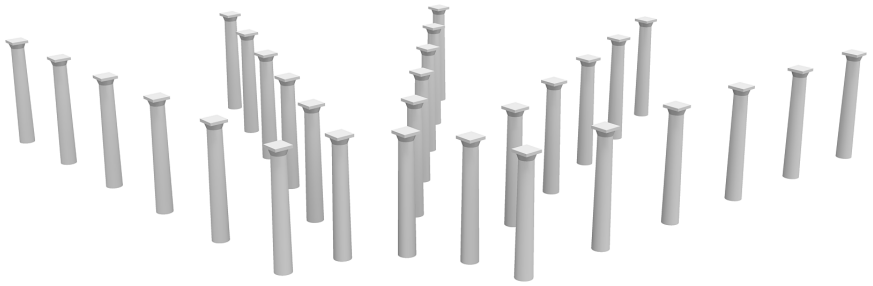
cujo resultado está apresentado na Figura 3.6:

Exercício 3.3.1 Embora a utilização do vector de separação entre colunas seja relativamente simples, é possível simplificar ainda mais através

do cálculo desse vector a partir dos pontos inicial e final da fileira de colunas. Usando a função `colunas_doricas`, defina uma função denominada `colunas_doricas_entre` que, dados os centros P e Q da base das colunas inicial e final, a altura h das colunas e, finalmente, o número de colunas, cria a fileira de colunas entre aqueles dois centros.

A título de exemplo, a imagem seguinte mostra o resultado da avaliação das seguintes expressões:

```
colunas_doricas_entre(pol(10, 0.0), pol(50, 0.0), 8, 6)
colunas_doricas_entre(pol(10, 0.4), pol(50, 0.4), 8, 6)
colunas_doricas_entre(pol(10, 0.8), pol(50, 0.8), 8, 6)
colunas_doricas_entre(pol(10, 1.2), pol(50, 1.2), 8, 6)
colunas_doricas_entre(pol(10, 1.6), pol(50, 1.6), 8, 6)
```



A partir do momento em que sabemos construir fileiras de colunas, torna-se relativamente fácil a construção das quatro fileiras necessárias para os templos em peristilo. Normalmente, a descrição destes templos faz-se em termos do número de colunas da fronte e do número de colunas do lado, mas assumindo que as colunas dos cantos contam para ambas as medidas. Isto quer dizer que num templo de, por exemplo, 6×12 colunas existem, na realidade, apenas $4 \times 2 + 10 \times 2 + 4 = 32$ colunas. Para a construção do peristilo, para além do número de colunas das frontes e lados, será necessário conhecer a posição das colunas extremas do templo e, claro, a altura das colunas.

Em termos de algoritmo, vamos começar por construir um dos cantos do peristilo, i.e., uma fronte e um lado:

```
canto_peristilo_dorico(p, altura, v0, n0, v1, n1) =
  begin
    colunas_doricas(p, altura, v0, n0)
    colunas_doricas(p + v1, altura, v1, n1 - 1)
  end
```

Note-se que, para evitar repetir colunas, a segunda fileira começa na segunda coluna e, logicamente, coloca menos uma coluna.

Para construirmos o peristilo completo, basta-nos construir um canto e, de seguida, construir o outro canto mas com menos uma coluna em cada lado e progredindo nas direcções opostas.

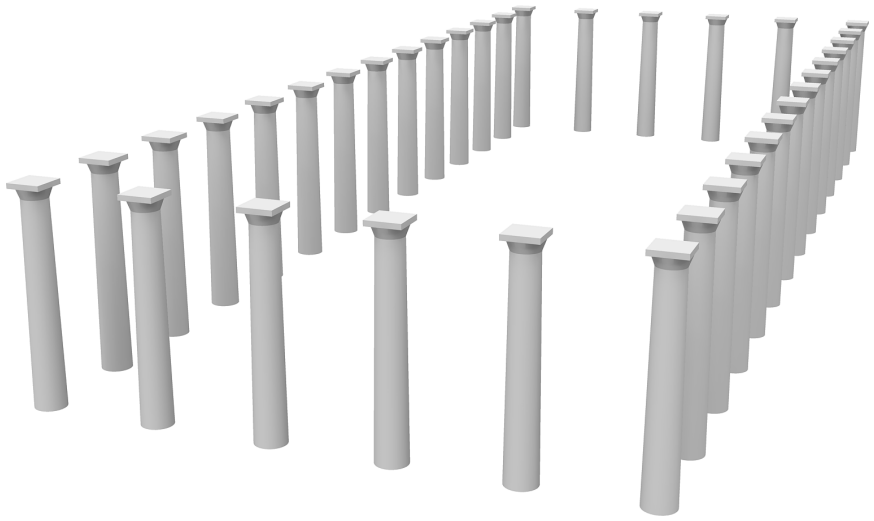


Figura 3.7: Uma perspectiva do peristilo do templo de Segesta. As colunas foram geradas pela função `peristilo_dorico` usando, como parâmetros, 6 colunas na fronte e 14 no lado, com distância intercolunar de 4.8 metros na fronte e 4.6 metros no lado, e colunas de 9 metros de altura.

```
peristilo_dorico(p, altura, v0, n0, v1, n1) =
  begin
    canto_peristilo_dorico(p, altura, v0, n0, v1, n1)
    canto_peristilo_dorico(p + v0*(n0 - 1) + v1*(n1 - 1),
                           altura, v0*-1, n0 - 1, v1*-1, n1 - 1)
  end
```

Para um exemplo realista podemos considerar o templo de Segesta que se encontra representado na Figura 2.12. Este templo é do tipo peristilo, composto por 6 colunas (i.e., Hexastilo) em cada fronte e 14 colunas nos lados, num total de 36 colunas de 9 metros de altura. A distância entre os eixos das colunas é de aproximadamente 4.8 metros nas frontes e de 4.6 nos lados. A expressão que cria o peristilo deste templo é, então:

```
peristilo_dorico(xy(0, 0), 9, vxy(4.8, 0), 6, vxy(0, 4.6), 14)
```

O resultado da avaliação da expressão anterior está representado na Figura 3.7.

Embora a grande maioria dos templos Gregos fosse de formato rectangular, também foram construídos templos de formato circular, a que chamaram *Tholos*. O Santuário de Atenas Pronaia, em Delfos, contém um bom exemplo de um destes edifícios. Embora pouco reste deste templo, não é difícil imaginar a sua forma original a partir do que ainda é visível na Figura 3.8.

Para simplificar a construção do *Tholos*, vamos dividi-lo em duas partes. Numa, iremos desenhar a base e, na outra, iremos posicionar as colunas.

Figura 3.8: O Templo de Atenas Pronaia em Delfos, construído no século quarto antes de Cristo. Fotografia de Michelle Kelley.

Para o desenho da base, podemos considerar um conjunto de cilindros achatados, sobrepostos de modo a formar degraus circulares, tal como se apresenta na Figura 3.9. Desta forma, a altura total da base a_b será dividida em passos de Δa_b e o raio da base também será reduzido em passos de Δr_b.

Para cada cilindro teremos de considerar o seu `raio` e a altura do espelho do degrau `d_altura`. Para passarmos ao cilindro seguinte temos ainda de ter em conta o aumento do raio `d_raio` devido ao comprimento do cobertor do degrau. Estes degraus serão construídos segundo um processo recursivo:

- Se o número de degraus a colocar é zero, não é preciso fazer nada.

- Caso contrário, colocamos um degrau (modelado por um cilindro) com o raio e a altura do degrau e, recursivamente, colocamos os restantes degraus em cima deste, i.e., numa cota igual à altura do degrau agora colocado e com um raio reduzido do comprimento do cobertor do degrau agora colocado.

Este processo é implementado pela seguinte função:

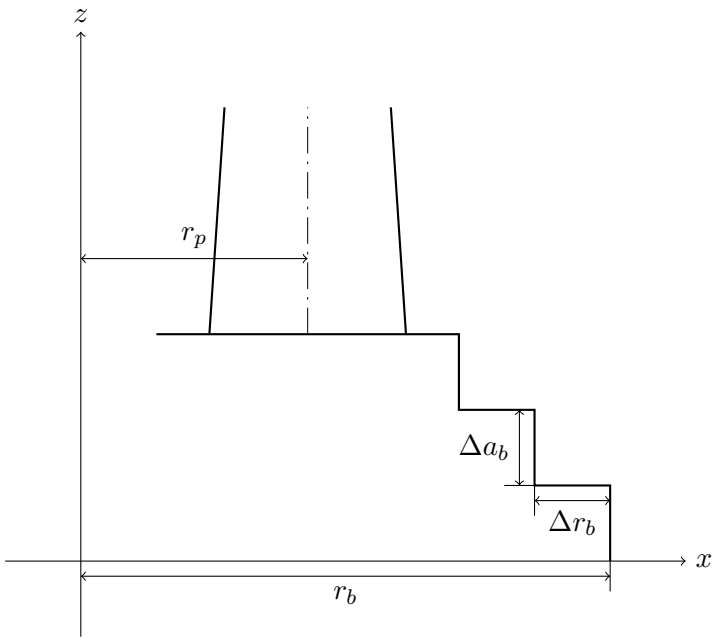


Figura 3.9: Corte da base de um *Tholos*. A base é composta por uma sequência de cilindros sobrepostos cujo raio de base r_b encolhe de Δr_b a cada degrau e cuja altura incrementa Δa_b a cada degrau.

```
base_tholos(p, n_degraus, raio, d_altura, d_raio) =
  if n_degraus == 0
    nothing
  else
    cylinder(p, raio, d_altura)
    base_tholos(p + vxyz(0, 0, d_altura),
                n_degraus - 1, raio - d_raio, d_altura, d_raio)
  end
```

Para o posicionamento das colunas, vamos também considerar um processo em que em cada passo apenas posicionamos uma coluna numa dada posição e, recursivamente, colocamos as restantes colunas a partir da posição circular seguinte.

Dada a sua estrutura circular, a construção deste género de edifícios é simplificada pelo uso de coordenadas circulares. De facto, podemos conceber um processo recursivo que, a partir do raio r_p do peristilo e do ângulo inicial ϕ, coloca uma coluna nessa posição e que, de seguida, coloca as restantes colunas usando o mesmo raio mas incrementando o ângulo ϕ de $\Delta\phi$, tal como se apresenta na Figura 3.10. O incremento angular $\Delta\phi$ obtém-se pela divisão da circunferência pelo número n de colunas a colocar, i.e., $\Delta\phi = \frac{2\pi}{n}$. Uma vez que as colunas se dispõem em torno de um círculo, o cálculo das coordenadas de cada coluna fica facilitado pelo uso de coor-

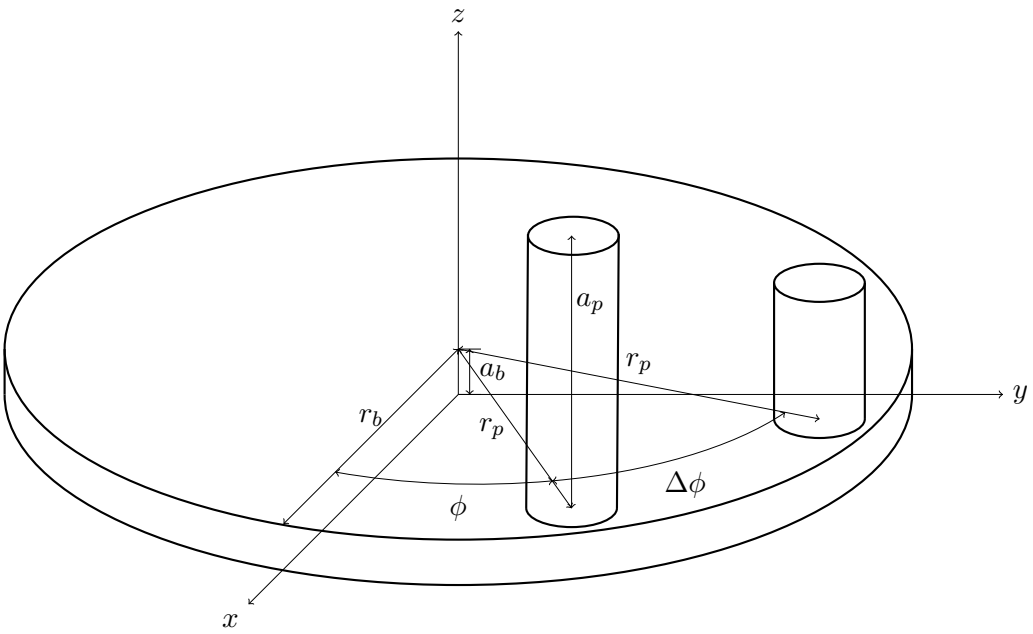


Figura 3.10: Esquema da construção de um *Tholos*: r_b é o raio da base, r_p é a distância do centro das colunas ao centro da base, a_p é a altura das colunas, a_b é a altura da base, ϕ é o ângulo inicial de posicionamento das colunas e $\Delta\phi$ é o ângulo entre colunas.

denadas polares. Tendo este algoritmo em mente, a definição da função fica:

```
colunas_tholos(p, n_colunas, raio, fi, d_fi, altura) =
  if n_colunas == 0
    nothing
  else
    coluna_dorica(p + vpol(raio, fi), altura)
    colunas_tholos(p, n_colunas - 1, raio, fi + d_fi, d_fi, altura)
  end
```

Finalmente, definimos a função `tholos` que, dados os parâmetros necessários às duas anteriores, as invoca em sequência:

```
tholos(p, n_degraus, rb, dab, drb, n_colunas, rp, ap) =
  begin
    base_tholos(p, n_degraus, rb, dab, drb)
    colunas_tholos(p + vz(n_degraus*dab),
                   n_colunas, rp, 0, 2*pi/ n_colunas, ap)
  end
```

A Figura 3.11 mostra a imagem gerada pela avaliação da seguinte expressão:

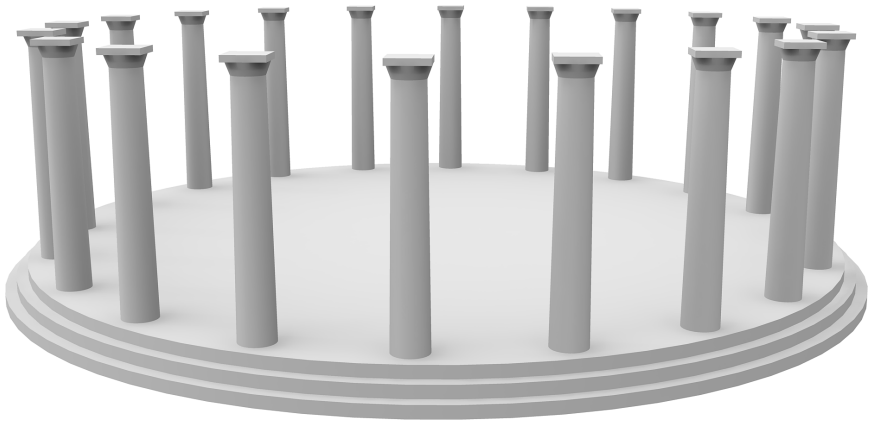
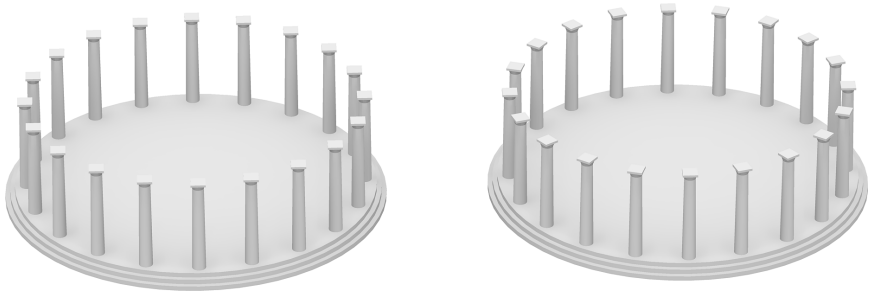


Figura 3.11: Perspectiva do *Tholos* de Atenas em Delfos, constituído por 20 colunas de estilo Dórico, de 4 metros de altura e colocadas num círculo com 7 metros de raio.

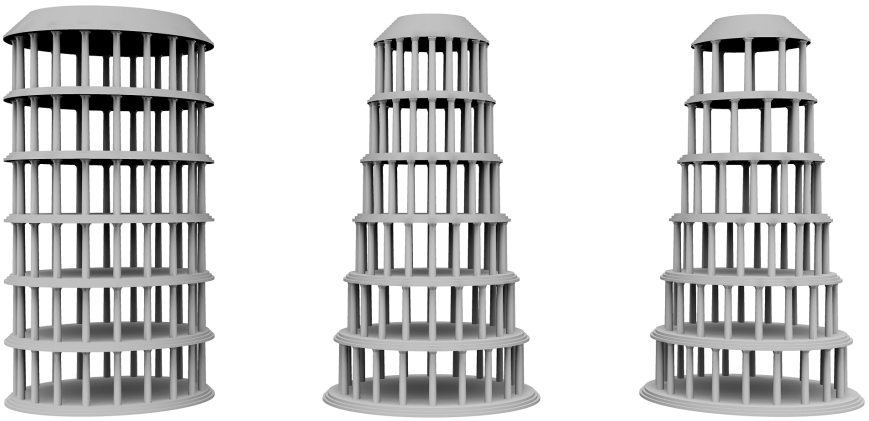
```
tholos(xyz(0, 0, 0), 3, 7.9, 0.2, 0.2, 20, 7, 4)
```

Exercício 3.3.2 Uma observação atenta do *Tholos* apresentado na Figura 3.11 mostra que existe um erro: os ábacos das várias colunas são paralelos uns aos outros (e aos eixos das abcissas e ordenadas) quando, na realidade, deveriam ter uma orientação radial. Essa diferença é evidente quando se compara uma vista de topo do desenho actual (à esquerda) com a mesma vista daquele que seria o desenho correcto (à direita):



Redefina a função `colunas_tholos` de modo a que cada coluna esteja orientada correctamente relativamente ao centro do *Tholos*.

Exercício 3.3.3 Considere a construção de uma torre composta por vários módulos em que cada módulo tem exactamente as mesmas características de um *Tholos*, tal como se apresenta na figura abaixo, à esquerda:



O topo da torre tem uma forma semelhante à da base de um *Tholos*, embora com mais degraus.

Defina a função `torre_tholos` que, a partir do centro da base da torre, do número de módulos, do número de degraus a considerar para o topo e dos restantes parâmetros necessários para definir um módulo idêntico a um *Tholos*, constrói a torre apresentada anteriormente.

Experimente a sua função criando uma torre composta por 6 módulos, com 10 degraus no topo, 3 degraus por módulo, qualquer deles com comprimento de espelho e de cobertor de 0.2, raio da base de 7.9 e 20 colunas por módulo, com raio de peristilo de 7 e altura de coluna de 4.

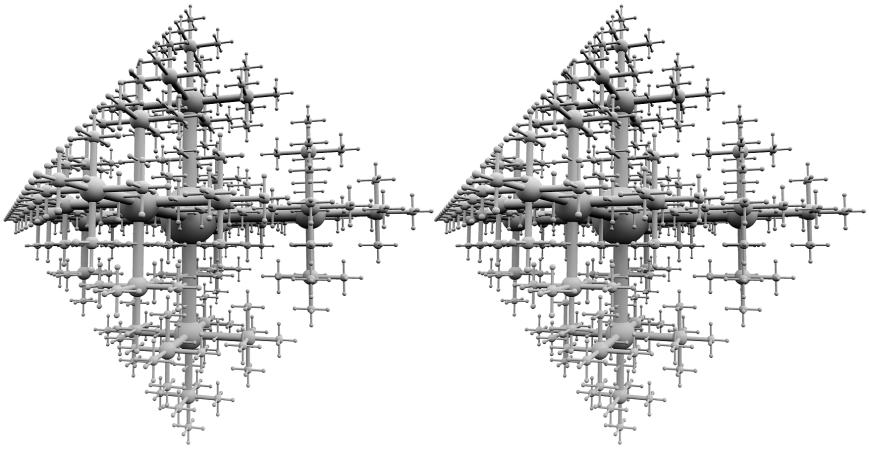
Exercício 3.3.4 Com base na resposta ao exercício anterior, redefina a construção da torre de forma a que a dimensão radial dos módulos se vá reduzindo à medida que se ganha altura, tal como acontece na torre apresentada no centro da imagem anterior.

Exercício 3.3.5 Com base na resposta ao exercício anterior, redefina a construção da torre de forma a que o número de colunas se vá também reduzindo à medida que se ganha altura, tal como acontece na torre da direita da imagem anterior.

Exercício 3.3.6 Considere a criação de uma cidade no espaço, composta apenas por cilindros com dimensões progressivamente mais pequenas, unidos uns aos outros por intermédio de esferas, tal como se apresenta (em perspectiva) na seguinte imagem estereoscópica:[1]

[1]Para visualizar a imagem estereoscópica, foque a atenção no meio das duas imagens e cruze os olhos, como se quisesse focar um objecto muito próximo. Irá reparar que as duas imagens passaram a ser quatro, embora ligeiramente desfocadas. Tente então alterar o cruzar dos olhos de modo a só ver três imagens, i.e., até que as duas imagens centrais fiquem sobrepostas. Concentre-se nessa sobreposição e deixe os olhos relaxarem até a imagem ficar focada.

Figura 3.12: Volutas de um capitel Jónico. Fotografia de See Wah Cheng.



Defina uma função que, a partir do centro da cidade e do raio dos cilindros centrais constrói uma cidade semelhante à representada.

3.4 A Ordem Jónica

A *voluta* foi um dos elementos arquitectónicos introduzidos na transição da Ordem Dórica para a Ordem Jónica. Uma voluta é um ornamento em forma de espiral colocado em cada extremo de um capitel Jónico. A Figura 3.12 mostra um exemplo de um capitel Jónico contendo duas volutas. Embora tenham sobrevivido inúmeros exemplos de volutas desde a antiguidade, nunca foi claro o processo do seu desenho.

Vitrúvio, no seu tratado de arquitectura, descreve a voluta Jónica: uma curva em forma de espiral que se inicia na base do ábaco, desenrola-se

numa série de voltas e junta-se a um elemento circular denominado o *olho*.

Vitrúvio descreve o processo de desenho da espiral através da composição de quartos de circunferência, começando pelo ponto mais exterior e diminuindo o raio a cada quarto de circunferência, até se dar a conjunção com o olho. Nesta descrição há ainda alguns detalhes por explicar, em particular, o posicionamento dos centros dos quartos de circunferência, mas Vitrúvio refere que será incluida uma figura e um cálculo no final do livro.

Infelizmente, nunca se encontrou essa figura ou esse cálculo, ficando assim por esclarecer um elemento fundamental do processo de desenho de volutas descrito por Vitrúvio. As dúvidas sobre esse detalhe tornaram-se ainda mais evidentes quando a análise de muitas das volutas que sobreviveram a antiguidade revelou diferenças em relação às proporções descritas por Vitrúvio.

Durante a Renascença, estas dúvidas levaram os investigadores a repensar o método de Vitrúvio e a sugerir interpretações pessoais ou novos métodos para o desenho da voluta. De particular relevância foram os métodos propostos no século XVI por Sebastiano Serlio, baseado na composição de semi-circunferências, por Giuseppe Salviati, baseado na composição de quartos-de-circunferência e por Guillaume Philandrier, baseado na composição de oitavos-de-circunferência.

Todos estes métodos diferem em vários detalhes mas, de forma genérica, todos se baseiam em empregar arcos de circunferência de abertura constante mas raio decrescente. Obviamente, para que haja continuidade nos arcos, os centros dos arcos vão mudando à medida que estes vão sendo desenhados. A Figura 3.13 esquematiza o processo para espirais feitas empregando quartos de circunferência.

Como se vê pela figura, para se desenhar a espiral temos de ir desenhando sucessivos quartos de circunferência. O primeiro quarto de circunferência será centrado no ponto P e terá raio r. Este primeiro arco vai desde o ângulo $\pi/2$ até ao ângulo π. O segundo quarto de circunferência será centrado no ponto P_1 e terá raio $r \cdot f$, sendo f um coeficiente de "redução" da espiral. Este segundo arco vai desde o ângulo π até ao ângulo $\frac{3}{2}\pi$. Um detalhe importante é a relação entre as coordenadas P_1 e P: para que o segundo arco tenha uma extremidade coincidente com o primeiro arco, o seu centro tem de estar na extremidade do vector $\vec{v}_0$ de origem em P, comprimento $r(1-f)$ e ângulo igual ao ângulo final do primeiro arco.

Este processo deverá ser seguido para todos os restantes arcos de circunferência, i.e., teremos de calcular as coordenadas P_2, P_3, etc., bem como os raios $r \cdot f \cdot f$, $r \cdot f \cdot f \cdot f$, etc, necessários para traçar os sucessivos arcos de circunferência.

Dito desta forma, o processo de desenho parece ser complicado. No entanto, é possivel reformulá-lo de modo a ficar muito mais simples. De facto, podemos pensar no desenho da espiral completa como sendo o desenho de um quarto de circunferência seguido do desenho de uma espiral

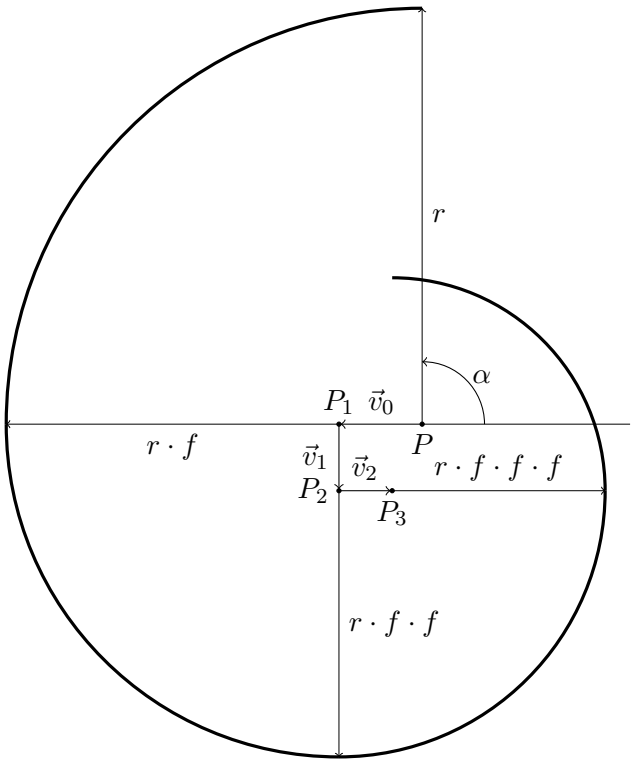


Figura 3.13: O desenho de uma espiral com arcos de circunferência.

mais pequena. Mais concretamente, podemos especificar o desenho da espiral de centro no ponto P, raio r e ângulo inicial α como sendo o desenho de um arco de circunferência de raio r centrado em P com ângulo inicial α e final $\alpha + \frac{\pi}{2}$, seguido de uma espiral de centro em $P + \vec{v}$, raio $r \cdot f$ e ângulo inicial $\alpha + \frac{\pi}{2}$. O vector $\vec{v}$ terá origem em P, módulo $r(1 - f)$ e ângulo $\alpha + \frac{\pi}{2}$.

Obviamente, sendo este um processo recursivo, é necessário definir o caso de paragem, havendo (pelo menos) três possibilidades:

- Terminar quando o número de quartos de círculo é zero.
- Terminar quando o raio r é inferior a um determinado limite.
- Terminar quando o ângulo α é superior a um determinado limite.

Por agora, vamos considerar a primeira possibilidade. De acordo com o nosso raciocínio, vamos definir a função que desenha a espiral de modo a receber, como parâmetros, o ponto inicial `p`, o raio inicial `r`, o ângulo inicial `a`, o número de quartos de círculo `n`, e o factor de redução `f`:

```
espiral(p, r, a, n, f) =
  if n == 0
    nothing
  else
    quarto_circunferencia(p, r, a)
    espiral(p + vpol(r*(1 - f), a + pi/2), r*f, a + pi/2, n - 1, f)
  end
```

Reparemos que a função `espiral` é recursiva, pois está definida em termos de si própria. Obviamente, o caso recursivo é mais simples que o caso original, pois o número de quartos de círculo é mais pequeno, aproximando-se progressivamente do caso de paragem.

Para desenhar o quarto de circunferência vamos empregar a operação `arc` do Julia que recebe o centro e o raio da circunferência, e o ângulo inicial e final do arco. Para melhor percebermos o processo de desenho da espiral vamos também traçar duas linhas com origem no centro a delimitar cada quarto de circunferência. Mais tarde, quando tivermos terminado o desenvolvimento destas funções, removeremos essas linhas.

```
quarto_circunferencia(p, r, a) =
  begin
    arc(p, r, a, pi/2)
    line(p, p + vpol(r, a))
    line(p, p + vpol(r, a + pi/2))
  end
```

Podemos agora experimentar um exemplo:

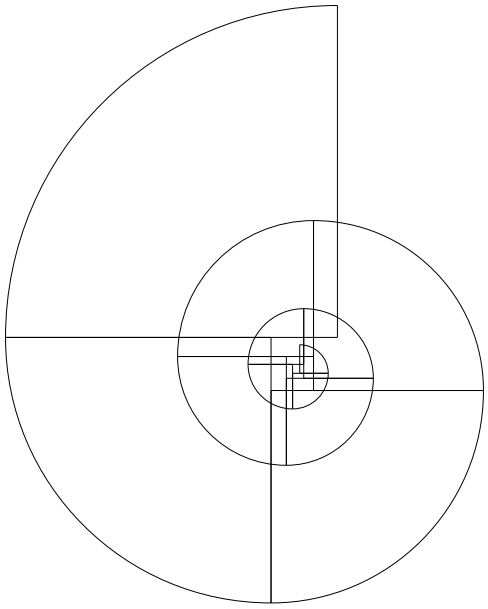


Figura 3.14: O desenho da espiral.

```
espiral(xy(0, 0), 10, pi/2, 12, 0.8)
```

A espiral traçada pela expressão anterior está representada na Figura 3.14.

A função `espiral` permite-nos definir um sem-número de espirais, mas tem uma restrição: cada arco de círculo corresponde a um incremento de ângulo de $\frac{\pi}{2}$. Logicamente, a função tornar-se-á mais útil se também este incremento de ângulo for um parâmetro.

Tal como se pode deduzir da observação da Figura 3.15, as modificações a fazer são relativamente triviais, bastando acrescentar um parâmetro `da`, representando o incremento de ângulo Δ_α de cada arco e substituir as ocorrências de $\frac{\pi}{2}$ por este parâmetro. Naturalmente, em vez de desenharmos um quarto de circunferência, temos agora de desenhar um arco de circunferência de amplitude angular Δ_α. Uma vez que a utilização deste parâmetro afecta também o significado do parâmetro n, que agora representará o número de arcos com aquela amplitude, é preferível explorarmos uma condição de paragem diferente, baseada no ângulo final `af` que pretendemos atingir. Temos, contudo, que ter atenção a um detalhe: o último arco pode não ser completo se a diferença entre o ângulo final e o inicial exceder o incremento de ângulo. Neste caso, o arco terá apenas essa diferença de ângulo. A nova definição é, então:

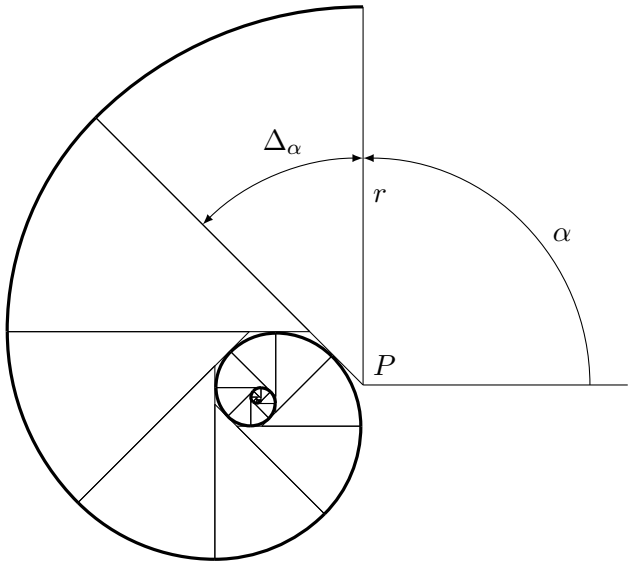


Figura 3.15: Espiral com incremento de ângulo como parâmetro.

```
espiral(p, r, a, da, af, f) =
  if af - a < da
    arco_espiral(p, r, a, af - a)
  else
    arco_espiral(p, r, a, da)
    espiral(p + vpol(r*(1 - f), a + da), r*f, a + da, da, af, f)
  end
```

A função que desenha o arco é uma generalização da que desenha o quarto de circunferência:

```
arco_espiral(p, r, a, da) =
  begin
    arc(p, r, a, da)
    line(p, p + vpol(r, a))
    line(p, p + vpol(r, a + da))
  end
```

Agora, para desenhar a mesma espiral representada na Figura 3.14, temos de avaliar a expressão:

```
espiral(xy(0, 0), 10, pi/2, pi/2, pi*6, 0.8)
```

É claro que agora podemos facilmente construir outras espirais. As seguintes expressões produzem as espirais representadas na Figura 3.16:

```
espiral(xy(0, 0), 10, pi/2, pi/2, pi*6, 0.9)
espiral(xy(20, 0), 10, pi/2, pi/2, pi*6, 0.7)
espiral(xy(40, 0), 10, pi/2, pi/2, pi*6, 0.5)
```

Outra possibilidade de variação está no ângulo de incremento. As seguintes expressões experimentam aproximações aos processos de Sebasti-

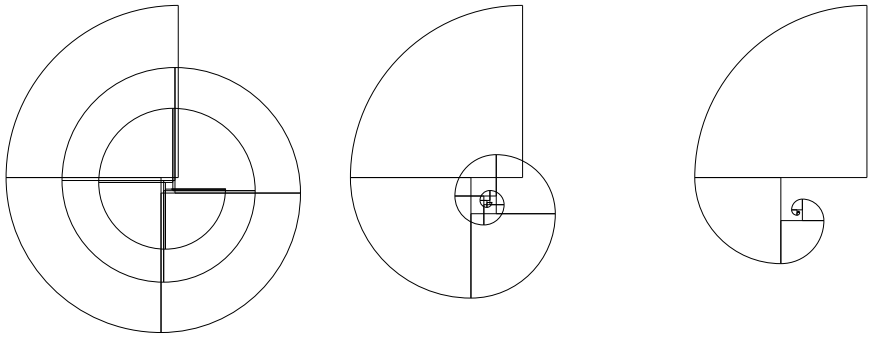


Figura 3.16: Várias espirais com razões de redução de 0.9, 0.7 e 0.5, respectivamente.

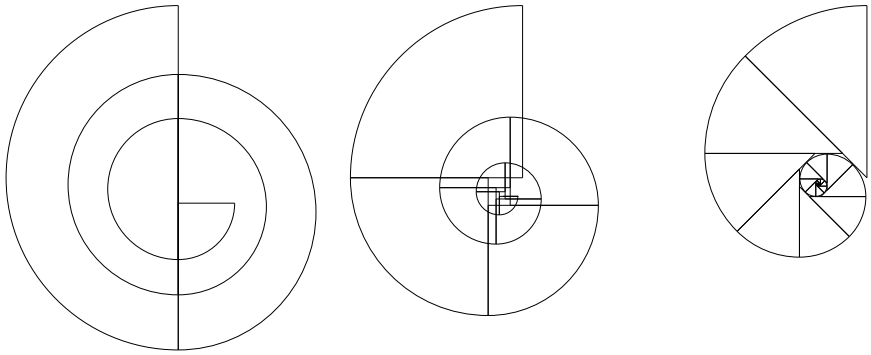


Figura 3.17: Várias espirais com razão de redução de 0.8 e incremento de ângulo de π, $\frac{\pi}{2}$ e $\frac{\pi}{4}$, respectivamente.

ano Serlio (semi-circunferências), Giuseppe Salviati (quartos-de-circunferência) e Guillaume Philandrier (oitavos-de-circunferência):[2]

```
espiral(xy(0, 0), 10, pi/2, pi, pi*6, 0.8)
espiral(xy(20, 0), 10, pi/2, pi/2, pi*6, 0.8)
espiral(xy(40, 0), 10, pi/2, pi/4, pi*6, 0.8)
```

Os resultados estão representados na Figura 3.17.

Finalmente, para podermos comparar os diferentes processos de construção de espirais, convém ajustarmos o factor de redução ao incremento de ângulo, de modo a que a redução se aplique a uma volta completa e não apenas ao incremento de ângulo usado. Assim, tempos as seguintes expressões:

[2]Note-se que se trata, tão somente, de aproximações. Os processos originais eram bastante mais complexos.

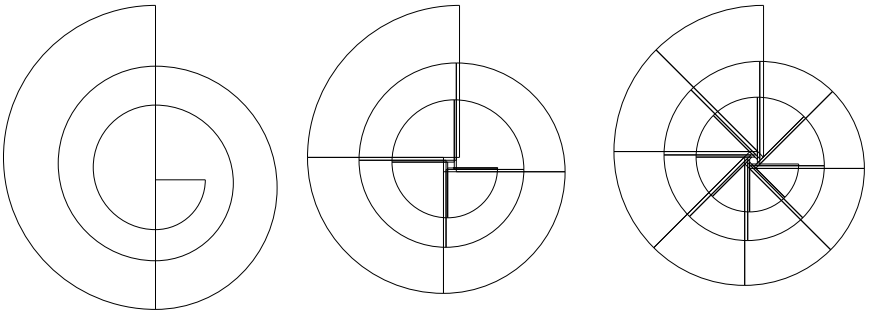


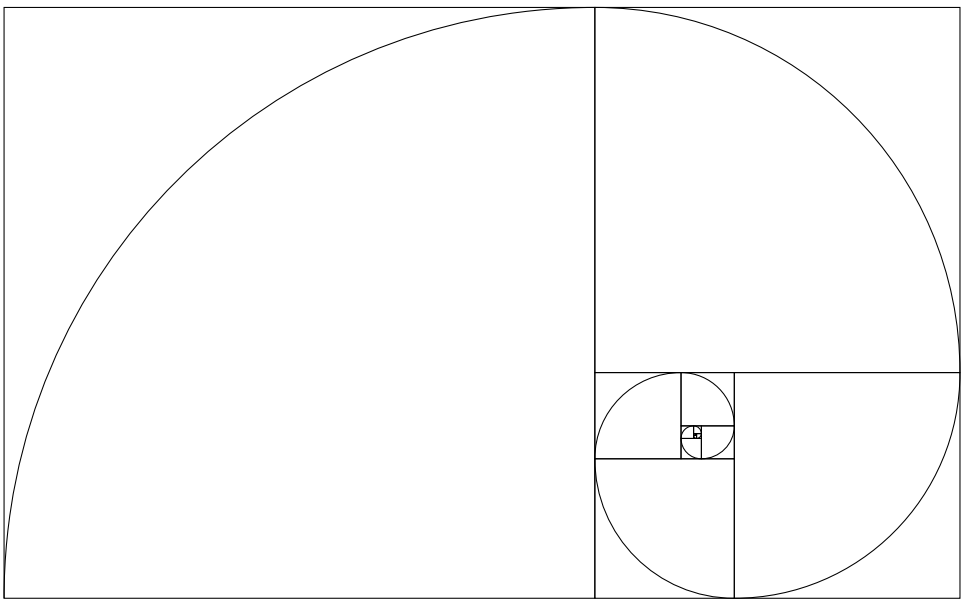
Figura 3.18: Várias espirais com razão de redução de 0.8 *por volta* e incremento de ângulo de π, $\frac{\pi}{2}$ e $\frac{\pi}{4}$, respectivamente.

```
espiral(xy(0, 0), 10, pi/2, pi, pi*6, 0.8)
espiral(xy(20, 0), 10, pi/2, pi/2, pi*6, 0.8^(1/2))
espiral(xy(40, 0), 10, pi/2, pi/4, pi*6, 0.8^(1/4))
```

Os resultados estão visiveis na Figura 3.18.

Exercício 3.4.1 A *espiral de ouro* é uma espiral cuja razão de crescimento é φ, sendo $\varphi = \frac{1+\sqrt{5}}{2}$ a *proporção de ouro*, popularizada por Luca Pacioli na sua obra *Divina Proportione* de 1509, embora haja inúmeros registos da sua utilização em datas muito anterioes.

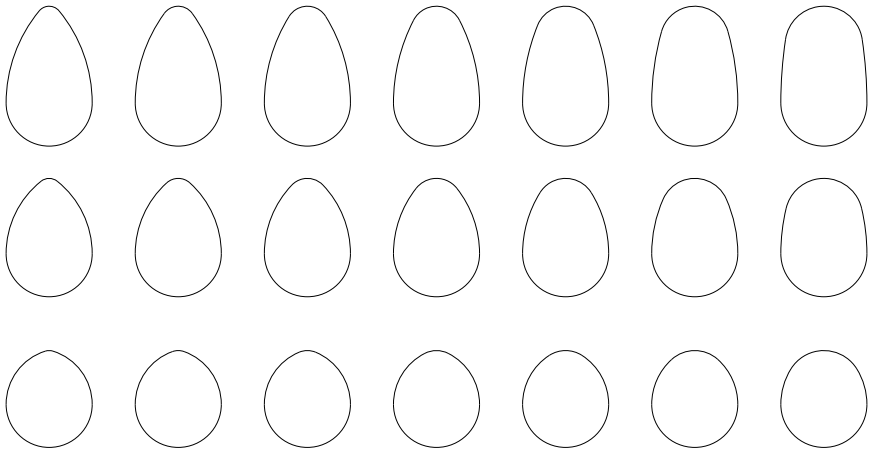
O seguinte desenho ilustra uma espiral de ouro realizada com arcos de círculo e contida no correspondente *rectângulo de ouro*, um rectângulo cujo lado maior é φ vezes o lado menor.



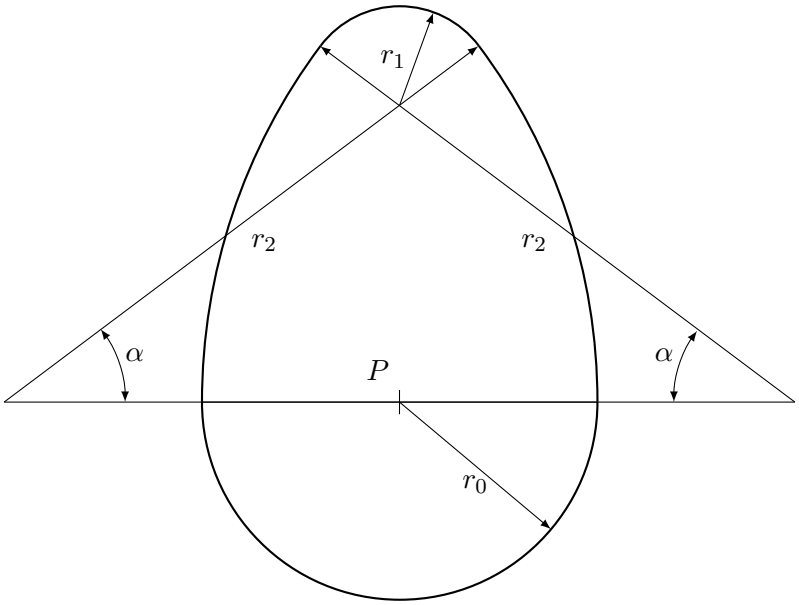
Como podemos observar no desenho anterior, o rectângulo de ouro tem a extraordinária propriedade de ser recursivamente (e infinitamente) decomponível num quadrado e noutro rectângulo de ouro.

Redefina a função `arco_espiral` de modo a que, para além de criar o arco, a função crie também um quadrado envolvente. De seguida, escreva a expressão Julia que cria a espiral de ouro anterior.

Exercício 3.4.2 Um *óvulo* é uma figura geomética que corresponde ao contorno dos ovos das aves. Dada a variedade de formas de ovos na Natureza, é natural considerar que também existe uma grande variedade de óvulos geométricos. A figura seguinte apresenta alguns exemplos em que se variam sistematicamente alguns dos parâmetros que caracterizam o óvulo:



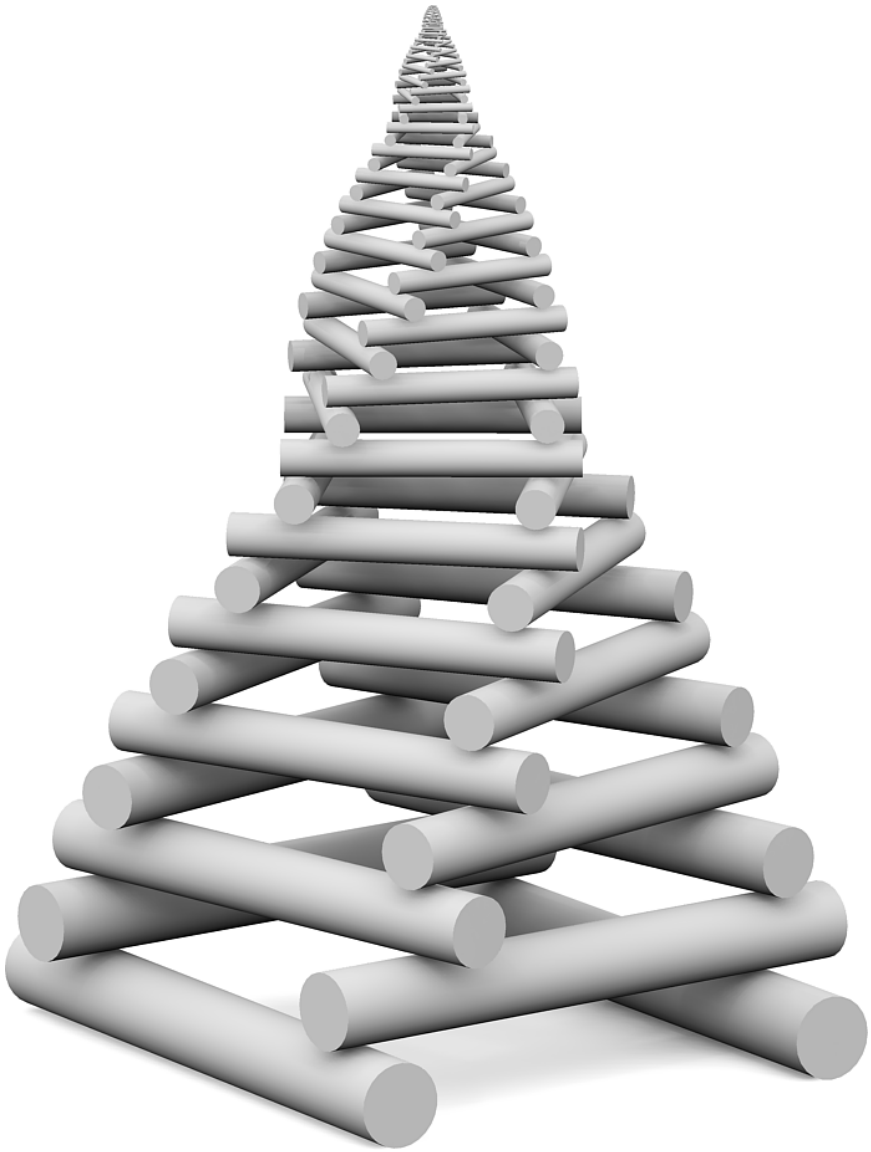
Um óvulo é composto por quatro arcos de circunferência concordantes, tal como se apresenta na imagem seguinte:



Os arcos de círculos necessários para a construção do ovo são caracterizados pelos raios r_0, r_1 e r_2. Note que o arco de circunferência de raio r_0 cobre um ângulo de π e o arco de circunferência de raio r_2 cobre um ângulo de α.

Defina uma função `ovo` que desenha um ovo. A função deverá receber, apenas, as coordenadas do ponto P, os raios r_0 e r_1 e, finalmente, a altura h do ovo.

Exercício 3.4.3 Defina a função `piramide_cilindros` capaz de construir uma pirâmide de cilindros empilhados uns sobre os outros, tal como se aresenta na imagem seguinte. Note que os cilindros vão diminuindo de tamanho (quer no comprimento, quer no raio) e vão sofrendo uma rotação à medida que vão sendo empilhados



3.5 Recursão na Natureza

A recursão está presente em inúmeros fenómenos naturais. As montanhas, por exemplo, apresentam irregularidades que, quando observadas numa escala apropriada, são em tudo idênticas a ... montanhas. Um rio possui afluentes e cada afluente é idêntico a ... um rio. Uma vaso sanguíneo possui ramificações e cada ramificação é idêntica a ... um vaso sanguíneo. Todas estas entidades naturais constituem exemplos de estruturas recursivas.

Uma árvore é outro bom exemplo de uma estrutura recursiva, pois os

Figura 3.19: A estrutura recursiva das árvores. Fotografia de Michael Bezzina.

ramos de uma árvore são como pequenas árvores que emanam do tronco. Como se pode ver da Figura 3.19, de cada ramo de uma árvore emanam outras pequenas árvores, num processo que se repete até se atingir uma dimensão suficientemente pequena para que apareçam outras estruturas, como folhas, flores, frutos, pinhas, etc.

Se, de facto, uma árvore possui uma estrutura recursiva, então deverá ser possível "construir" árvores através de funções recursivas. Para testarmos esta teoria, vamos começar por considerar uma versão muito simplista de uma árvore, em que temos um tronco que, a partir de uma certa altura, se divide em dois. Cada um destes subtroncos cresce fazendo um certo ângulo com o tronco de onde emanou e atinge um comprimento que deverá ser uma fracção do comprimento desse tronco, tal como se apresenta na Figura 3.20. O caso de paragem ocorre quando o comprimento do tronco se tornou tão pequeno que, em vez de se continuar a divisão, aparece simplesmente uma outra estrutura. Para simplificar, vamos designar a extremidade de um ramo por folha e iremos representá-la com um pequeno círculo.

Para darmos dimensões à árvore, vamos considerar que a função `arvore` recebe, como argumento, as coordenadas P da base da árvore, o comprimento c do tronco e o ângulo α do tronco. Para a fase recursiva, teremos como parâmetros a diferença de ângulo de abertura Δ_α que o novo tronco

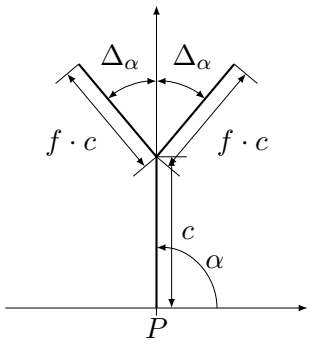


Figura 3.20: Parâmetros de desenho de uma árvore.

deverá fazer com o anterior e o factor de redução f do comprimento do tronco. O primeiro passo é a computação do topo do tronco. Em seguida, desenhamos o tronco desde a base até ao topo. Finalmente, testamos se o tronco desenhado é suficientemente pequeno. Se for, terminamos com o desenho de um círculo centrado no topo. Caso contrário fazemos uma dupla recursão para desenhar uma sub-árvore para a direita e outra para a esquerda. A definição da função fica:

```
arvore(p, c, a, da, f) =
  let topo = p + vpol(c, a)
    ramo(p, topo)
    if c < 2
      folha(topo)
    else
      arvore(topo, c*f, a + da, da, f)
      arvore(topo, c*f, a - da, da, f)
    end
  end

ramo(p, topo) =
  line(p, topo)

folha(topo) =
  circle(topo, 0.2)
```

Um primeiro exemplo de árvore gerado com a expressão

```
arvore(xy(0, 0), 20, pi/2, pi/8, 0.7)
```

está representado na Figura 3.21.

A Figura 3.22 apresenta outros exemplos em que se fez variar o ângulo de abertura e o factor de redução. A sequência de expressões que as gerou foi a seguinte:

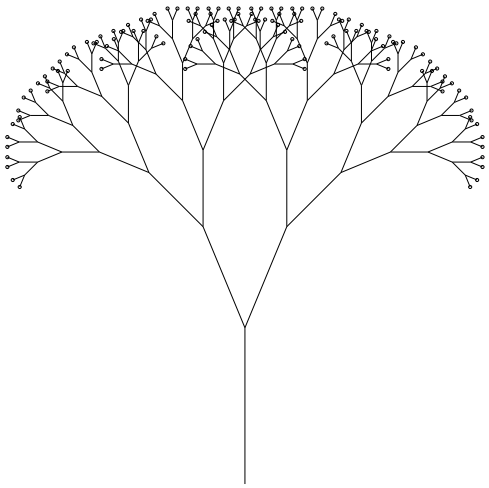


Figura 3.21: Uma "árvore" de comprimento 20, ângulo inicial $\frac{\pi}{2}$, abertura $\frac{\pi}{8}$ e factor de redução 0.7.

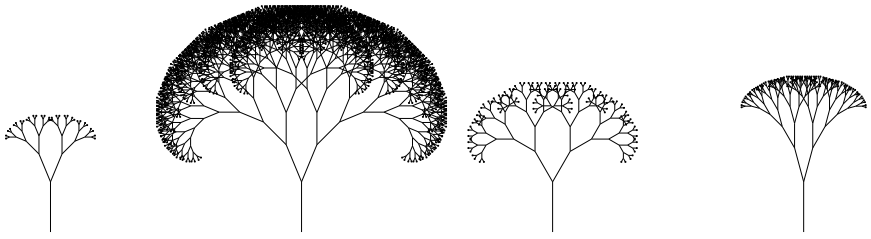


Figura 3.22: Várias "árvores" geradas com diferentes ângulos de abertura e factores de redução do comprimento dos ramos.

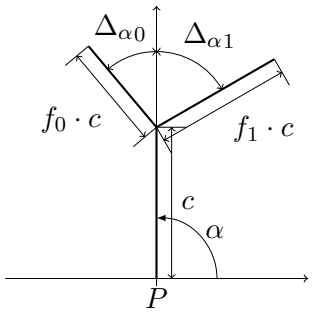


Figura 3.23: Parâmetros de desenho de uma árvore com crescimento assimétrico.

```
arvore(xy(0, 0), 20, pi/2, pi/8, 0.6)
arvore(xy(100, 0), 20, pi/2, pi/8, 0.8)
arvore(xy(200, 0), 20, pi/2, pi/6, 0.7)
arvore(xy(300, 0), 20, pi/2, pi/12, 0.7)
```

Infelizmente, as árvores apresentadas são "excessivamente" simétricas: no mundo natural é literalmente impossível encontrar simetrias perfeitas. Por este motivo, convém tornar o modelo um pouco mais sofisticado através da introdução de parâmetros diferentes para o crescimento dos troncos à direita e à esquerda. Para isso, em vez de termos um só ângulo de abertura e um só factor de redução de comprimento, vamos empregar dois, tal como se apresenta na Figura 3.23.

A adaptação da função `arvore` para lidar com os parâmetros adicionais é simples:

```
arvore(p, c, a, da0, f0, da1, f1) =
  let topo = p + vpol(c, a)
    ramo(p, topo)
    if c < 2
      folha(topo)
    else
      arvore(topo, c*f0, a + da0, da0, f0, da1, f1)
      arvore(topo, c*f1, a - da1, da0, f0, da1, f1)
    end
  end
```

A Figura 3.24 apresenta novos exemplos de árvores com diferentes ângulos de abertura e factores de redução dos ramos esquerdo e direito, geradas pelas seguintes expressões:

```
arvore(xy(0, 0), 20, pi/2, pi/8, 0.6, pi/8, 0.7)
arvore(xy(100, 0), 20, pi/2, pi/4, 0.7, pi/16, 0.7)
arvore(xy(200, 0), 20, pi/2, pi/6, 0.6, pi/16, 0.8)
```

As árvores geradas pela função `arvore` são apenas um modelo muito grosseiro da realidade. Embora existam sinais evidentes de que vários fe-

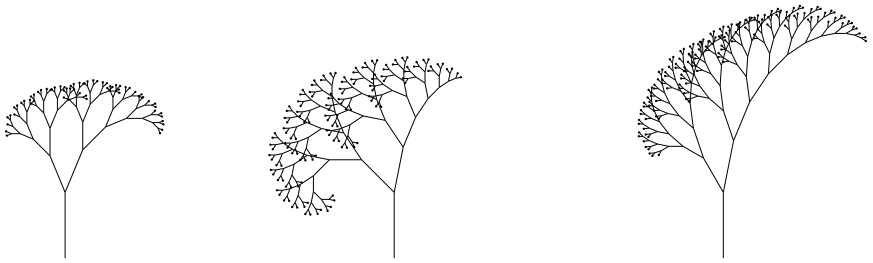


Figura 3.24: Várias árvores geradas com diferentes ângulos de abertura e factores de redução do comprimento para os ramos esquerdo e direito.

nómenos naturais se podem modelar através de funções recursivas, a natureza não é tão determinista quanto as nossas funções e, para que a modelação se aproxime mais da realidade, é fundamental incorporar também alguma aleatoriedade. Esse será o tema da próxima seccção.

Capítulo 4

Estado

4.1 Introdução

Como vimos, a linguagem Julia permite-nos estabelecer uma associação entre um nome e uma entidade através do operador =. Nesta secção, iremos ver que o Julia permite-nos também usar esse operador para alterar essa associação, fazendo com que o nome passe a estar associado a uma outra entidade. Esta operação denomina-se *atribuição*.

Nesta secção vamos discutir mais em profundidade o conceito de atribuição. Para motivar vamos começar por introduzir um tópico importante onde a atribuição tem um papel primordial: a aleatoriedade.

4.2 Aleatoriedade

Desenhar implica tomar decisões conscientes que conduzam a um objectivo pretendido. Neste sentido, desenhar aparenta ser um processo racional onde não há lugar para a *aleatoriedade*, a *sorte* ou a *incerteza*. De facto, nos desenhos que temos feito até agora a racionalidade tem sido crucial, pois o computador exige uma especificação rigorosa do que se pretende, não permitindo quaisquer ambiguidades. No entanto, é sabido que um problema de desenho frequentemente exige que o arquitecto experimente diferentes soluções antes de encontrar aquela que o satisfaz. Essa experimentação passa por empregar alguma aleatoriedade no processo de escolha dos "parâmetros" do desenho. Assim, embora um desenho final possa apresentar uma estrutura que espelha uma intenção racional do arquitecto, o processo que conduziu a esse desenho final não é necessariamente racional e pode ter passado por fases de ambiguidade e incerteza.

Quando a arquitectura se inspira na natureza surge um factor adicional de aleatoriedade: em inúmeros casos, a natureza é intrinsecamente aleatória. Essa aleatoriedade, contudo, não é total e está sujeita a restrições. Este facto é facilmente compreendido quando pensamos que, embora não

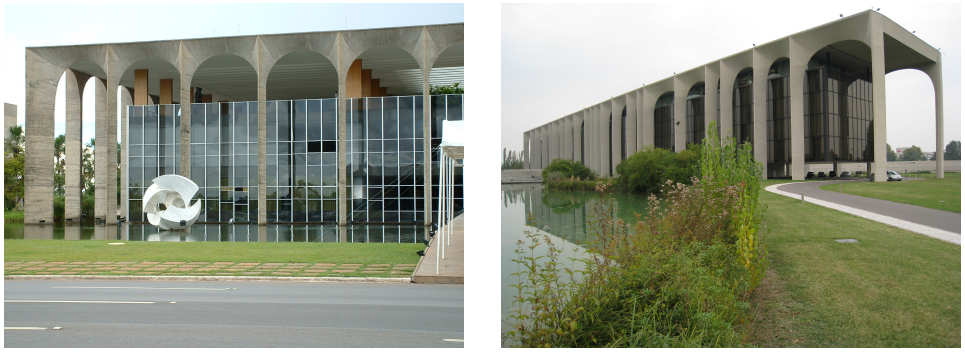


Figura 4.1: Duas obras de Oscar Niemeyer. À esquerda, o Palácio Itamaraty, projectado em 1962. À direita, o Palácio Mondadori, projectado em 1968. Fotografias de Bruno Kussler Marques e Tristan Nitot, respectivamente.

existam dois pinheiros iguais, todos somos capazes de reconhecer o padrão que caracteriza um pinheiro. Em todas as suas obras, a natureza combina regularidade e aleatoriedade. Nalguns casos, como o crescimento de cristais, por exemplo, há mais regularidade que aleatoriedade. Noutros casos, como no comportamento de partículas subatómicas, há mais aleatoriedade que regularidade.

À semelhança do que acontece na natureza, esta combinação de aleatoriedade e regularidade é claramente visível em inúmeras obras arquitecturais modernas. A título de exemplo, consideremos duas importantes obras do arquitecto Oscar Niemeyer: o Palácio Itamaraty e o Palácio Mondadori, visíveis na Figura 4.1. Apesar das notórias semelhança entre estas duas obras, a primeira prima pela regularidade das arcadas, em oposição à aleatoriedade evidente que Niemeyer empregou na segunda.

Se pretendemos empregar computadores para o desenho e este desenho pressupõe aleatoriedade, então teremos de ser capazes de a incorporar nos nossos algoritmos. A aleatoriedade pode ser incorporada de inúmeras formas, como, por exemplo:

- A cor de um artefacto pode ser escolhida aleatoriamente.
- O comprimento de uma parede pode ser um valor aleatório entre determinados limites.
- A decisão de dividir uma área ou mantê-la íntegra pode ser tomada aleatoriamente.
- A forma geométrica a empregar para um determinado elemento arquitectónico pode ser escolhida aleatoriamente.

4.2.1 Números Aleatórios

Em qualquer dos casos anteriores, podemos reduzir a aleatoriedade à escolha de números dentro de certos limites. Para uma cor aleatória, podemos gerar aleatoriamente três números que representem os valores RGB[1] da cor. Para um comprimento aleatório, podemos gerar aleatoriamente um número dentro dos limites desse comprimento. Para uma decisão lógica, podemos gerar aleatoriamente um número inteiro entre zero e um, decidindo de um modo se o número for zero, e de outro modo se o número for um. Para uma escolha aleatória de entre um conjunto de alternativas podemos simplesmente gerar um número aleatório entre um e o número de elementos do conjunto e escolher a alternativa correspondente ao número aleatório gerado.

Estes exemplos mostram-nos que o fundamental é conseguirmos gerar um número aleatório dentro de um certo intervalo. A partir dessa operação podemos implementar todas as outras.

Há dois processos fundamentais para se gerarem números aleatórios. O primeiro processo baseia-se na medição de processos físicos intrínsecamente aleatórios, como seja o ruido electrónico ou o decaimento radioactivo. O segundo processo baseia-se na utilização de funções aritméticas que, a partir de um valor inicial (denominado a *semente*), produzem uma sequência de números aparentemente aleatórios, sendo cada número da sequência gerado a partir do número anterior. Neste caso dizemos que estamos perante um *gerador de números pseudo-aleatórios*. O termo *pseudo* justifica-se, pois, se repetirmos o valor da semente original, repetiremos também a sequência de números de gerados.

Muito embora um gerador de números pseudo-aleatórios gere uma sequência de números que, na verdade, não são aleatórios, ele possui duas importantes vantagens:

- Pode ser facilmente implementado usando uma qualquer linguagem de programação, não necessitando de outros mecanismos adicionais para se obter a fonte de aleatoriedade.

- É frequente os nossos programas conterem erros. Para identificarmos a causa do erro pode ser necessário reproduzirmos exactamente as mesmas condições que deram origem ao erro. Para além disso, depois da correcção do erro é necessário repetirmos novamente a execução do programa para nos certificarmos que o erro não ocorre de novo. Infelizmente, quando o comportamento de um programa depende de uma sequência de números verdadeiramente aleatórios torna-se impossível reproduzir as condições de execução: da próxima

[1] RGB é a abreviatura de *Red-Green-Blue*, um modelo de cores em que qualquer cor é vista como uma combinação das três cores primárias vermelho, verde e azul.

vez que executarmos o programa, a sequência de números aleatórios será quase de certeza diferente.

Por estes motivos, de agora em diante vamos apenas considerar geradores de números pseudo-aleatórios, que iremos abusivamente designar por geradores de números aleatórios. Um gerador deste tipo caracteriza-se por uma função f que, a partir de um argumento x_i, produz um número $x_{i+1} = f(x_i)$ *aparentemente* não relacionado com x_i. A semente do gerador é o elemento x_0 da sequência.

Falta agora encontramos uma função f apropriada. Para isso, entre outras qualidades, exige-se que os números gerados por f sejam *equiprováveis*, i.e., todos os números dentro de um determinado intervalo têm igual probabilidade de serem gerados e que a sequência de números gerados tenha um *período* tão grande quanto possível, i.e., que a sequência dos números gerados só se comece a repetir ao fim de muito tempo.

Têm sido estudadas inúmeras funções com estas características. A mais utilizada é denominada *função geradora congruencial linear* que é da forma

$$x_{i+1} = (ax_i + b) \bmod m$$

Por exemplo, se tivermos $a = 25173, b = 13849, m = 65536$ e começarmos com uma semente qualquer, por exemplo, $x_0 = 12345$, obtemos a sequência de números pseudo-aleatórios[2] $2822, 11031, 21180, 42629, 27202, 49667, 50968, 33041, 37566, 43823, 2740, 43997, 57466, 29339, 39312, 21225, 61302, 58439, 12204, 57909, 39858, 3123, 51464, 1473, 302, 13919, 41380, 43405, 31722, 61131, 13696, 63897, 42982, 375, 16540, 25061, 24866, 31331, 48888, 36465, \ldots$

Podemos facilmente confirmar estes resultados usando o Julia:

```
proximo_aleatorio(ultimo_aleatorio) =
 (25173*ultimo_aleatorio + 13849)%65536

julia> proximo_aleatorio(12345)
2822
julia> proximo_aleatorio(2822)
11031
julia> proximo_aleatorio(11031)
21180
```

Esta abordagem, contudo, implica que só podemos gerar um número "aleatório" x_{1+i} se nos recordarmos do número x_i gerado imediatamente antes para o podermos dar como argumento à função `proximo_aleatorio`. Infelizmente, o momento e o ponto do programa em que podemos precisar de um novo número aleatório pode ocorrer muito mais tarde e muito

[2]Na verdade, esta sequência não é suficientemente aleatória pois existe um padrão que se repete continuamente. Consegue descobri-lo?

mais longe do que o momento e o ponto do programa em que foi gerado o último número aleatório, o que complica substancialmente a escrita do programa. Seria preferível que, ao invés da função `proximo_aleatorio`, que depende do último valor gerado x_i, tivéssemos uma função `aleatorio` que não precisasse de receber o último número gerado para conseguir gerar o próximo número. Assim, em qualquer ponto do programa em que precisássemos de gerar um novo número aleatório, limitávamo-nos a invocar a função `aleatorio` sem termos de nos recordar do último número gerado. Partindo do mesmo valor da semente, teriamos:

```
julia> aleatorio()
2822
julia> aleatorio()
11031
julia> aleatorio()
21180
```

4.3 Estado

A função `aleatorio` apresenta um comportamento diferente do das funções que vimos até agora. Até este momento, todas as funções que tínhamos definido comportavam-se como as funções matemáticas tradicionais: dados argumentos, a função produz resultados e, mais importante, dados os mesmos argumentos, a função produz *sempre* os mesmos resultados. Por exemplo, independentemente do número de vezes que invocarmos a função `factorial`, para um dado argumento ela irá sempre produzir o factorial desse argumento.

A função `aleatorio` é diferente de todas as outras, pois, para além de não precisar de argumentos, ela produz um resultado diferente de cada vez que é invocada.

Do ponto de vista matemático, uma função sem parâmetros não tem nada de anormal: é precisamente aquilo que se designa por uma *constante*. De facto, tal como escrevemos $\sin\cos x$ para designar $\sin(\cos(x))$, também escrevemos $\sin\pi$ para designar $\sin(\pi())$, onde se vê que π pode ser interpretado como uma função sem argumentos.

Do ponto de vista matemático, uma função que produz resultados diferentes a cada invocação não tem nada de normal: é uma aberração, pois, de acordo com a definição matemática de função, esse comportamento não é possível. E, no entanto, é precisamente este o comportamento que gostaríamos de ter na função `aleatorio`.

Para se obter o pretendido comportamento é necessário introduzir o conceito de *estado*. Dizemos que uma função tem estado quando o seu comportamento depende da sua história, i.e., das suas invocações anteriores. A função `aleatorio` é um exemplo de uma função com estado, mas há inúmeros outros exemplos no mundo real. Uma conta bancária também tem

um estado que depende de todas as transacções anteriores que lhe tiverem sido feitas. O depósito de combustível de um carro também tem um estado que depende dos enchimentos e dos trajectos realizados.

Para que uma função possa ter história é necessário que tenha *memória*, i.e., que se possa recordar de acontecimentos passados para assim poder influenciar os resultados futuros. Até agora vimos que o operador = nos permitia *definir* associações entre nomes e valores, mas o que ainda não tínhamos discutido é a possibilidade de *modificar* essas associações, i.e., alterar o valor que estava associado a um determinado nome. É esta possibilidade que nos permite incluir memória nas nossas funções.

No caso da função `aleatorio` vimos que a memória que nos interessa é saber qual foi o último número aleatório gerado. Imaginemos então que tínhamos uma associação entre o nome `numero_aleatorio` e esse número. No momento inicial, naturalmente, esse nome deverá estar associado à semente da sequência de números aleatórios. Para isso, definimos:

```
numero_aleatorio = 12345
```

De seguida, já podemos definir a "função" `aleatorio` que não só usa o último valor associado àquele nome como actualiza essa associação com o novo valor gerado, i.e.:

```
aleatorio() =
  numero_aleatorio = proximo_aleatorio(numero_aleatorio)
```

No entanto, quando testamos a função, ela não aparenta ter o comportamento desejado:

```
julia> aleatorio()
ERROR: UndefVarError: numero_aleatorio not defined
```

O problema surge porque o Julia considera que sempre que aparece uma atribuição no corpo de uma função então o nome que está a ser atribuído é *local* à função, i.e., só é visível dentro da função e só existe a partir do momento em que a função está a ser invocada. Esta regra fazia sentido em todas as funções que tínhamos definido mas agora, em que necessitamos de alterar o nome a cada invocação da função `aleatorio`, a regra torna-se um obstáculo pois a mera presença da atribuição a `numero_aleatorio` faz com que esse nome seja local e sem qualquer relação com o nome global que definimos previamente. Para além disso, por ser local, precisa de uma atribuição que lhe dê o valor inicial mas como se pode ver no caso da função `aleatorio` esse valor inicial depende precisamente do próprio nome que queremos definir e que, por isso, ainda está indefinido. É isso que é explicitamente referido na mensagem de erro apresentada pelo Julia.

Para resolver este problema, o Julia disponibiliza uma *declaração* que permite indicar que um determinado nome que vai ser atribuído numa função corresponde à definição *global* feita previamente. É essa declaração que empregamos na seguinte definição:

```
aleatorio() =
    global numero_aleatorio = proximo_aleatorio(numero_aleatorio)
```

Com esta definição, já a função tem o comportamento pretendido.

```
julia> aleatorio()
2822
julia> aleatorio()
11031
julia> numero_aleatorio
11031
```

Como se vê, de cada vez que a função `aleatorio` é invocada, o valor associado a `numero_aleatorio` é actualizado, permitindo assim influenciar o futuro comportamento da função. Obviamente, em qualquer momento, podemos re-iniciar a sequência de números aleatórios simplesmente repondo o valor da semente:

```
julia> aleatorio()
21180
julia> aleatorio()
42629
julia> numero_aleatorio = 12345
julia> aleatorio()
2822
julia> aleatorio()
11031
julia> aleatorio()
21180
```

4.4 Escolhas Aleatórias

Se observarmos a definição da função `proximo_aleatorio` constatamos que a sua última operação é computar o resto da divisão por 65536, o que implica que a função produz sempre valores entre 0 e 65535. Embora (pseudo) aleatórios, estes valores estão contidos num intervalo que só muito excepcionalmente será útil. Na realidade, é muito mais frequente precisarmos de números aleatórios contidos em intervalos muito mais pequenos. Por exemplo, se pretendermos simular o lançar de uma moeda ao ar, estaremos interessados em ter apenas os números aleatórios 0 ou 1, representando "caras" ou "coroas."

Tal como os números aleatórios que produzimos são limitados ao intervalo $[0, 65536[$ pela obtenção do resto da divisão por 65536, também agora podemos voltar aplicar a mesma operação para produzir um intervalo mais pequeno. Assim, no caso do lançamento de uma moeda ao ar, podemos simplesmente usar a função `aleatorio` para gerar um número aleatório e, de seguida, aplicamos-lhe o resto da divisão por dois. No caso geral, em

que pretendemos números aleatórios no intervalo $[0, x[$, aplicamos o resto da divisão por x. Assim, podemos definir uma nova função que gera um número aleatório entre zero e o seu parâmetro e que, por tradição, iremos baptizar de `random`:

```
random(x) =
  aleatorio() % x
```

Note-se que a função `random` nunca deve receber um argumento maior que 65535 pois isso faria a função perder a característica da *equiprobabilidade* dos números gerados: todos os números superiores a 65535 terão probabilidade zero de ocorrerem.[3]

É agora possível simularmos inúmeros fenómenos aleatórios como, por exemplo, o lançar de uma moeda:

```
cara_ou_coroa() =
  if random(2) == 0
    "cara"
  else
    "coroa"
  end
```

Infelizmente, quando testamos repetidamente a nossa função, o seu comportamento parece muito pouco aleatório:

```
julia> cara_ou_coroa()
"cara"
julia> cara_ou_coroa()
"coroa"
julia> cara_ou_coroa()
"cara"
julia> cara_ou_coroa()
"coroa"
julia> cara_ou_coroa()
"cara"
julia> cara_ou_coroa()
"coroa"
```

Na realidade, os resultados que obtemos são uma repetição sistemática do par `cara`/`coroa`, o que mostra que a expressão `random(2)` se limita a gerar a sequência:

```
0101010101010101010101010101010101010101010101010101010101010101
```

O mesmo fenómeno ocorre para outros limites do intervalo de geração. Por exemplo, a expressão `random(4)` deveria gerar um número aleatório no conjunto $\{0, 1, 2, 3\}$ mas a sua aplicação repetida gera a seguinte sequência de números:

[3] Na verdade, o limite superior deve ser bastante inferior ao limite da função `aleatorio` para manter a equiprobabilidade dos resultados.

```
0123012301230123012301230123012301230123012301230123012301
```

Embora a equiprobabilidade dos números seja perfeita, é evidente que não há qualquer aleatoriedade.

O problema das duas sequências anteriores está no facto de terem um *período* muito pequeno. O período é o número de elementos que são gerados antes de o gerador entrar em ciclo e voltar a repetir os mesmos elementos que gerou anteriormente. Obviamente, quanto maior for o período, melhor é o gerador de números aleatórios e, neste sentido, o gerador que apresentámos é de baixa qualidade.

Enormes quantidades de esforço têm sido investidas na procura de bons geradores de números aleatórios e, embora os melhores sejam produzidos usando métodos bastante mais sofisticados do que os que empregámos até agora, também é possível encontrar um bom gerador congruencial linear desde que se faça uma judiciosa escolha dos seus parâmetros. De facto, o gerador congruencial linear

$$x_{i+1} = (ax_i + b) \bmod m$$

pode constituir um bom gerador pseudo-aleatório desde que tenhamos $a = 16807$, $b = 0$ e $m = 2^{31}-1 = 2147483647$. Uma tradução directa da definição matemática produz a seguinte função Julia:

```
proximo_aleatorio(ultimo_aleatorio) =
  16807*ultimo_aleatorio % 2147483647
```

Usando esta definição da função, a repetida avaliação da expressão `random(2)` produz a sequência:

```
01001010001011110100100011000111100110110101101111000011110
```

e, para a expressão `random(4)`, produz:

```
21312020300221331333233301112313001012333020321123122330222
```

É razoavelmente evidente que qualquer das sequências geradas tem agora um período demasiado grande para se conseguir detectar qualquer padrão de repetição pelo que, de ora em diante, será esta a definição de `proximo_aleatorio` que iremos empregar.

4.4.1 Números Aleatórios Fraccionários

O processo de geração de números aleatórios que implementámos apenas é capaz de gerar números aleatórios do tipo inteiro. No entanto, é também frequente precisarmos de gerar números aleatórios fraccionários, por exemplo, no intervalo $[0, 1[$.

Para combinar estes dois requisitos, é usual que a função `random` receba o limite superior de geração x e o analise para determinar se é inteiro ou real, devolvendo um valor aleatório adequado em cada caso. Para isso, não é preciso mais do que mapear o intervalo de geração de inteiros que, como vimos, é $[0, 2147483647[$, no intervalo $[0, x[$. A implementação assenta na possibilidade de, em Julia, se *especializar* uma função para diferentes tipos de argumentos:

```
random(x) =
  if x isa Int
    random_number()%x
  else
    x*random_number()/2147483647.0
  end
```

Dada a sua utilidade, a função `random` já se encontra pré-definida em Khepri.

4.4.2 Números Aleatórios num Intervalo

Se, ao invés de gerarmos números aleatórios no intervalo $[0, x[$, preferirmos gerar números aleatórios no intervalo $[x_0, x_1[$, então basta-nos gerar um número aleatório no intervalo $[0, x_1 - x_0[$ e somar-lhe x_0. A função `random_range` implementa esse comportamento:

```
random_range(x0, x1) =
  if x0 == x1
      x0
  else
      x0 + random(x1 - x0)
  end
```

Tal como a função `random`, também `random_range` produz um valor real no caso de algum dos limites ser real. Note-se que, à semelhança da função `random`, a função `random_range` já se encontra pré-definida no módulo Khepri.

Para visualizarmos um exemplo de utilização desta função, vamos recuperar a função `arvore` que modelava uma árvore, cuja definição era:

```
arvore(p, c, a, da0, f0, da1, f1) =
  let topo = p + vpol(c, a)
    ramo(p, topo)
    if c < 2
      folha(topo)
    else
      arvore(topo, c*f0, a + da0, da0, f0, da1, f1)
      arvore(topo, c*f1, a - da1, da0, f0, da1, f1)
    end
  end
```

Para incorporarmos alguma aleatoriedade neste modelo de árvore po-

Figura 4.2: Várias árvores geradas com ângulos de abertura aleatórios no intervalo $[\frac{\pi}{2}, \frac{\pi}{16}[$ e factores de redução do comprimento aleatórios no intervalo $[0.6, 0.9[$.

demos considerar que os ângulos de abertura dos ramos e os factores de redução de comprimento desses ramos não possuem valores constantes ao longo da recursão, antes variando dentro de certos limites. Assim, em vez de nos preocuparmos em ter diferentes aberturas e factores para o ramo esquerdo e direito, teremos simplesmente variações aleatórias em ambos:

```
arvore(p, c, a, min_a, max_a, min_f, max_f) =
  let topo = p + vpol(c, a)
    ramo(p, topo)
    if c < 2
      folha(topo)
    else
      arvore(topo,
             c*random_range(min_f, max_f),
             a + random_range(min_a, max_a),
             min_a, max_a, min_f, max_f)
      arvore(topo,
             c*random_range(min_f, max_f),
             a - random_range(min_a, max_a),
             min_a, max_a, min_f, max_f)
    end
  end
```

Usando esta nova versão, podemos gerar inúmeras árvores semelhantes e, contudo, suficientemente diferentes para parecerem bastante mais naturais. As árvores apresentadas na Figura 4.2 foram geradas usando exactamente os mesmos parâmetros de crescimento:

```
arvore(xy(0, 0), 20, pi/2, pi/16, pi/4, 0.6, 0.9)
arvore(xy(150, 0), 20, pi/2, pi/16, pi/4, 0.6, 0.9)
arvore(xy(300, 0), 20, pi/2, pi/16, pi/4, 0.6, 0.9)
arvore(xy(0, 150), 20, pi/2, pi/16, pi/4, 0.6, 0.9)
arvore(xy(150, 150), 20, pi/2, pi/16, pi/4, 0.6, 0.9)
arvore(xy(300, 150), 20, pi/2, pi/16, pi/4, 0.6, 0.9)
```

Exercício 4.4.1 As árvores produzidas pelas função `arvore` são pouco realista pois são totalmente bidimensionais, com troncos que são simples linhas e folhas que são pequenas circunferências. O cálculo das coordenadas dos ramos e folhas é também bidimensional, assentando sobre coordenadas polares que são dadas pelos parâmetros `comprimento` e `angulo`.

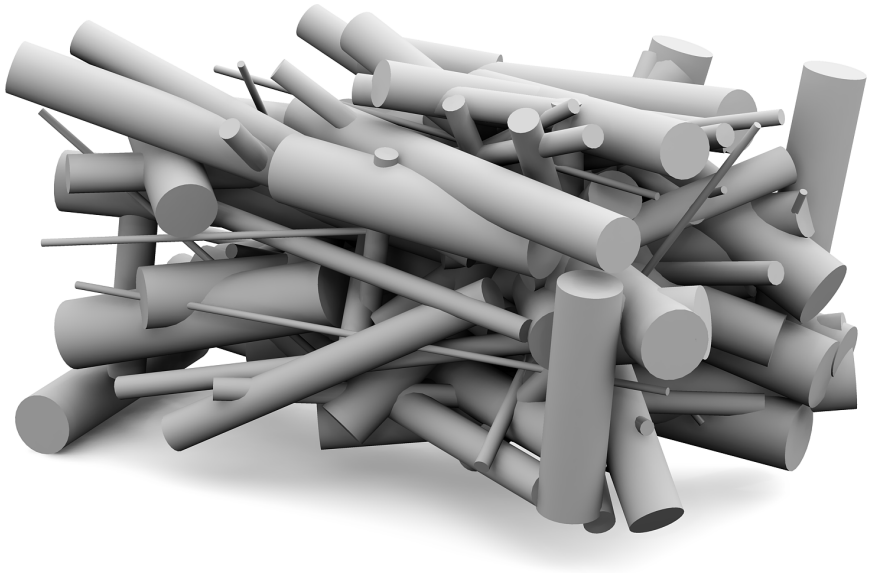
Pretende-se que redefina as funções `ramo`, `folha` e `arvore` de modo a aumentar o realismo das árvores geradas.

Para simplificar, considere que as folhas podem ser simuladas por pequenas esferas e que os ramos podem ser simulados por troncos de cone, cujo raio da base é 10% do comprimento do tronco e cujo raio do topo é 90% do raio da base.

Para que a geração de árvores passe a ser verdadeiramente tridimensional, redefina também a função `arvore` de modo a que o topo de cada ramo seja um ponto em coordenadas esféricas escolhidas aleatoriamente a partir da base do ramo. Isto implica que a função árvore, ao invés de ter dois parâmetros para o comprimento e o ângulo das coordenadas polares, precisará de ter três, para o comprimento, a longitude e a co-latitude. Do mesmo modo, ao invés de receber os quatro limites para a geração de comprimentos e ângulos aleatórios, precisará de seis limites para os três parâmetros.

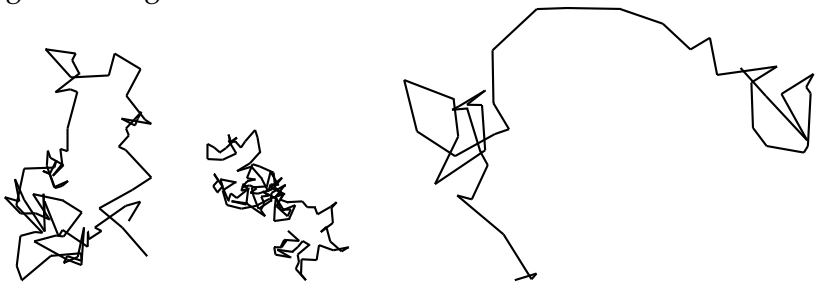
Experimente diferentes argumentos para a sua redefinição da função `arvore` de modo a gerar imagens como a seguinte:

Exercício 4.4.2 Defina uma função denominada `cilindros_aleatorios` que recebe como argumento um número de cilindros n e que gera n cilindros colocados em posições aleatórias, com raios aleatórios e comprimentos aleatórios, tal como se apresenta na seguinte imagem:



Exercício 4.4.3 Uma *caminhada aleatória* é uma formalização do movimento de uma particula que está sujeita a impulsos de intensidade aleatória vindos de direcções aleatórias. É este fenómeno que acontece, por exemplo, a um grão de pólen caído na água: à medida que as moléculas da água chocam com ele, o grão move-se aleatoriamente.

A seguinte imagem mostra três caminhadas aleatórias:

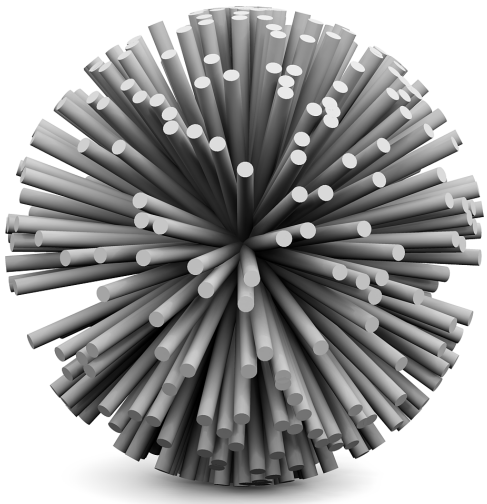


Considere uma modelação da *caminhada aleatória* num plano bi-dimensional. Defina a função `caminhada_aleatoria` que recebe o ponto inicial P da caminhada, a distância máxima d que a particula pode percorrer quando é impulsionada e o número n de impulsos sucessivos que a partícula vai receber. Note que, a cada impulso, a particula desloca-se, numa direcção aleatória, uma distância aleatória entre zero e a distância máxima. A sua função deverá simular a caminhada aleatória correspondente, tal como se apresenta na figura anterior que foi criada por três execuções diferen-

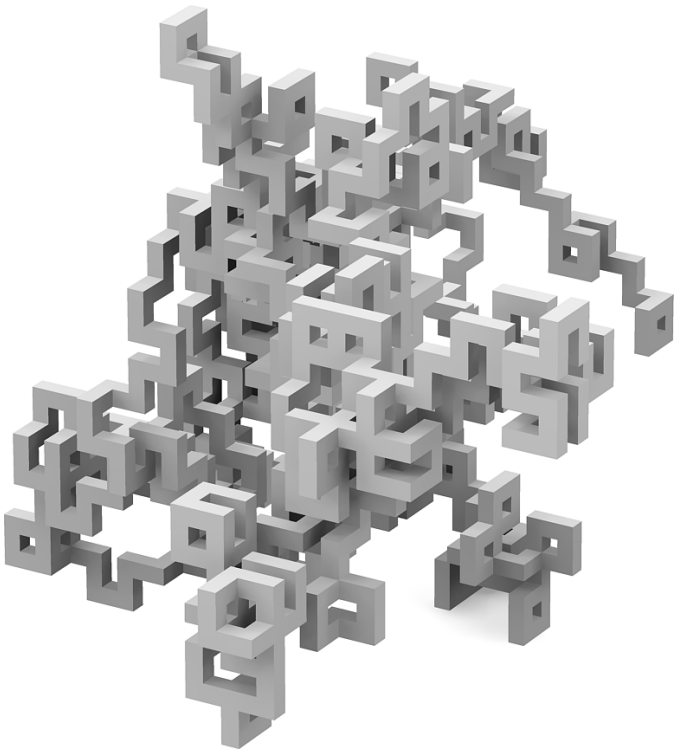
tes desta função. Da esquerda para a direita, foram usados os parâmetros $d = 5, n = 100$, $d = 2, n = 200$, e $d = 8, n = 50$,

Exercício 4.4.4 Defina uma função `cara_ou_coroa` *viciada* de tal forma que, embora a função produza aleatoriamente a *string* `"cara"` ou a *string* `"coroa"`, a "cara" sai, em média, apenas 30% das vezes em que a função é invocada.

Exercício 4.4.5 Defina a função `cilindros_esfera` que, dado um ponto P, um comprimento l, um raio r e um número n, cria n cilindros de comprimento l e raio r, cuja base está centrada no ponto P e cujo topo é posicionado aleatoriamente, tal como se apresenta na seguinte imagem:



Exercício 4.4.6 Defina uma função denominada `blocos_conexos` capaz de contruir um aglomerado de blocos conexos, i.e., onde os blocos estão fisicamente ligados entre si, tal como se vê na seguinte imagem:



Note que os blocos adjacentes possuem sempre orientações ortogonais.

4.5 Planeamento Urbano

As cidades constituem um bom exemplo de combinação entre estruturação e aleatoriedade. Embora muitas das cidades mais antigas aparentem ter uma forma caótica resultante do seu desenvolvimento não ter sido planeado, a verdade é que desde muito cedo se sentiu a necessidade de estruturar a cidade de modo a facilitar a sua utilização e o seu crescimento, sendo conhecidos exemplos de cidades planeadas desde 2600 antes de Cristo.

Embora haja bastantes variações, as duas formas mais usuais de planear uma cidade são através do plano ortogonal ou através do plano circular. No plano ortogonal, as avenidas são rectas e fazem ângulos rectos entre si. No plano circular as avenidas principais convergem numa praça central e as avenidas secundárias desenvolvem-se em círculos concêntricos em torno deste ponto, acompanhando o crescimento da cidade. A Figura 4.3 apresenta um bom exemplo de uma cidade essencialmente desenvolvida a partir de planos ortogonais. Nesta secção vamos explorar a aleatoriedade para produzir variações em torno destes planos.

Num plano ortogonal, a cidade organiza-se em termos de quarteirões, em que cada quarteirão assume uma forma quadrada ou rectangular e pode conter vários edifícios. Para simplificar, vamos assumir que cada

Figura 4.3: Vista aérea da cidade de New York nos Estados Unidos. Fotografia de Art Siegel.

quarteirão será de forma quadrada e irá conter um único edifício. Cada edifício terá uma largura determinada pela largura do quarteirão e uma altura máxima imposta. Os edifícios serão separados uns dos outros por ruas com uma largura fixa. Este modelo de cidade encontra-se representado na Figura 4.4.

Para estruturarmos a função que constrói a cidade em plano ortogonal, vamos decompor o processo na construção de sucessivas ruas. A construção de cada rua será então decomposta na construção dos sucessivos prédios. Assim, teremos de parametrizar a função com o número de ruas a construir na cidade e com o número de prédios por rua. Para além disso iremos necessitar de saber as coordenadas do ponto p onde se começa a construir a cidade, a largura e a altura de cada prédio, respectivamente, l e h e, finalmente, a largura da rua s. A função irá construir uma faixa de prédios seguida de uma rua e, recursivamente, construirá as restantes faixas de prédios e ruas no ponto correspondente à faixa seguinte. Para simplificar, vamos assumir que as ruas estarão alinhadas com os eixos X e Y, pelo que cada nova rua corresponde a um deslocamento ao longo do eixo Y e cada novo prédio corresponde a um deslocamento ao longo do eixo X. Assim, temos:

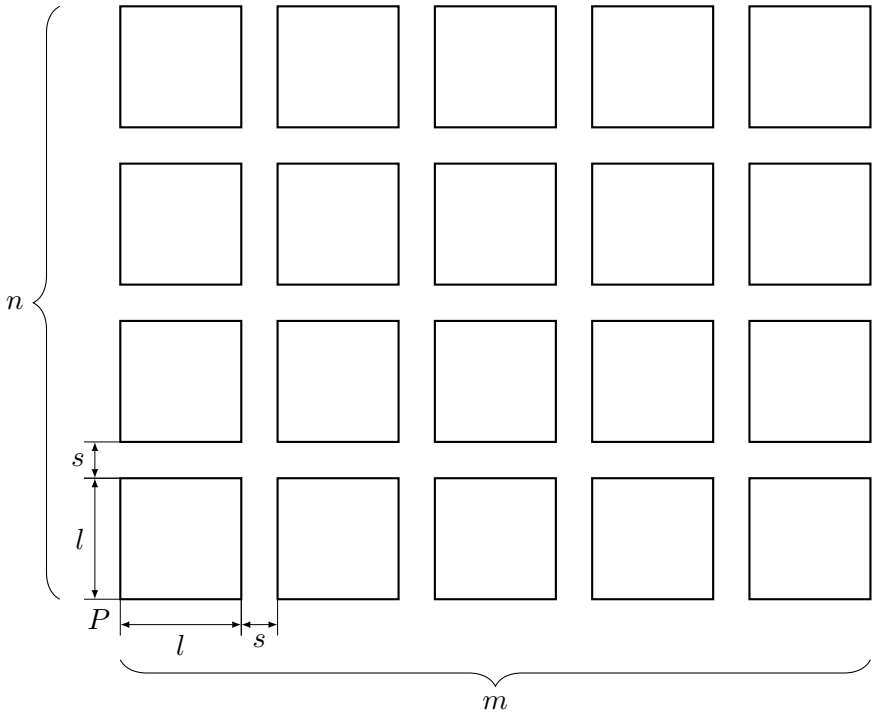


Figura 4.4: Modelo de uma cidade que segue um plano ortogonal.

```
malha_predios(p, n_ruas, m_predios, l, h, s) =
  if n_ruas == 0
    nothing
  else
    rua_predios(p, m_predios, l, h, s)
    malha_predios(p + vy(l + s), n_ruas - 1, m_predios, l, h, s)
  end
```

Para a construção das ruas com prédios, o raciocício é idêntico: colocamos um prédio nas coordenadas correctas e, recursivamente, colocamos os restantes prédios após o deslocamento correspondente. A seguinte função implementa este processo:

```
rua_predios(p, m_predios, l, h, s) =
  if m_predios == 0
    nothing
  else
    predio(p, l, h)
    rua_predios(p + vx(l + s), m_predios - 1, l, h, s)
  end
```

Finalmente, precisamos de definir a função que cria um prédio. Para simplificar, vamos modelá-lo como um paralelipípedo:

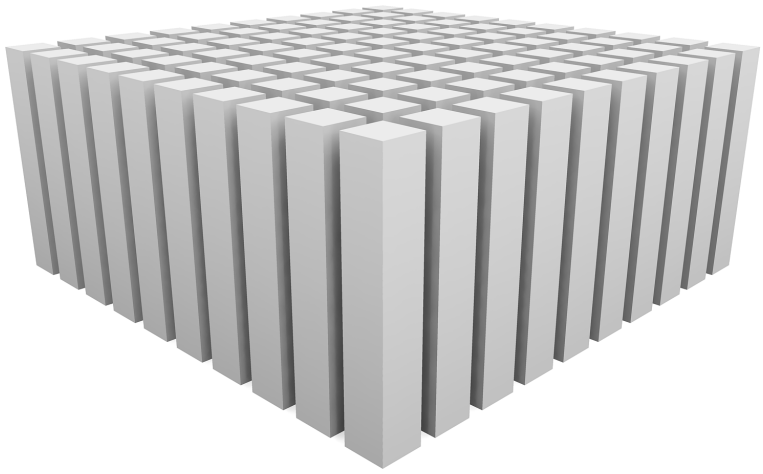


Figura 4.5: Uma urbanização em malha ortogonal com cem edifícios todos da mesma altura.

```
predio(p, l, h) =
  box(p, l, l, h)
```

Com estas três funções já podemos experimentar a construção de uma nova urbanização. Por exemplo, a seguinte expressão cria um arranjo de dez ruas com dez prédios por rua:

```
malha_predios(xyz(0, 0, 0), 10, 10, 100, 400, 40)
```

O resultado está representado na Figura 4.5.

Como é óbvio pela Figura 4.5, a urbanização produzida é pouco elegante pois falta-lhe alguma da aleatoriedade que caracteriza as cidades.

Para incorporarmos essa aleatoriedade vamos começar por considerar que a altura dos prédios pode variar aleatoriamente entre a altura máxima dada e um décimo dessa altura. Para isso, redefinimos a função `predio`:

```
predio(p, l, h) =
  box(p, l, l, random_range(0.1, 1.0)*h)
```

Usando exactamente os mesmos parâmetros anteriores em duas avaliações sucessivas da mesma expressão, conseguimos agora construir as urbanizações esteticamente mais apelativas representadas nas Figuras 4.6 e 4.7.

Exercício 4.5.1 As urbanizações produzidas pelas funções anteriores não apresentam variabilidade suficiente, pois os edifícios têm todos a mesma forma. Para melhorar a qualidade estética da urbanização pretende-se empregar diferentes funções para a construção de diferentes tipos de edifícios: a função `predio0` deverá construir um paralelipípedo de altura aleatória

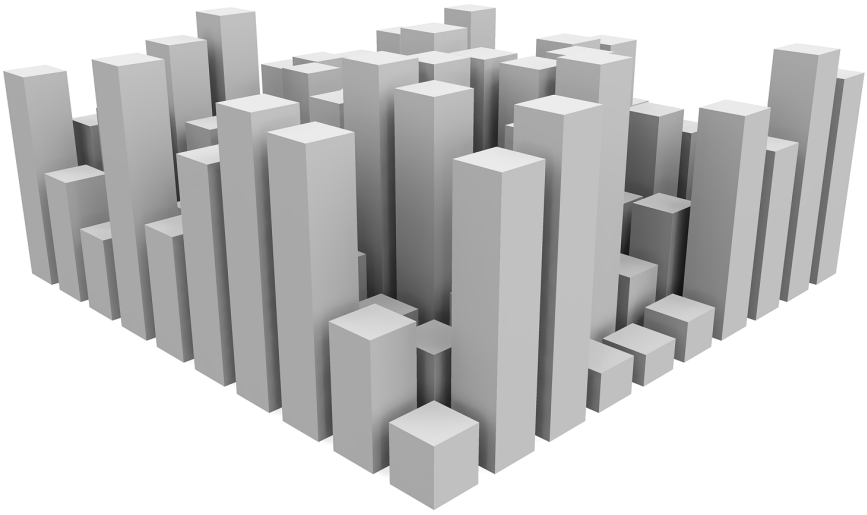


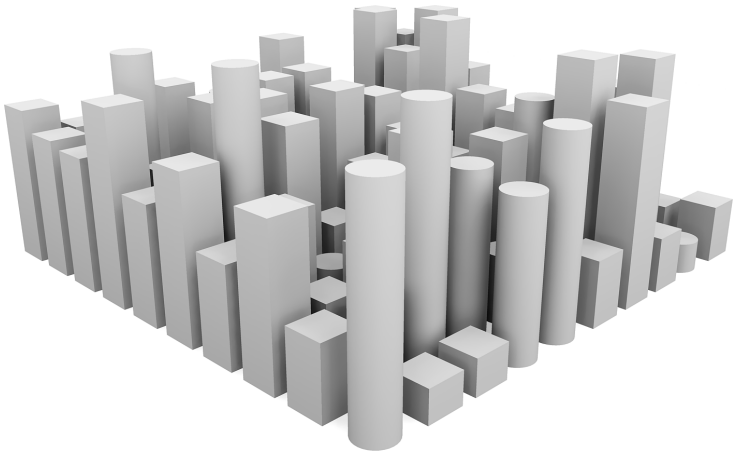
Figura 4.6: Uma urbanização em malha ortogonal com cem edifícios com alturas aleatórias.

Figura 4.7: Uma urbanização em malha ortogonal com cem edifícios com alturas aleatórias.

Figura 4.8: Vista de alguns prédios de Manhattan. Fotografia de James K. Poole.

tal como anteriormente e a função `predio1` deverá construir uma torre cilíndrica de altura aleatória e contida no espaço de um quarteirão. Defina estas duas funções.

Exercício 4.5.2 Pretende-se que use as duas funções `predio0` e `predio1` definidas no exercício anterior para a redefinição da função `predio` de modo a que esta, aleatoriamente, construa prédios diferentes. A urbanização resultante deverá ser constituída aproximadamente por 20% de torres circulares e 80% de prédios rectangulares, tal como é exemplificado na imagem seguinte:

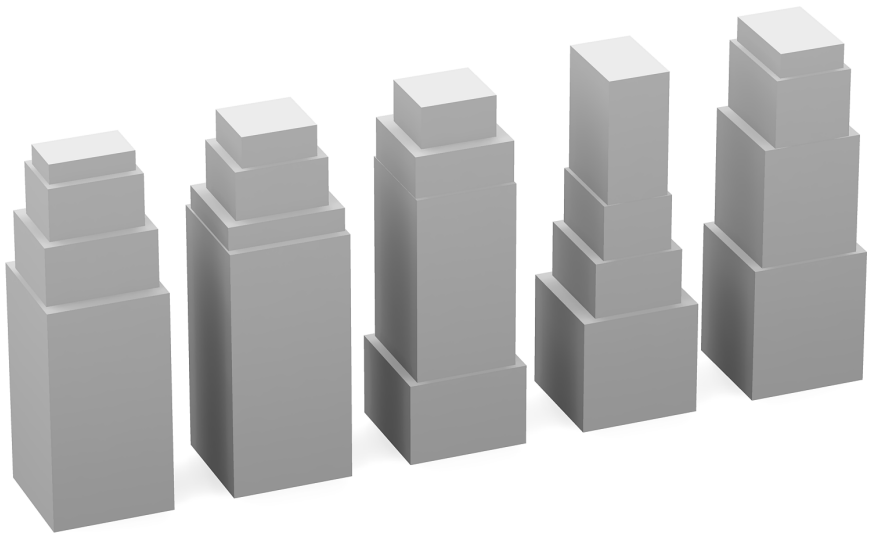


Exercício 4.5.3 As cidades criadas nos exercícios anteriores apenas permitem dois tipos de edifícios: torres paralelipipédicas ou torres cilíndricas.

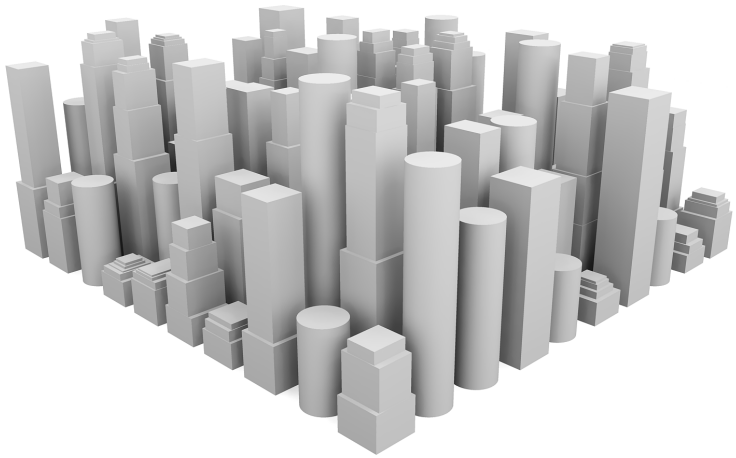
Contudo, quando observamos uma cidade real como a apresentada na Figura 4.8, verificamos que existem prédios com muitas outras formas pelo que, para aumentarmos o realismo das nossas simulações, teremos de implementar um vasto número de funções, cada uma capaz de construir um edifício diferente.

Felizmente, uma observação atenta da Figura 4.8 mostra que, na verdade, muitos dos prédios seguem um padrão e podem ser modelado por paralelipípedos sobrepostos com dimensões aleatórias mas sempre sucessivamente mais pequenos em função da altura, algo que podemos facilmente implementar com uma única função.

Considere que este modelo de edifício é parametrizado pelo número de "blocos" sobrepostos pretendidos, pelas coordenadas do primeiro bloco e pelo comprimento, largura e altura do edifício. O bloco de base tem exactamente o comprimento e largura especificados mas a sua altura deverá estar entre 20% e 80% da altura total do edifício. Os blocos seguintes estão centrados sobre o bloco imediatamente abaixo e possuem um comprimento e largura que estão entre 70% e 100% dos parâmetros correspondentes no bloco imediatamente abaixo. A altura destes blocos deverá ser entre 20% e 80% da altura restante do edifício. A imagem seguinte mostra alguns exemplos deste modelo de edifícios:



Com base nesta especificação, defina a função `predio_blocos` e use-a para redefinir a função `predio0` de modo a que sejam gerados prédios com um número aleatório de blocos sobrepostos. Com esta redefinição, a função `malha_predios` deverá ser capaz de gerar urbanizações semelhantes à imagem seguinte, em que se empregou para cada prédio um número de blocos entre 1 e 6:

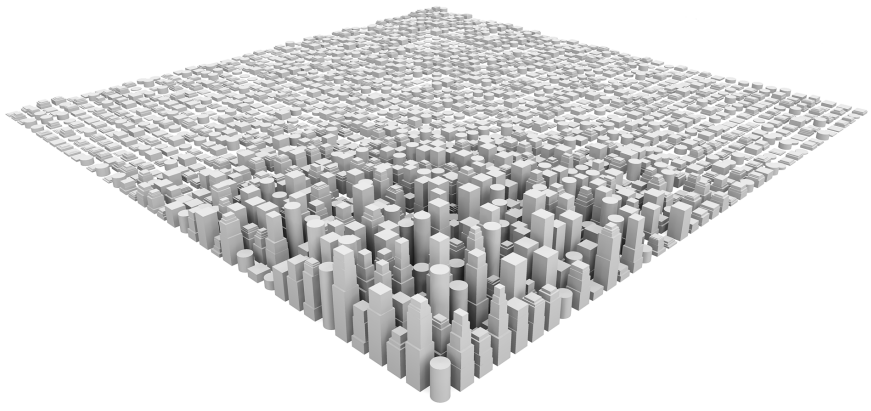


Exercício 4.5.4 Em geral, as cidades possuem um núcleo de edifícios relativamente altos no denominado *centro financeiro* (abreviado CBD, de *central business district*) e, à medida que nos afastamos, a altura tende a diminuir, sendo este fenómeno perfeitamente visível na Figura 4.3.

A variação da altura dos edifícios pode ser modelada por diversas funções matemáticas mas, neste exercício, pretende-se que empregue uma distribuição Gaussiana bi-dimensional dada por

$$f(x,y) = e^{-\left((\frac{x-x_o}{\sigma_x})^2+(\frac{y-y_o}{\sigma_y})^2\right)}$$

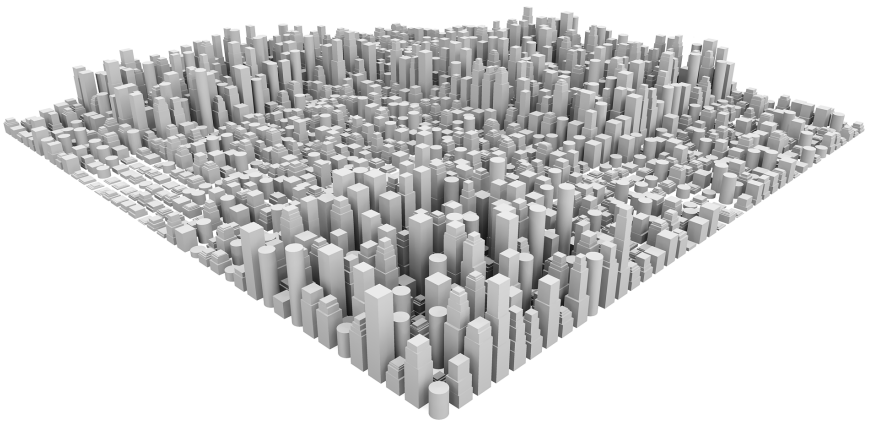
em que f é o factor de ponderação da altura, (x_0, y_0) são as coordenadas do ponto mais alto da superfície Gaussiana e σ_0 e σ_1 são os factores que afectam o alargamento bi-dimensional dessa superfície. Para simplificar, assuma que o centro financeiro fica nas coordenadas $(0, 0)$ e que $\sigma_x = \sigma_y = 25l$, sendo l a largura do edifício. A imagem seguinte mostra o aspecto de uma destas urbanizações:



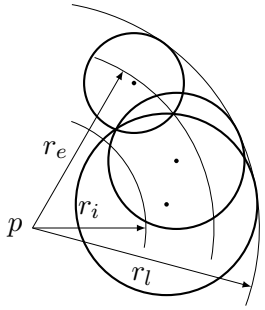
Incorpore esta distribuição no processo de construção de cidades para produzir cidades mais realistas.

Exercício 4.5.5 Frequentemente, as cidades possuem mais do que um núcleo de edifícios altos. Estes núcleos encontram-se separados uns dos outros por zonas de edifícios menos altos. Tal como no exercício anterior, cada núcleo de edifícios pode ser modelado por uma distribuição Gaussiana. Admitindo a independência dos vários núcleos, a altura dos edifícios pode ser modelada por distribuições Gaussianas sobrepostas, i.e., em cada ponto, a altura do edifício será o máximo das alturas das distribuições Gaussianas dos vários núcleos.

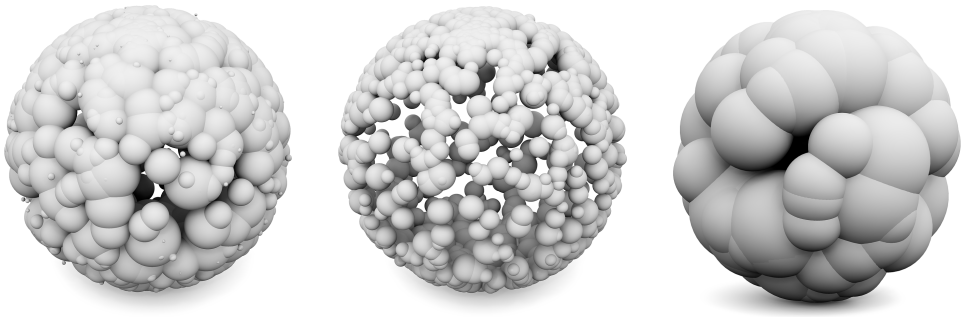
Utilize a abordagem anterior para exemplificar a modelação de uma cidade com três núcleos de "arranha-céus", à semelhança do que se apresenta na imagem seguinte:



Exercício 4.5.6 Pretende-se criar um conjunto de n esferas tangentes a uma esfera virtual de centro em p e raio limite r_l. As esferas possuem centros a uma distância de p que é um valor aleatório compreendido entre um raio interior r_i e um raio exterior r_e, tal como se pode visualizar no seguinte esquema:



Defina uma função denominada `esferas_na_esfera` que, a partir do centro p, dos raios r_i, r_e, e r_l e ainda do número n, cria este conjunto de esferas, permitindo produzir imagens como as que se seguem:



Exercício 4.5.7 Um dos aspectos mais fascinantes da aleatoriedade é a sua capacidade para fazer emergir regularidades, no mínimo, estranhas. Por exemplo, dadas três posições arbitrárias P_0, P_1, e P_2, escolhemos aleatoriamente uma delas como posição inicial Q_0 e calculamos a posição intermédia Q_1 entre Q_0 e uma das posições P_i escolhida aleatoriamente. O cálculo da posição intermédia é então repetido para a posição Q_1, gerando a sequência $Q_0, Q_1, Q_2, \ldots$ de posições posicionada aleatoriamente dentro do triângulo formado pelas posições P_i. Qual é a imagem que resulta da "nuvem" de posições gerada por essa sequência?

Capítulo 5

Estruturas

5.1 Introdução

A grande maioria das funções que definimos nas secções anteriores visa a produção de formas geométricas concretas. Nesta secção, iremos abordar funções cujo objectivo é a produção de formas geométricas abstractas, no sentido de serem formas geométricas representadas apenas por um aglomerado de posições no espaço. Por exemplo, uma linha poligonal pode ser representada apenas pela sequência de posições por onde a linha passa. Essa sequência de posições pode, no entanto, ser usada para vários outros fins como, por exemplo, para criar uma sequência de esferas com os centros nessas posições ou para definir a trajectória de um tubo que passa por essas posições. Estes exemplos mostram que a manipulação destes aglomerados de posições é independentemente do uso subsequente que lhes pretendemos dar.

Para podermos tratar aglomerados de posições como um todo é necessário agrupar essas posições no que se designa por *estruturas de dados*. Estas estruturas são arranjos particulares de dados que permitem tratá-los como um todo. Uma lista telefónica, por exemplo, pode ser vista como uma estrutura de dados que organiza associações entre nomes e números de telefone.

Uma das estruturas de dados mais flexível que mais iremos utilizar existe em muitas linguagens de programação e denomina-se *lista*.

5.2 Listas

Em termos genéricos, uma lista é apenas uma sequência de elementos. Diferentes linguagens de programação implementam listas de diferentes maneiras. Em Julia, as listas são um caso particular de uma estrutura de dados mais sofisticada denominada *array*. Por uma questão de simplicidade, vamos discutir apenas este caso particular, que vamos continuar a denominar por *lista*.

Julia, podemos construir uma lista de elementos "embrulhando" os elementos entre parêntesis rectos. Podemos aceder a cada um destes elementos utilizando o operador de *indexação*, baseado em justapor à lista o índice do elemento a que se pretende aceder, "embrulhado" em parêntesis rectos. Eis um exemplo do uso destas operações:

```
julia> amigos = ["Filipa", "Pedro", "Carlos", "Maria"]
julia> amigos[1]
"Filipa"
julia> amigos[4]
"Maria"
julia> amigos[end]
"Maria"
```

Como podemos ver pela interacção anterior, a expressão `amigos[i]` retorna o elemento da lista na posição `i`. O índice `end` representa a última posição da lista. Para além disso, a indexação pode também retornar listas de elementos contíguos (denominadas *slices*) de uma lista através da indicação de um índice inicial e um índice final (ambos inclusivos). Eis alguns exemplos:

```
julia> amigos[2:3]
["Pedro", "Carlos"]
julia> amigos[2:end]
["Pedro", "Carlos", "Maria"]
julia> amigos[1:3]
["Filipa", "Pedro", "Carlos"]
julia> amigos[2:end-1]
["Pedro", "Carlos"]
julia> amigos[2:end-2]
["Pedro"]
julia> amigos[2:end-3]
[]
```

Para além da possibilidade de criar listas e de aceder aos seus elementos, existem ainda muitas outras operações para a manipulação de listas. Comecemos por introduzir a operação de *concatenação* que nos permite criar uma lista maior a partir de duas listas mais pequenas:

```
julia> conhecidos = ["Bernardo", "Paula"]
julia> novos_amigos = [conhecidos..., amigos...]
julia> novos_amigos
["Bernardo", "Paula", "Filipa", "Pedro", "Carlos", "Maria"]
julia> amigos
["Filipa", "Pedro", "Carlos", "Maria"]
```

Como podemos ver, ao concatenarmos listas, produzimos uma nova lista com todos os elementos das listas concatenadas e pela mesma ordem. Notemos também que a lista original fica inalterada pela concatenação.

É importante referir que as listas podem conter um número arbitrário de elementos, incluindo zero. A lista com zero elementos, também designada lista *vazia*, escreve-se `[]`.

5.3 Operações com Listas

A possibilidade de criar listas, produzir sublistas e concatenar listas abre a porta para um conjunto muito vasto de operações adicionais que podemos definir. Por exemplo, podemos definir uma função que nos diz quantos elementos existem numa lista. Para isso, podemos pensar recursivamente:

- Se a lista é vazia, então o número de elementos é, obviamente, zero.
- Caso contrário, o número de elementos será um mais o número de elementos da lista sem o primeiro elemento.

Para nos assegurarmos da correcção do nosso raciocínio temos de garantir que se verificam os pressupostos a que todas as definições recursivas têm de obedecer, nomeadamente:

- Que há uma redução da complexidade do problema a cada invocação recursiva: de facto, a cada invocação recursiva a lista usada é mais pequena.
- Que há um caso mais simples de todos onde existe uma resposta imediata: quando a lista é vazia, a resposta é zero.
- Que existe uma equivalência entre o problema original e o uso do resultado da recursão: é verdade que o número de elementos de uma lista é o mesmo que somar 1 ao número de elementos dessa lista sem o primeiro elemento.

A verificação destes pressupostos é suficiente para garantirmos que a função está correcta.

Traduzindo este raciocínio para Julia, obtemos:

```
numero_elementos(lista) =
  if lista == []
    0
  else
    1 + numero_elementos(lista[2:end])
  end
```

Experimentando, temos:

```
julia> numero_elementos([1, 2, 3, 4])
4
julia> numero_elementos([])
0
```

É importante percebermos que a presença de sublistas não altera o número de elementos de uma lista. De facto, os elementos das sublistas *não* são considerados elementos da lista. Por exemplo:

```
julia> numero_elementos([[1, 2], [3, 4, 5, 6]])
2
```

Como se vê no exemplo anterior, a lista apenas contém dois elementos, embora cada um deles seja uma lista com mais elementos. Na verdade, a função `numero_elementos` já existe em Julia com o nome `length` e encontra-se implementada de forma bastante mais eficiente.

Exercício 5.3.1 Escreva uma função que dada uma lista de elementos devolve um elemento dessa lista, escolhido aleatoriamente.

Exercício 5.3.2 Defina a função `um_de_cada` que, dada uma lista de listas, constrói uma lista contendo, por ordem, um elemento aleatório de cada uma das listas. Por exemplo:

```
julia> um_de_cada([[0, 1, 2], [3, 4], [5, 6, 7, 8]])
[2 4 7]
julia> um_de_cada([[0, 1, 2], [3, 4], [5, 6, 7, 8]])
[1 3 5]
julia> um_de_cada([[0, 1, 2], [3, 4], [5, 6, 7, 8]])
[1 4 8]
```

Exercício 5.3.3 Defina a função `elementos_aleatorios` que, dado um número n e uma lista de elementos devolve n elementos dessa lista escolhidos aleatoriamente. Por exemplo:

```
julia> elementos_aleatorios(3, [0, 1, 2, 3, 4, 5, 6])
[1, 5, 2]
```

Exercício 5.3.4 Defina uma função `inverte` que recebe uma lista e devolve outra lista que possui os mesmos elementos da primeira só que por ordem inversa.

5.3.1 Enumerações

Vamos agora considerar uma função capaz de criar enumerações. Dados os limites a e b de um intervalo $[a, b]$ e um incremento i, a função deverá devolver uma lista com todos os números a, $a+i$, $a+2i, a+3i, a+4i, \ldots, a+ni$ que não são superiores a b.

Mais uma vez, a recursão ajuda: a enumeração dos números no intervalo $[a, b]$ com um incremento i é exactamente o mesmo que o número a seguido da enumeração dos números no intervalo $[a+i, b]$. O caso mais simples de todos é uma enumeração num intervalo $[a, b]$ em que $a > b$. Neste caso o resultado é simplesmente uma lista vazia. Esta análise permite-nos definir a função:

```
enumera(a, b, i) =
  if a > b
    []
  else
    [a, enumera(a+i, b, i)...]
  end
```

Como exemplo, temos:

```
julia> enumera(1, 5, 1)
[1, 2, 3, 4, 5]
julia> enumera(1, 5, 2)
[1, 3, 5]
```

Para tornarmos a função ainda mais genérica, podemos tratar também o caso em que o incremento d é negativo, caso em que a função deverá fazer uma enumeração regressiva.

Note-se que a definição actual da função `enumera` não permite esse comportamento pois a utilização de um incremento negativo provoca recursão infinita. O problema está no facto do teste de paragem ser feito com a função > que só é apropriada para o caso em que o incremento é positivo. No caso de incremento negativo deveriamos usar <. Assim, para resolvermos o problema temos de identificar primeiro qual a enumeração correcta a fazer:

```
enumera(a, b, i) =
  begin
    enumera_crescente(a) =
      if a > b
        []
      else
        [a, enumera_crescente(a+i)...]
      end
    enumera_decrescente(a) =
      if a < b
        []
      else
        [a, enumera_decrescente(a+i)...]
      end
    if i > 0
      enumera_crescente(a)
    else
      enumera_decrescente(a)
    end
  end
```

Agora já podemos ter:

```
julia> enumera(1, 5, 1)
[1, 2, 3, 4, 5]
julia> enumera(5, 1, -1)
[5, 4, 3, 2, 1]
julia> enumera(6, 0, -2)
[6, 4, 2, 0]
```

Exercício 5.3.5 A funcão ι (pronuncia-se "iota") representa uma enumeração desde 0 até um limite superior n *exclusive*, com os elementos da enumeração separados por um dado incremento, i.e.:

```
julia> iota(10, 1)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
julia> iota(10, 2)
[0, 2, 4, 6, 8]
```

Defina a função `iota` à custa da função `enumera`.

Vimos que a função `enumera` permite a criação de progressões aritméticas e, na verdade, a linguagem Julia disponibiliza sintaxe específica precisamente para este propósito. A expressão `a:b` representa o mesmo que `enumera(a, b, 1)`, enquanto que a expressão `a:i:b` representa o mesmo que `enumera(a, b, i)`.

Exercício 5.3.6 Escreva uma função `pertence` que recebe um número e uma lista e verifica se aquele número existe na lista. Eis dois exemplos do uso da função:

```
julia> pertence(3, [5, 2, 1, 3, 4, 6])
true
julia> pertence(7, [5, 2, 1, 3, 4, 6])
false
```

Exercício 5.3.7 Escreva uma função `elimina1` que recebe, como argumentos, um número e uma lista e devolve, como resultado, outra lista onde a primeira ocorrência desse número foi eliminada. Eis um exemplo:

```
julia> elimina1(3, [1, 2, 3, 4, 3, 2, 1])
[1, 2, 4, 3, 2, 1]
```

Exercício 5.3.8 Escreva uma função `elimina` que recebe, como argumentos, um número e uma lista e devolve, como resultado, outra lista onde esse número não aparece. Eis um exemplo:

```
julia> elimina(3, [1, 2, 3, 4, 3, 2, 1])
[1, 2, 4, 2, 1]
```

Exercício 5.3.9 Escreva uma função `substitui` que recebe dois números e uma lista como argumentos e devolve outra lista com todas as ocorrências do segundo argumento substituídas pelo primeiro. Como exemplo, considere:

```
julia> substitui(0, 1, [1, 2, 1, 3])
[0, 2, 0, 3]
julia> substitui(0, 1, [2, 3, 4, 5])
[2, 3, 4, 5]
```

Exercício 5.3.10 Escreva uma função `remove_duplicados` que recebe uma lista de números como argumento e devolve outra lista com todos os elementos da primeira mas sem duplicados, i.e.:

```
julia> remove_duplicados([1, 2, 3, 3, 2, 4, 5, 4, 1])
[3, 2, 5, 4, 1]
```

Exercício 5.3.11 Escreva uma função `ocorrencias` que recebe um número e uma lista como argumentos e devolve o número de ocorrências daquele número na lista. Por exemplo:

```
julia> ocorrencias(4, [1, 2, 3, 3, 2, 4, 5, 4, 1])
2
```

Exercício 5.3.12 Escreva uma função `posicao` que recebe um número e uma lista como argumentos e devolve a posição da primeira ocorrência daquele número na lista. Note que as posições na lista começam em zero. Por exemplo:

```
julia> posicao(4, [1, 2, 3, 3, 2, 4, 5, 4, 1])
5
```

5.4 Polígonos

As listas são particularmente úteis para a representação de entidades geométricas abstractas, i.e., entidades para as quais apenas nos interessa saber algumas propriedades como, por exemplo, a sua posição. Para isso, podemos usar uma lista de posições, i.e., uma lista cujos elementos são o resultado de invocações dos construtores de coordenadas.

Imaginemos, a título de exemplo, que pretendemos representar um *polígono*. Por definição, um polígono é uma figura plana limitada por um caminho fechado composto por uma sequência de segmentos de recta. Cada segmento de recta é uma *aresta* (ou *lado*) do polígono. Cada ponto onde se encontram dois segmentos de recta é um *vértice* do polígono.

A partir da definição de polígono é fácil vermos que uma das formas mais simples de representar um polígono será através da sequência de posições que indica qual a ordem pela qual devemos unir os vértices com uma aresta e onde se admite que o último elemento será unido ao primeiro. É precisamente essa sequência de argumentos que fornecemos no seguinte exemplo, onde usamos a função pré-definida `polygon`:

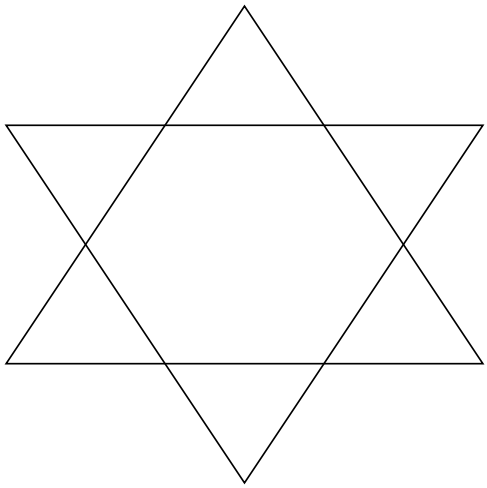


Figura 5.1: Dois polígonos sobrepostos.

```
polygon(xy(-2, -1), xy(0, 2), xy(2, -1))
polygon(xy(-2, 1), xy(0, -2), xy(2, 1))
```

Imaginemos agora que, ao invés de indicarmos explicitamente as posições dos vértices, pretendemos que seja uma função a computá-las. Naturalmente, isso obrigaria a função a computar uma sequência de posições. Ora é precisamente aqui que as listas vêm dar uma ajuda preciosa: elas permitem facilmente representar sequências e, para simplificar ainda mais a sua utilização, muitas das funções geométricas, como `line` e `polygon`, aceitam também listas como argumento. Na verdade, a Figura 5.1 pode ser também obtida pelas seguintes expressões:

```
polygon([xy(-2, -1), xy(0, 2), xy(2, -1)])
polygon([xy(-2, 1), xy(0, -2), xy(2, 1)])
```

Assim, uma lista com as posições de quatro vértices pode representar um quadrilátero, um octógono será representado por uma lista de oito vértices, um hentriacontágono será representado por uma lista de trinta e um vértices, etc. Agora, apenas é necessário concentrarmo-nos na criação destas listas de posições.

5.4.1 Estrelas Regulares

O polígono representado na Figura 5.1 foi gerado "manualmente" pela sobreposição de dois triângulos. Um polígono mais interessante é o famoso *pentagrama*, ou estrela de cinco pontas, que tem sido usado com conotações simbólicas, mágicas ou decorativas desde os tempos da Babilónia.[1] A

[1] O pentagrama foi símbolo de perfeição matemática para os Pitagóricos, foi símbolo do domínio do espírito sobre os quatro elementos da matéria pelos ocultistas, represen-

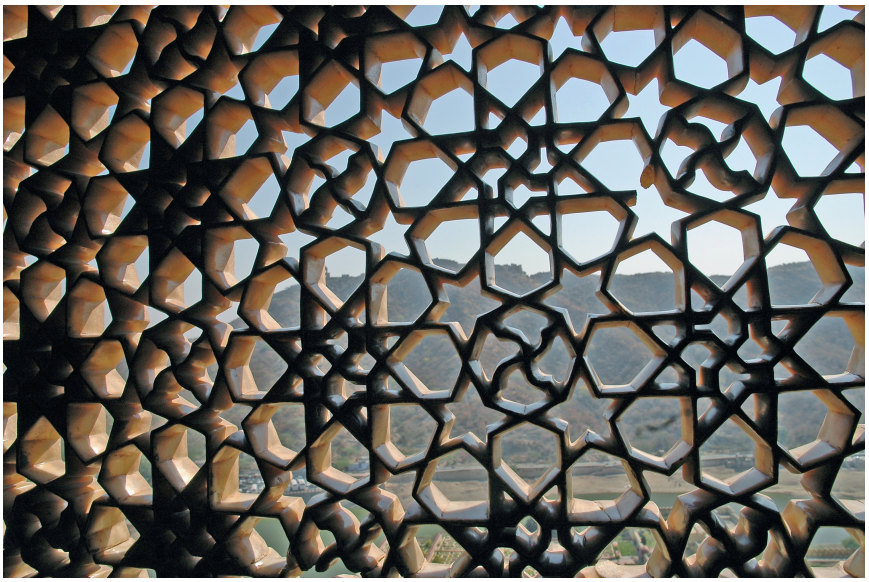


Figura 5.2: Variações de Estrelas numa janela do Forte de Amber, localizado no estado de Jaipur, na Índia. Fotografia de David Emmett Cooley.

Figura 5.2 demonstra o uso do pentagrama (bem como do octograma, do octógono e do hexágono) como elemento decorativo e estrutural numa janela de pedra.

À parte as conotações extra-geométricas, o pentagrama é, antes de tudo, um polígono. Por este motivo, é desenhável pela função `polygon` desde que consigamos produzir uma lista com as posições dos vértices. Para isso, vamos reportar-nos à Figura 5.3 onde é visível que os cinco vértices do pentagrama dividem o círculo em 5 partes, com arcos de $\frac{2\pi}{5}$ cada. Dado o centro do pentagrama, o seu vértice superior faz um ângulo de $\frac{\pi}{2}$ com o eixo das abcissas. Esse vértice deve ser unido, não com o vértice seguinte, mas sim com o imediatamente a seguir a esse, i.e., após uma rotacão de dois arcos ou $\frac{2 \cdot 2\pi}{5} = \frac{4}{5}\pi$. Estamos agora em condições de definir a função `vertices_pentagrama` que constrói a lista com os vértices do pentagrama:

tou as cinco chagas de Cristo crucificado para os Cristãos, foi associado às proporções do corpo humano, foi usado como símbolo maçónico e, quando invertido, até foi associado ao satanismo.

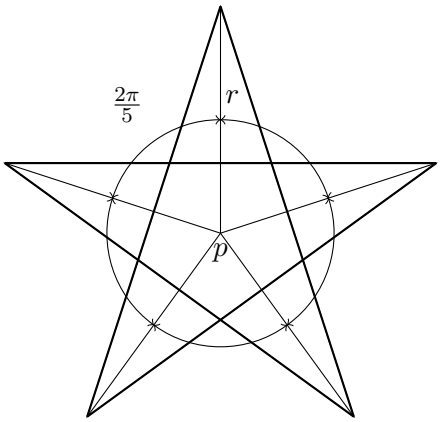


Figura 5.3: Construção do Pentagrama.

```
vertices_pentagrama(p, raio) =
  [p+vpol(raio, pi/2+0*4/5*pi),
   p+vpol(raio, pi/2+1*4/5*pi),
   p+vpol(raio, pi/2+2*4/5*pi),
   p+vpol(raio, pi/2+3*4/5*pi),
   p+vpol(raio, pi/2+4*4/5*pi)]

polygon(vertices_pentagrama(xy(0, 0), 1))
```

Como é óbvio, a função `vertices_pentagrama` é demasiado repetitiva, pelo que seria preferível encontrarmos uma forma mais estruturada de gerarmos aqueles vértices. Para isso, vamos começar por pensar na generalização da função.

O caso geral de um pentagrama é a *estrela regular*, em que se faz variar o número de vértices e o número de arcos que separam um vértice do vértice a que ele se une. O pentagrama é um caso particular da estrela regular em que o número de vértices é cinco e o número de arcos de separação é dois. Matematicamente falando, uma estrela regular representa-se pelo símbolo de Schläfli[2] $\{\frac{v}{a}\}$ em que v é o número de vértices e a é o número de arcos de separação. Nesta notação, um pentagrama escreve-se como $\{\frac{5}{2}\}$.

Para desenharmos estrelas regulares vamos idealizar uma função que, dado o centro da estrela, o raio do círculo circunscrito, o número de vértices v (a que chamaremos `n_vertices`) e o número de arcos de separação a (a que chamaremos `n_arcos`), calcula o tamanho do arco $\Delta\phi$ que é preciso avançar a partir de cada vértice. Esse arco é, obviamente, $\Delta\phi = a\frac{2\pi}{v}$. Tal como no pentagrama, para primeiro vértice vamos considerar um ângulo inicial de $\phi = \frac{\pi}{2}$. A partir desse primeiro vértice não é preciso mais do que ir aumentando o ângulo ϕ de $\Delta\phi$ de cada vez. As seguintes funções

[2] Ludwig Schläfli foi um matemático e geómetra Suíço que fez importantes contribuições, em particular, na geometria multidimensional.

implementam este raciocínio:

```
vertices_estrela(p, raio, n_vertices, n_arcos) =
  pontos_circulo(p, raio, pi/2, n_arcos*2*pi/n_vertices, n_vertices)

pontos_circulo(p, raio, fi, dfi, n) =
  if n == 0
    []
  else
    [p+vpol(raio, fi), pontos_circulo(p, raio, fi+dfi, dfi, n-1)...]
  end
```

Quando o objectivo é gerar uma lista de n entidades mas em que há uma fórmula para produzir cada uma dessas entidades, o Julia disponibiliza uma sintaxe própria que simplifica essa computação. Esta sintaxe é fortemente inspirada na sintaxe empregue em Matemática para descrever conjuntos em compreensão. Usando como exemplo a função `vertices_pentagrama`, a sua definição pode ser simplificada para:

```
vertices_pentagrama(p, raio) =
  [p+vpol(raio, pi/2+i*4/5*pi)
   for i in [0, 1, 2, 3, 4]]
```

Notemos, na definição anterior, que a lista de posições é produzida a partir de um cálculo envolvendo a variável `i`, em que `i` assume sucessivamente cada um dos valores da lista `[0, 1, 2, 3, 4]`. Por sua vez, esta última lista pode produzida por uma enumeração, ficando a definição com a forma:

```
vertices_pentagrama(p, raio) =
  [p + vpol(raio, pi/2 + i*4/5*pi)
   for i in 0:4]
```

A partir daqui, podemos generalizar a função, introduzindo parâmetros no lugar das constantes:

```
pontos_circulo(p, raio, fi, dfi, n) =
  [p + vpol(raio, fi + i*dfi)
   for i in 0:n-1]
```

Esta definição da função `pontos_circulo` é, naturalmente, equivalente à definição recursiva que tínhamos anteriormente. No entanto, é substancialmente mais simples, evitando por completo a necessidade de usar recursão. Convém ter presente, no entanto, que a utilização desta forma de produção de lista *em compreensão* não consegue resolver todos os casos de funções recursivas que produzem listas, mas apenas aqueles que têm uma estrutura suficientemente regular.

Voltando à função `vertices_estrela` é agora fácil gerar os vértices de qualquer estrela regular. A Figura 5.4 apresenta as estrelas regulares desenhadas a partir das seguintes expressões:

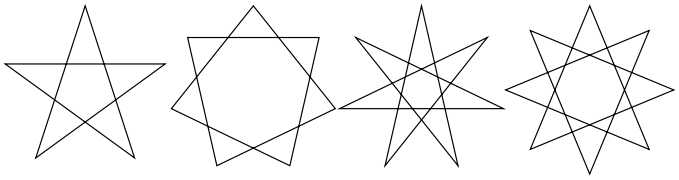


Figura 5.4: Estrelas regulares. Da esquerda para a direita, temos um pentagrama ($\{\frac{5}{2}\}$), dois heptagramas ($\{\frac{7}{2}\}$ e $\{\frac{7}{3}\}$) e um octograma ($\{\frac{8}{3}\}$).

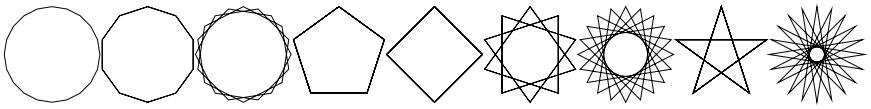


Figura 5.5: Estrelas regulares $\{\frac{p}{r}\}$ com $p = 20$ e r a variar desde 1 (à esquerda) até 9 (à direita).

```
polygon(vertices_estrela(xy(0, 0), 1, 5, 2))

polygon(vertices_estrela(xy(2, 0), 1, 7, 2))

polygon(vertices_estrela(xy(4, 0), 1, 7, 3))

polygon(vertices_estrela(xy(6, 0), 1, 8, 3))
```

A Figura 5.5 mostra uma sequência de estrelas regulares $\{\frac{20}{r}\}$ com r a variar desde 1 até 9.

É muito importante percebermos que a construção de estrelas está separada em duas partes distintas. De um lado, o da função `vertices_estrela`, produzimos as coordenadas dos vértices das estrelas pela ordem em que estes devem ser unidos. No outro lado, o da função `polygon`, usamos essas coordenadas para criar uma representação gráfica dessa estrela baseada em linhas que ligam os seus vértices. A passagem das coordenadas de um lado para o outro é realizada através de uma lista que é "produzida" num lado e "consumida" no outro.

Esta utilização de listas para separar diferentes processos é fundamental e será por nós repetidamente explorada para simplificar os nossos programas.

5.4.2 Polígonos Regulares

Um *polígono regular* é um polígono que tem todos os lados de igual comprimento e todos os ângulos de igual amplitude. A Figura 5.6 ilustra exemplos de polígonos regulares. Como é óbvio, um *polígono regular* é um caso particular de uma estrela regular em que o número de arcos de separação é um.

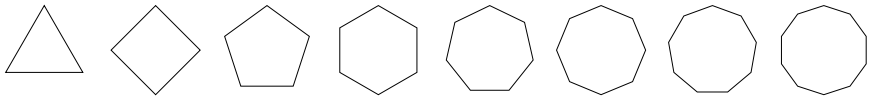


Figura 5.6: Polígonos regulares. Da esquerda para a direita temos um triângulo equilátero, um quadrado, um pentágono regular, um hexágono regular, um heptágono regular, um octógono regular, um eneágono regular e um decágono regular.

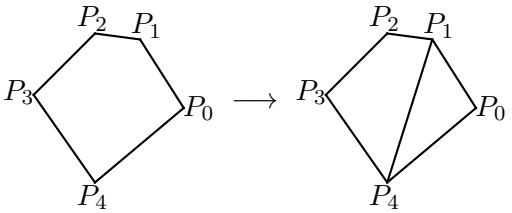
Para criarmos polígonos regulares, vamos definir uma função denominada `vertices_poligono_regular` que gera uma lista de coordenadas correspondente às posições dos vértices de um polígono regular de n lados, inscrito numa circunferência de raio r centrada em p, cujo "primeiro" vértice faz um ângulo ϕ com o eixo X. Sendo um polígono regular um caso particular de uma estrela regular, basta-nos invocar a função que computa os vértices de uma estrela regular mas usando, para o parâmetro Δ_ϕ, apenas a divisão da circunferência 2π pelo número de lados n, i.e.:

```
vertices_poligono_regular(p, r, fi, n) =
  pontos_circulo(p, r, fi, 2*pi/n, n)
```

Finalmente, podemos encapsular a geração dos vértices com o seu uso para criar o poligono correspondente:

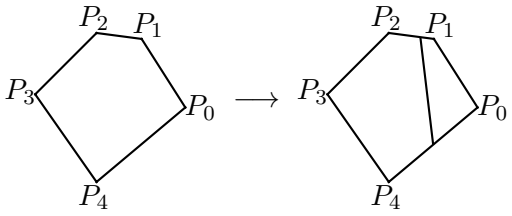
```
poligono_regular(p, r, fi, n) =
  polygon(vertices_poligono_regular(p, r, fi, n))
```

Exercício 5.4.1 Considere um polígono representado pela lista dos seus vértices `(`p_0 p_1 `...` p_n`)`, tal como apresentamos no seguinte esquema, à esquerda:



Pretende-se dividir o polígono em dois sub-polígonos, tal como apresentado no esquema anterior, à direita. Defina a função `divide_poligono` que, dada a lista de vértices do polígono e dados dois índices i e j onde se deve fazer a divisão (com $i < j$), calcula as listas de vértices de cada um dos sub-polígonos resultantes e devolve-as numa lista.

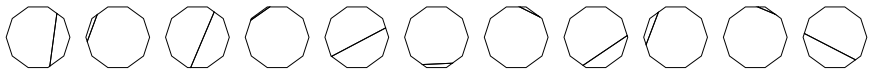
Exercício 5.4.2 Podemos considerar uma generalização do exercício anterior baseada na *bissecção* do polígono usando uma linha arbitrária, tal como podemos ver no esquema seguinte:



Para simplificar o processo, considere que cada extremidade da linha de bissecção está localizada a uma determinada fracção $f_i \in [0, 1]$ da distância que vai do vértice P_i ao vértice sequinte P_{i+1} (com $P_{n+1} = P_0$). Por exemplo, no esquema anterior, os vértices em questão são P_1 e P_4 e a fracções são, respectivamente, de $f_1 = 0.3$ e $f_4 = 0.5$.

Defina a função `bisseccao_poligono` que, dada a lista de vértices do polígono e dados os dois índices i e j imediatamente anteriores aos extremos da linha de bissecção e dadas as fracções f_i e f_j da distância, respectivamente, entre os vértices (P_i, P_{i+1}) e entre (P_j, P_{j+1}), calcula as listas de vértices de cada um dos sub-polígonos resultantes e devolve-as numa lista (i.e., devolve uma lista de listas de vértices).

Exercício 5.4.3 Defina a função `bisseccao_aleatoria_poligono` que, dados os vértices de um polígono, devolve uma lista de listas de vértices dos sub-polígonos correspondentes a uma divisão aleatória do polígono. A seguinte imagem mostra exemplos de divisões aleatórias de um octógono.



Exercício 5.4.4 Usando a função `bisseccao_aleatoria_poligono` definida no exercício anterior, defina a função `divisao_aleatoria_poligono` que, de forma aleatória, divide e subdivide recursivamente um poligono até que se atinja um determinado nível de recursão. A seguinte imagem mostra a divisão aleatória de um decágono para sucessívos níveis de recursão desde 0 até 10.

5.5 Linhas Poligonais e *Splines*

Vimos que as listas permitem armazenar um número variável de elementos e vimos como é possível usar listas para separar os programas que definem os vértices das figuras geométricas daqueles que usam esses vértices para as representarem graficamente.

No caso da função `polygon` discutida na secção 5.4, as listas de vértices são usadas para a criação de polígonos, i.e., figuras planas limitadas por

um caminho fechado composto por uma sequência de segmentos de recta. Acontece que nem sempre queremos que as nossas figuras sejam limitadas por caminhos fechados, ou que esses caminhos sejam uma sequência de segmentos de recta ou, sequer, que as figuras sejam planas. Para resolver este problema, necessitamos de usar funções capazes de, a partir de listas de posições, criarem outros tipos de figuras geométricas.

O caso mais simples é o de uma linha poligonal não necessariamente planar que, se for aberta, pode ser criada pela função `line` e, se for fechada, pela função `closed_line`. Em qualquer destes casos, as funções recebem um número variável de argumentos correspondentes aos vértices da linha poligonal ou, alternativamente, uma lista com esses vértices.

No caso de não pretendermos linhas poligonais mas sim curvas "suaves" que passem por uma sequência de posições, podemos empregar a função `spline` para curvas abertas e a função `closed_spline` para curvas fechadas. Tal como as funções `line` e `closed_line`, também estas podem receber um número variável de argumentos ou uma lista com esses argumentos.

A diferença entre uma linha poligonal e uma *spline* é visível na Figura 5.7, onde comparamos uma sequência de pontos unida com uma linha poligonal com a mesma sequência unida com uma *spline*. A imagem foi produzida pela avaliação das seguintes expressões:

```
pontos = [xy(0, 2), xy(1, 4),
          xy(2, 0), xy(3, 3),
          xy(4, 0), xy(5, 4),
          xy(6, 0), xy(7, 2),
          xy(8, 1), xy(9, 4)]

line(pontos)
spline(pontos)
```

Naturalmente, a especificação "manual" das coordenadas dos pontos é pouco conveniente, sendo preferível que essas coordenadas sejam computadas automaticamente de acordo com uma especificação matemática da curva pretendida. Imaginemos, por exemplo, que pretendemos traçar uma curva sinusóide a partir de um ponto P. Infelizmente, da lista de figuras geométricas disponibilizadas pelo Khepri—pontos, linhas rectas, rectângulos, polígonos, círculos, arcos de círculo, etc.—não consta a sinusóide.

Para resolvermos este problema, podemos criar uma aproximação a uma sinusóide. Para isso, podemos calcular uma sequência de pontos pertencentes à curva da sinusóide e traçar rectas ou, ainda melhor, curvas que passem por esses pontos. Para calcular o conjunto de valores da função seno no intervalo $[x_0, x_1]$ basta-nos considerar um incremento Δ_x e, começando no ponto x_0 e passando do ponto x_i para o ponto x_{i+1} através de $x_{i+1} = x_i + \Delta_x$, vamos sucessivamente calculando o valor da expressão $\sin(x_i)$ até que x_i exceda x_1. Para obtermos uma definição recur-

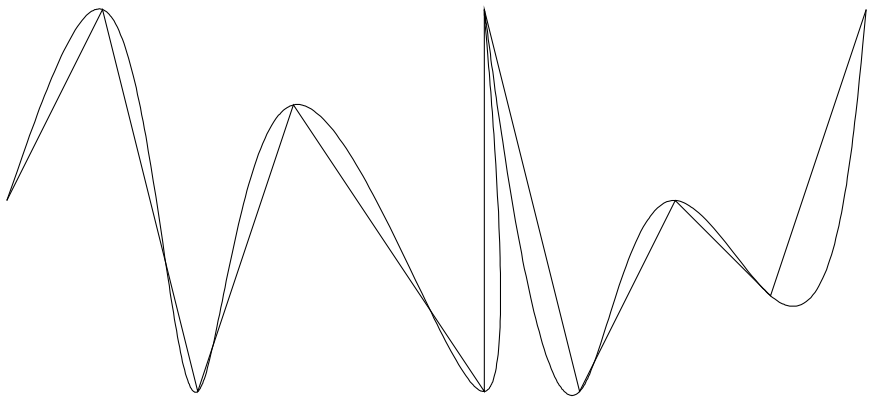


Figura 5.7: Comparação entre uma linha poligonal e uma *spline* que unem o mesmo conjunto de pontos.

siva para este problema podemos pensar que quando $x_0 > x_1$ o resultado será uma lista vazia de coordenadas, caso contrário juntamos a coordenada $(x_0, \sin(x_0))$ à lista de coordenadas do seno para o intervalo $[x_0 + \Delta_x, x_1]$. Esta definição é traduzida directamente para a seguinte função:

```
pontos_seno(x0, x1, dx) =
  if x0 > x1
    []
  else
    [xy(x0, sin(x0)), pontos_seno(x0+dx, x1, dx)...]
  end
```

Para termos uma maior liberdade de posicionamento da curva sinusóise no espaço, podemos modificar a função anterior para incorporar um ponto P em relação ao qual se calcula a curva:

```
pontos_seno(p, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p+vy(sin(x0)), pontos_seno(p+vx(dx), x0+dx, x1, dx)...]
  end
```

A Figura 5.8 mostra as curvas traçadas pelas seguintes expressões que unem os pontos por intermédio de linhas poligonais:

```
line(pontos_seno(xy(0.0, 1.0), 0.0, 6.0, 1.0))
line(pontos_seno(xy(0.0, 0.5), 0.0, 6.5, 0.5))
line(pontos_seno(xy(0.0, 0.0), 0.0, 6.4, 0.2))
```

Note-se, na Figura 5.8, que demos um deslocamento vertical às curvas para melhor se perceber a diferença de precisão entre elas. Como é plenamente evidente, quantos mais pontos se usarem para calcular a curva, mais

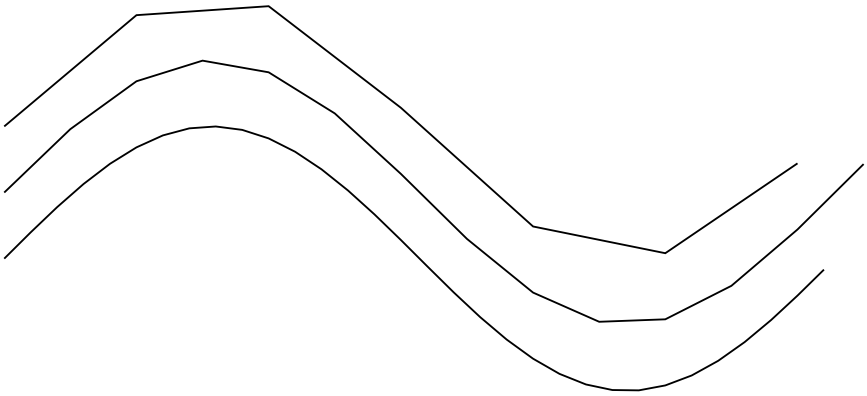


Figura 5.8: Senos desenhados usando linhas poligonais com um número crescente de pontos.

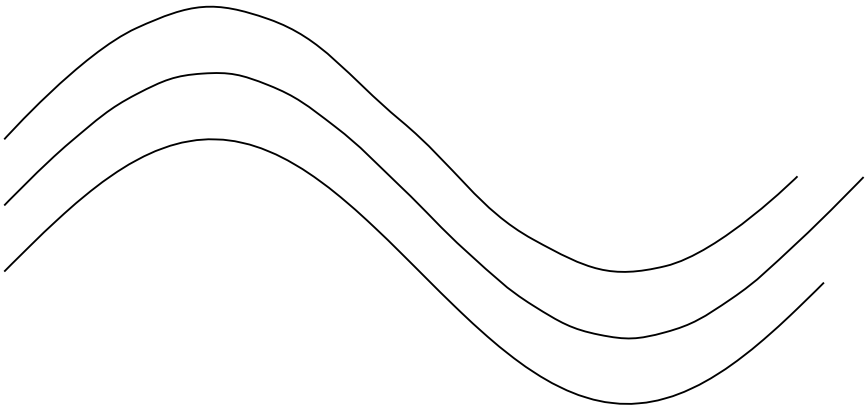
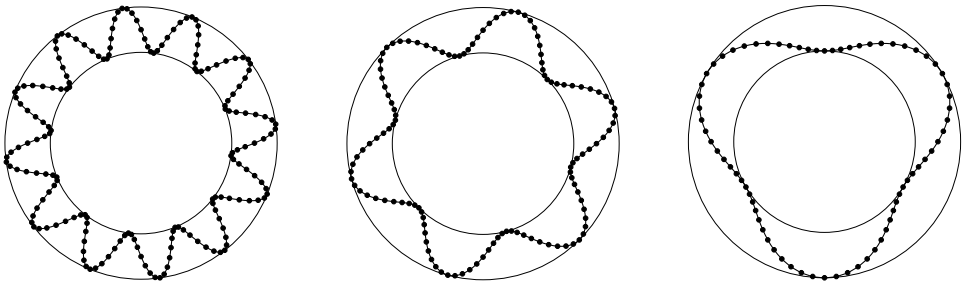


Figura 5.9: Senos desenhados usando *splines* com um número crescente de pontos.

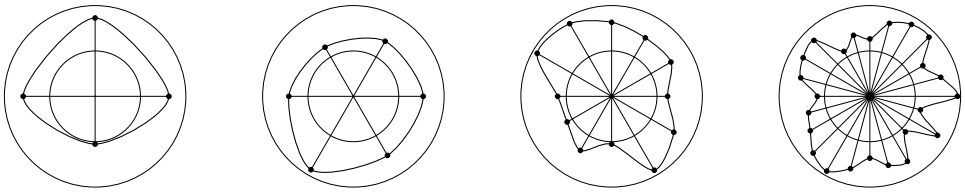
próxima será a linha poligonal da verdadeira curva. No entanto, há também que ter em conta que o aumento do número de pontos obriga o Khepri (e a ferramenta de CAD) a maior esforço computacional.

Para se obterem ainda melhores aproximações, embora à custa de ainda maior esforço computacional, podemos usar *splines*, simplesmente mudando as expressões anteriores para usarem a função `spline` no lugar de `line`. O resultado está visível na Figura 5.9.

Exercício 5.5.1 Defina a função `pontos_sinusoide_circular` de parâmetros p, r_i, r_e, c e n que computa n pontos de uma curva fechada com a forma de uma sinusoide com c ciclos que se desenvolve num anel circular centrado no ponto p e delimitado pelos raios interior r_i e exterior r_e, tal como se pode ver nos vários exemplos apresentados na seguinte figura onde, da esquerda para a direita, o número de ciclos c é 12, 6, e 3.



Exercício 5.5.2 Defina a função `pontos_circulo_raio_aleatorio` de parâmetros p, r_0, r_1 e n que computa n pontos de uma curva fechada de forma aleatória contida num anel circular centrado no ponto p e delimitado pelos raios interior r_i e exterior r_e, tal como se pode ver nos vários exemplos apresentados na seguinte figura onde, da esquerda para a direita, fomos aumentando progressivamente o número de pontos usados, assim aumentando a irregularidade da curva.



Sugestão: para a computação dos pontos, considere a utilização de coordenadas polares para distribuir uniformemente os pontos em torno de um círculo mas com a distância ao centro desse círculo a variar aleatoriamente entre r_i e r_e. A título de exemplo, considere que a curva mais à esquerda na figura anterior foi gerada pela expressão

```
closed_spline(pontos_circulo_raio_aleatorio(xy(0, 0), 1.0, 2.0, 6))
```

Solomon "Sol" LeWitt foi um dos grandes impulsionadores da Arte Conceptual, um movimento que considera que a ideia artística é, por si só, uma obra de arte e, portanto, independente da sua concretização. À semelhança da arquitectura, em que a concretização da ideia arquitetónica é relegada para as áreas da engenharia e construção, LeWitt acreditava que a concretização da ideia artística podia ser realizada por outros, desde que as instruções para o fazer fossem claras. Assim, empenhou-se em detalhar essas instruções para muitas das suas criações.

5.6 Treliças

Uma treliça é uma estrutura composta por barras rígidas que se unem em nós, formando unidades triangulares. Sendo o triângulo o único polígono

Figura 5.10: A esfera geodésica de Buckminster Füller. Fotografia de Glen Fraser.

intrinsecamente estável, a utilização de triângulos convenientemente interligados permite que as treliças sejam estruturas indeformáveis. Apesar da simplicidade dos elementos triangulares, diferentes arranjos destes elementos permitem diferentes tipos de treliças.

É conhecido o uso de treliças desde a Grécia antiga, em que eram utilizadas para o suporte dos telhados. No século dezasseis, nos seus *Quatro Livros de Arquitectura*, Andrea Palladio ilustra pontes de treliças. No século dezanove, com o uso extensivo de metal e a necessidade de ultrapassar vãos cada vez maiores, inventaram-se vários tipos de treliças que se distinguem pelos diferentes arranjos de barras verticais, horizontais e diagonais e que, frequentemente, se denominam de acordo com os seus inventores. Temos assim treliças Pratt, treliças Howe, treliças Town, treliças Warren, etc. Nas últimas décadas as treliças começaram a ser intensivamente utilizadas como elemento artístico ou para a construção de superfícies elaboradas. No conjunto de exemplos mais famosos incluem-se a esfera geodésica de Buckminster Füller para a Exposição Universal de 1967, apresentada (reconstruída) na Figura 5.10 e as treliças em forma de banana de Nicolas Grimshaw para o terminal de Waterloo, visíveis na Figura 5.11.

As treliças apresentam um conjunto de propriedades que as tornam particularmente interessantes do ponto de vista arquitectónico:

- É possível construir treliças muito grandes a partir de elementos relativamente pequenos, facilitando a produção, transporte e erecção.
- Desde que as cargas sejam aplicadas apenas nos nós da treliça, as

Figura 5.11: Treliças em forma de banana para o terminal de Waterloo, por Nicolas Grimshaw. Fotografia de Thomas Hayes.

barras ficam apenas sujeitas a forças axiais, i.e., trabalham apenas à tracção ou à compressão, permitindo formas estruturais de grande eficiência.

- Uma vez que os elementos básicos de construção são barras e nós, é fácil adaptar as dimensões destes às cargas previstas, permitindo assim grande flexibilidade.

5.6.1 Desenho de Treliças

O passo fundamental para o desenho de treliças é a construção dos elementos triangulares fundamentais. Embora frequentemente se considerem apenas treliças bi-dimensionais (também chamadas *treliças planas*), iremos tratar o caso geral de uma treliça tri-dimensional composta por semi-octaedros. Esta forma de treliça denomina-se de *space frame*. Cada semi-octaedro é denominado *módulo*.

A Figura 5.12 apresenta o esquema de uma treliça. Embora nesta figura os nós da treliça estejam igualmente espaçados ao longo de rectas paralelas, nada obriga a que assim seja. A Figura 5.13 mostra uma outra treliça em que tal não se verifica.

Assim, para o desenho de uma treliça, vamos considerar, como base de trabalho, três sequências arbitrárias de pontos em que cada ponto define um nó da treliça. A partir destas três sequências podemos criar as ligações necessárias entre cada par de nós. A Figura 5.15 apresenta o esquema de

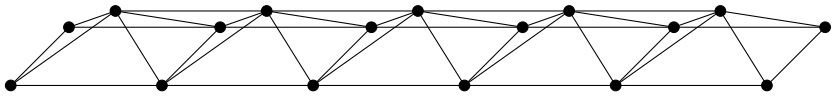


Figura 5.12: Treliça composta por elementos triangulares iguais.

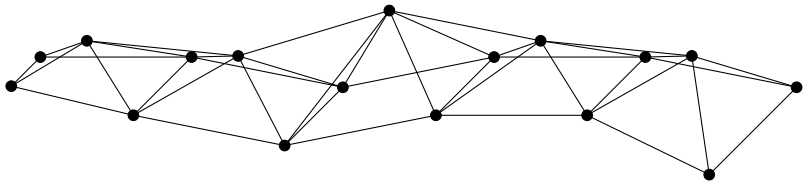


Figura 5.13: Treliça cujos elementos triangulares não são idênticos entre si.

ligação a partir de três sequências de pontos (a_0, a_1, a_2), (b_0, b_1) e (c_0, c_1, c_2). Note-se que a sequência de topo estabelecida pelos pontos b_i da sequência intermédia tem sempre menos um elemento que as sequências a_i e c_i.

Para a construção da treliça precisamos de encontrar um processo que, a partir das listas de pontos a_i, b_i e c_i, não só crie os nós correspondentes aos vários pontos, como os interligue da forma correcta. Comecemos por tratar da criação dos nós:

```
nos_trelica(ps) =
  if ps == []
    nothing
  else
    no_trelica(ps[1])
    nos_trelica(ps[2:end])
  end
```

A função `no_trelica` (notemos o singular, por oposição ao plural empregue na função `nos_trelica`) recebe uma posição e é responsável por criar o modelo tridimensional que representa o nó da treliça centrado nessa posição. Uma hipótese simples será esta função criar uma esfera onde encaixarão as barras, mas, por agora, vamos deixar a decisão sobre qual o modelo em concreto para mais tarde e vamos simplesmente admitir que a função `no_trelica` fará algo apropriado para criar o nó. Nestes casos em que o que se pretende é simplesmente realizar uma operação sobre todos

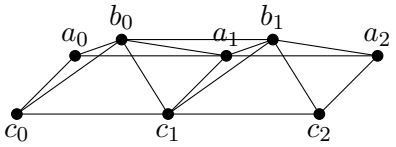


Figura 5.14: Esquema de ligação de barras de uma treliça em *space frame*.

os elementos de uma lista, é possível utilizar uma sintaxe especial do Julia, tal como ilustramos em seguida:

```
nos_trelica(ps) =
  for p in ps
    no_trelica(p)
  end
```

Notemos que a leitura do programa sugere precisamente que, para cada valor `p` na lista `ps`, invocamos a função `no_trelica`.

Tendo a função `nos_trelica`, podemos começar a idealizar a função que constrói a treliça completa a partir das listas de posições `ais`, `bis` e `cis`:

```
trelica(ais, bis, cis) =
  begin
    nos_trelica(ais)
    nos_trelica(bis)
    nos_trelica(cis)
    ...
  end
```

De seguida, vamos tratar de estabelecer as barras entre os nós. Da análise da Figura 5.15 ficamos a saber que temos uma ligação entre cada posição a_i e cada c_i, outra entre a_i e b_i, outra entre c_i e b_i, outra entre b_i e a_{i+1}, outra entre b_i e c_{i+1}, outra entre a_i e a_{i+1}, outra entre b_i e b_{i+1} e, finalmente, outra entre c_i e c_{i+1}. Admitindo que a função `barra_trelica` cria o modelo tridimensional dessa barra (por exemplo, um cilindro, ou uma barra prismática), podemos começar por definir uma função denominada `barras_trelica` (notemos o plural) que, dadas duas listas de pontos `ps` e `qs`, cria barras de ligação ao longo dos sucessivos pares de pontos. Para criar uma barra, a função necessita de um elemento dos `ps` e outro dos `qs`, o que implica que a função deve terminar assim que uma destas listas estiver vazia. A definição fica então:

```
barras_trelica(ps, qs) =
  if ps == [] || qs == []
    nothing
  else
    barra_trelica(ps[1], qs[1])
    barras_trelica(ps[2:end], qs[2:end])
  end
```

À semelhança do que aconteceu com a função `no_trelica` é também possível simplificar a função `barras_trelica` percorrendo, em simultâneo, os elementos das duas listas `ps` e `qs`:

```
barras_trelica(ps, qs) =
  for (p, q) in zip(ps, qs)
    barra_trelica(p, q)
  end
```

Uma vez que é necessário percorrer duas listas, o operador `for` necessita de duas variáveis, neste caso, `p` e `q`, variáveis essas que irão ser sucessivamente atribuídas com os elementos correspondentes das duas listas, sendo a função `zip` quem assegura essa correspondência.[3]

Para interligar cada nó a_i ao correspondente nó c_i, apenas temos de avaliar `barras_trelica(ais, cis)`. O mesmo poderemos dizer para interligar cada nó b_i ao nó a_i correspondente e para interligar cada b_i a cada c_i. Assim, temos:

```
trelica(ais, bis, cis) =
  begin
    nos_trelica(ais)
    nos_trelica(bis)
    nos_trelica(cis)
    barras_trelica(ais, cis)
    barras_trelica(bis, ais)
    barras_trelica(bis, cis)
    ...
  end
```

Para ligar os nós b_i aos nós a_{i+1} podemos simplesmente subtrair o primeiro nó da lista `ais` e estabelecer a ligação como anteriormente. O mesmo podemos fazer para ligar cada b_i a cada c_{i+1}. Finalmente, para ligar cada a_i a cada a_{i+1} podemos usar a mesma ideia mas aplicando-a apenas à lista `ais`. O mesmo podemos fazer para a lista `cis`. A função completa fica, então:

```
trelica(ais, bis, cis) =
  begin
    nos_trelica(ais)
    nos_trelica(bis)
    nos_trelica(cis)
    barras_trelica(ais, cis)
    barras_trelica(bis, ais)
    barras_trelica(bis, cis)
    barras_trelica(bis, ais[2:end])
    barras_trelica(bis, cis[2:end])
    barras_trelica(ais[2:end], ais)
    barras_trelica(cis[2:end], cis)
    barras_trelica(bis[2:end], bis)
  end
```

As funções anteriores constroem treliças com base nas funções "elementares" `no_trelica` e `barra_trelica`. Embora o seu significado seja óbvio,

[3] Esta forma de percorrer várias listas em simultâneo emprega o conceito de *tuplo*, uma estrutura de dados que ainda não discutimos mas que existe em Julia.

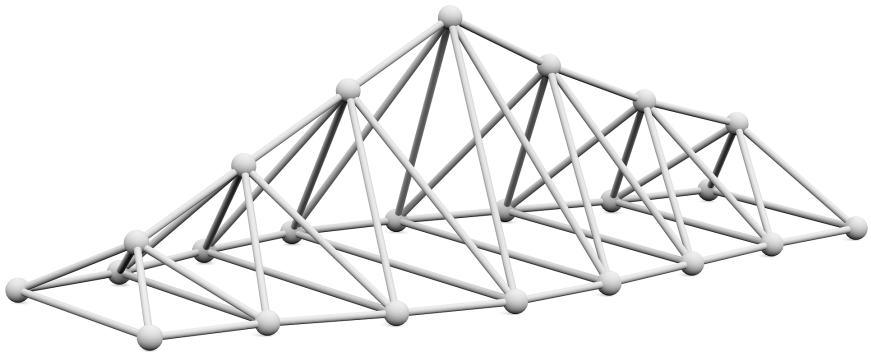


Figura 5.15: Treliça construída a partir de pontos especificados arbitrariamente.

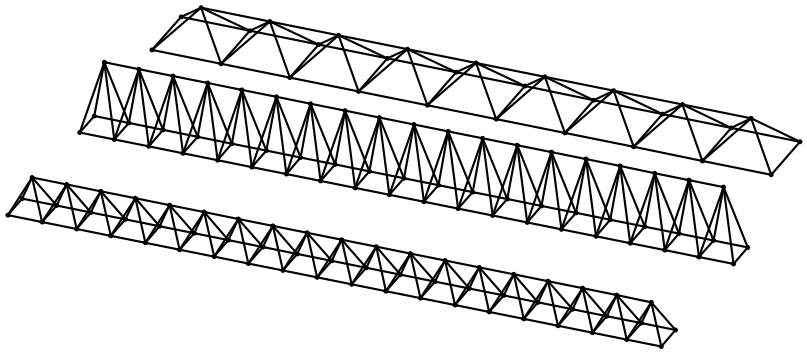
ainda não definimos estas funções e existem várias possibilidades. Numa primeira abordagem, vamos considerar que cada nó da treliça será constituído por uma esfera onde se irão unir as barras, barras essas que serão definidas por cilindros. O raio das esferas e da base dos cilindros será determinado por um nome global, para que possamos facilmente alterar o seu valor. Assim, temos:

```
raio_no_trelica = 0.1

no_trelica(p) =
  sphere(p, raio_no_trelica)

raio_barra_trelica = 0.03

barra_trelica(p0, p1) =
  cylinder(p0, raio_barra_trelica, p1)
```

Podemos agora criar as treliças com as formas que entendermos. A Figura 5.15 mostra uma treliça desenhada a partir da expressão:

```
trelica([xyz(0, -1, 0), xyz(1, -1.1, 0), xyz(2, -1.4, 0),
         xyz(3, -1.6, 0), xyz(4, -1.5, 0), xyz(5, -1.3, 0),
         xyz(6, -1.1, 0), xyz(7, -1, 0)],
        [xyz(0.5, 0, 0.5), xyz(1.5, 0, 1), xyz(2.5, 0, 1.5),
         xyz(3.5, 0, 2), xyz(4.5, 0, 1.5), xyz(5.5, 0, 1.1),
         xyz(6.5, 0, 0.8)],
        [xyz(0, 1, 0), xyz(1, 1.1, 0), xyz(2, 1.4, 0),
         xyz(3, 1.6, 0), xyz(4, 1.5, 0), xyz(5, 1.3, 0),
         xyz(6, 1.1, 0), xyz(7, 1, 0)])
```

Exercício 5.6.1 Defina uma função denominada `trelica_recta` capaz de construir qualquer uma das treliças que se apresentam na imagem seguinte.



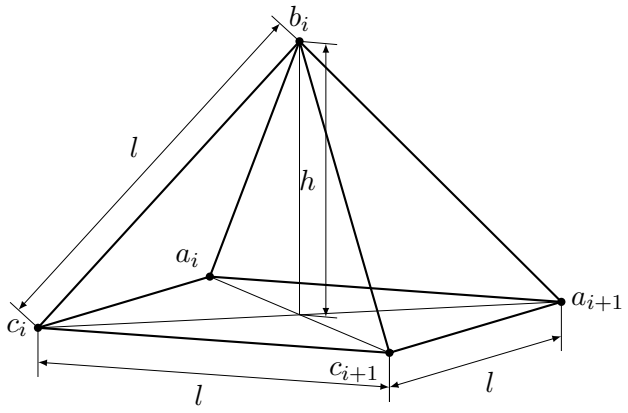
Para simplificar, considere que as treliças se desenvolvem segundo o eixo X. A função `trelica_recta` deverá receber o ponto inicial da treliça, a altura e largura da treliça e o número de nós das fileiras laterais. Com esses valores, a função deverá produzir três listas de coordenadas que passará como argumentos à função `trelica`. Como exemplo, considere que as três treliças apresentadas na imagem anterior foram o resultado da avaliação das expressões:

```
trelica_recta(xyz(0, 0, 0), 1.0, 1.0, 20)
trelica_recta(xyz(0, 5, 0), 2.0, 1.0, 20)
trelica_recta(xyz(0, 10, 0), 1.0, 2.0, 10)
```

Sugestão: comece por definir a função `coordenadas_x` que, dado um ponto inicial p, um afastamento l entre pontos e um número n de pontos, devolve uma lista com as coordenadas dos n pontos dispostos ao longo do eixo X.

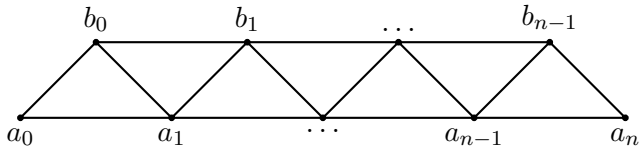
Exercício 5.6.2 O custo total de uma treliça é muito dependente do número de diferentes comprimentos que as barras podem ter: quanto menor for esse número, maiores economias de escala se conseguem obter e, consequentemente, mais económica fica a treliça. O caso ideal é aquele em que existe um único comprimento igual para todas as barras.

Atendendo ao seguinte esquema, determine a altura h da treliça em função da largura l do *módulo* de modo a que todas as barras tenham o mesmo comprimento.



Defina ainda a função `trelica_modulo` que constrói uma treliça com barras todas do mesmo comprimento, orientada segundo o eixo X. A função deverá receber o ponto inicial da treliça, a largura da treliça e o número de nós das fileiras laterais.

Exercício 5.6.3 Considere o desenho de uma treliça *plana*, tal como se apresenta na seguinte figura:

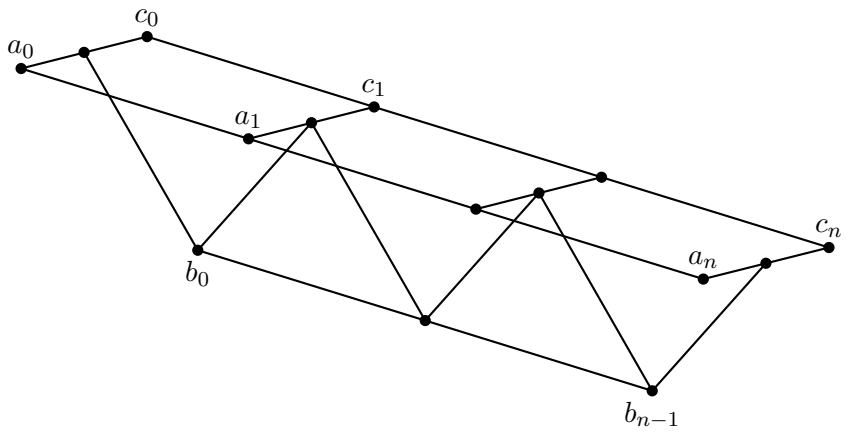


Defina uma função `trelica_plana` que recebe, como parâmetros, duas listas correspondentes às posições desde a_0 até a_n e desde b_0 até b_{n-1} e que cria os nós nessas posições e as barras que os unem. Considere, como pré-definidas, as funções `nos_trelica` que recebe uma lista de posições como argumento e `barras_trelica` que recebe duas listas de posições como argumentos.

Teste a função de definir com a seguinte expressão:

```
trelica_plana(coordenadas_x(xyz(0, 0, 0), 2.0, 20),
              coordenadas_x(xyz(1, 0, 1), 2.0, 19))
```

Exercício 5.6.4 Considere o desenho da treliça especial apresentado na seguinte figura:



Defina uma função `trelica_especial` que recebe, como parâmetros, três listas de pontos correspondentes aos pontos desde a_0 até a_n, desde b_0 até b_{n-1} e desde c_0 até c_n e que cria os nós nesses pontos e as barras que os unem. Considere, como pré-definidas, as funções `nos_trelica` que recebe uma lista de posições como argumento e `barras_trelica` que recebe duas listas de posições como argumentos.

5.6.2 Geração de Posições

Como vimos na secção anterior, podemos idealizar um processo de criação de uma treliça a partir das listas de posições dos seus nós. Estas listas, naturalmente, podem ser especificadas manualmente mas esta abordagem só será realizável para treliças muito pequenas. Ora sendo uma treliça uma estrutura capaz de vencer vãos muito grandes, no caso geral, o número de nós da treliça é demasiado elevado para que possamos produzir manualmente as listas de posições. Para resolver este problema temos de pensar em processos automatizados para criar essas listas, processos esses que tenham em conta a geometria pretendida para a treliça.

A título de exemplo, idealizemos um processo de criação de treliças em arco, em que as sequências de nós a_i, b_i e c_i formam arcos de circunferência. A Figura 5.16 mostra uma versão de uma destas treliças, definida pelos arcos de circunferência de raio r_0 e r_1.

Para tornar a treliça uniforme, os nós encontram-se igualmente espaçados ao longo do arco. O ângulo Δ_ψ corresponde a esse espaçamento e calcula-se pela divisão da amplitude angular do arco pelo número de nós pretendidos n. Atendendo a que o arco intermédio tem sempre menos um nó do que os arcos laterais, temos de dividir o ângulo Δ_ψ pelas duas extremidades do arco intermédio, de modo a centrar os nós desse arco entre os nós dos arcos laterais, tal como se pode verificar na Figura 5.16.

Uma vez que o arco é circular, a forma mais simples de calcularmos as posições dos nós será empregando coordenadas esféricas (ρ, ϕ, ψ). Esta

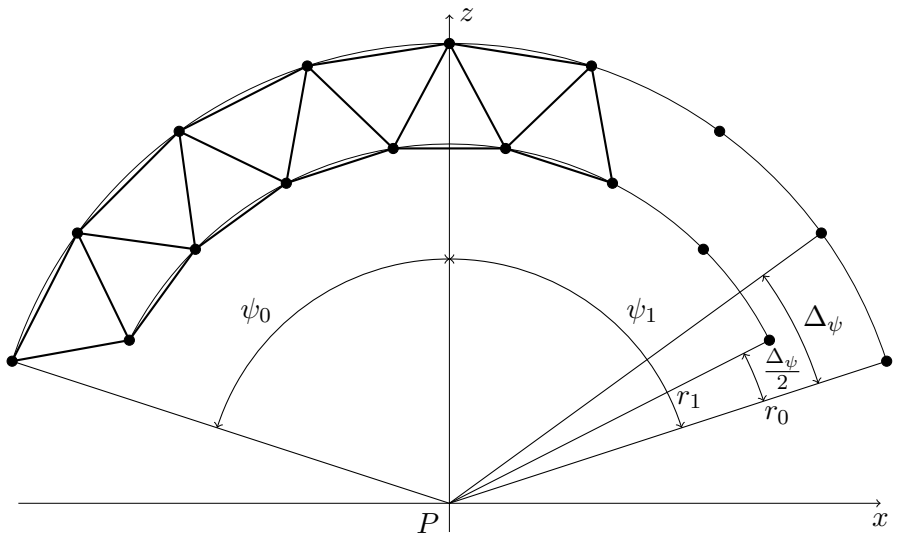


Figura 5.16: Alçado frontal de uma treliça em forma de arco de círculo.

decisão leva-nos a considerar que os ângulos inicial e final dos arcos devem ser medidos relativamente ao eixo Z, tal como é visível na Figura 5.16. Para flexibilizar a produção das coordenadas dos nós do arco vamos definir uma função que recebe o centro P do arco, o raio r desse arco, o ângulo ϕ, o ângulo inicial ψ, o incremento de ângulo Δ_ψ e, finalmente, o número de posições a produzir. Assim, temos:

```
pontos_arco(p, r, fi, psi0, dpsi, n) =
  if n == 0
    []
  else
    [p+vsph(r, fi, psi0),
     pontos_arco(p, r, fi, psi0+dpsi, dpsi, n-1)...]
  end
```

Para construirmos a treliça em arco podemos agora definir uma função que cria três dos arcos anteriores. Para isso, a função terá de receber o centro P do arco central, o raio r_{ac} dos arcos laterais, o raio r_b do arco central, o ângulo ϕ, os ângulos inicial ψ_0 e final ψ_1 e, ainda, a separação e entre os arcos laterais e o número n de nós dos arcos laterais. A função irá calcular o incremento $\Delta_\psi = \frac{\psi_1 - \psi_0}{n}$ e, de seguida, invoca a função `trelica` com os parâmetros apropriados:

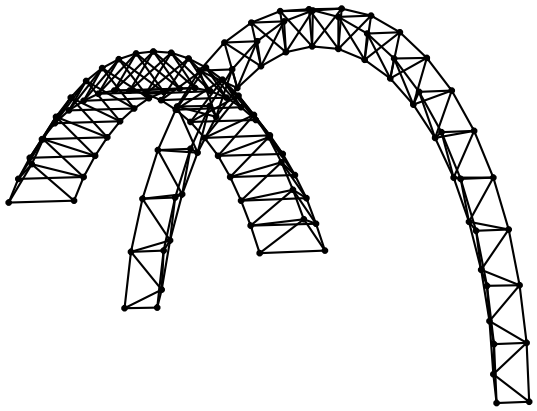


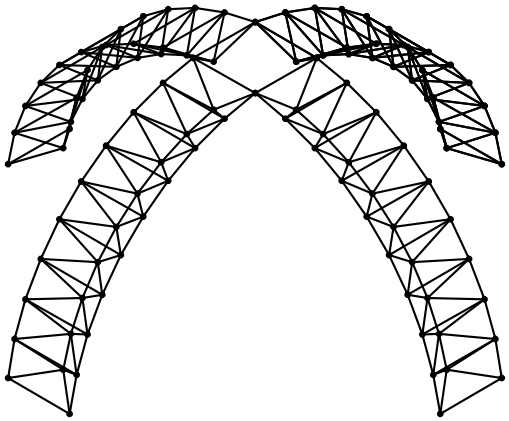
Figura 5.17: Treliças em arco criadas com parâmetros diferentes.

```
trelica_arco(p, rac, rb, fi, psi0, psi1, e, n) =
  let dpsi = (psi1-psi0)/(n-1),
      v0 = vpol(e/2, fi+pi/2),
      v1 = vpol(e/2, fi-pi/2)
    trelica(pontos_arco(p+v0, rac, fi, psi0, dpsi, n),
            pontos_arco(p, rb, fi, psi0+dpsi/2, dpsi, n-1),
            pontos_arco(p+v1, rac, fi, psi0, dpsi, n))
  end
```

A Figura 5.17 mostra as treliças construídas a partir das expressões:

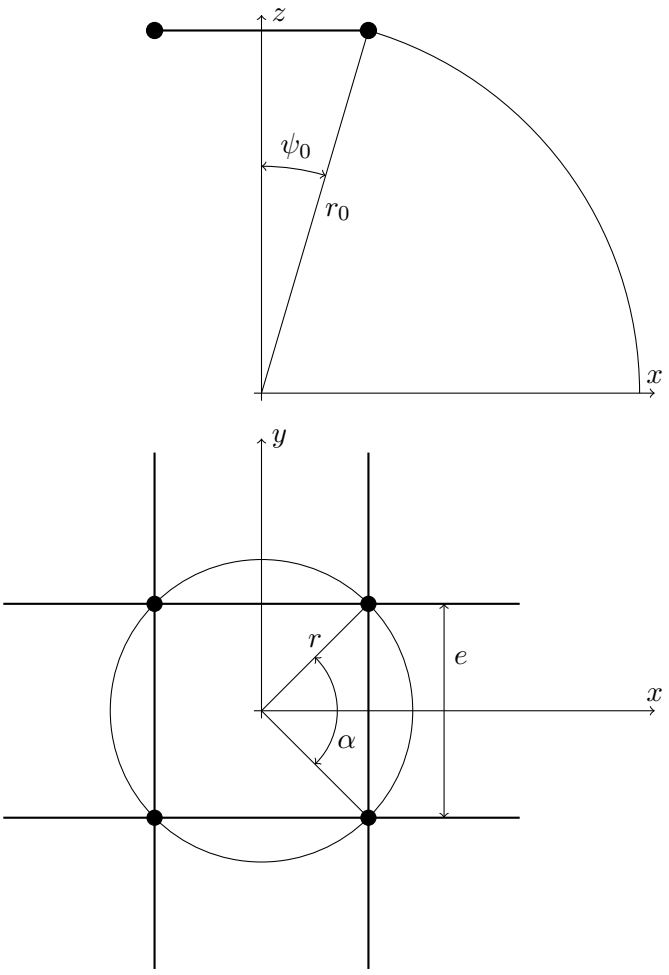
```
trelica_arco(xyz(0, 0, 0), 10, 9, 0, -pi/2, pi/2, 1.0, 20)
trelica_arco(xyz(0, 5, 0),  8, 9, 0, -pi/3, pi/3, 2.0, 20)
```

Exercício 5.6.5 Considere a construção de abóbabas apoiadas em treliças distribuidas radialmente, tal como a que se apresenta na imagem seguinte:



Esta abóbada é constituída por um determinado número de treliças de arco circular. A largura e de cada treliça e o ângulo inicial ψ_0 a que se dá

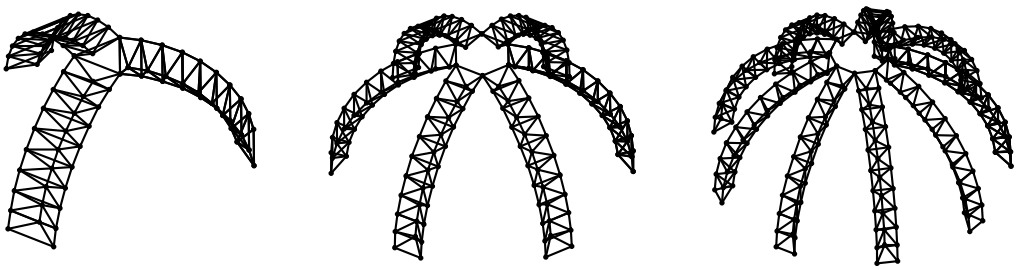
o arranque de cada treliça são tais que os nós dos topos das treliças são coincidentes dois a dois e estão dispostos ao longo de um círculo de raio r, tal como se apresenta no esquema seguinte:



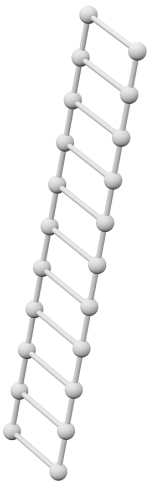
Defina a função `abobada_trelicas` que constrói uma abóbada de trelicas a partir do centro da abóbada P, do raio r_{ac} dos arcos laterais de cada treliça, do raio r_b do arco central de cada treliça, do raio r do "fecho" das treliças, do número de nós n em cada treliça e, finalmente, do número de treliças n_ϕ.

A título de exemplo, considere a figura seguinte que foi produzida pela avaliação das seguintes expressões:

```
abobada_trelicas(xyz(0, 0, 0), 10, 9, 2.0, 10, 3)
abobada_trelicas(xyz(25, 0, 0), 10, 9, 2.0, 10, 6)
abobada_trelicas(xyz(50, 0, 0), 10, 9, 2.0, 10, 9)
```

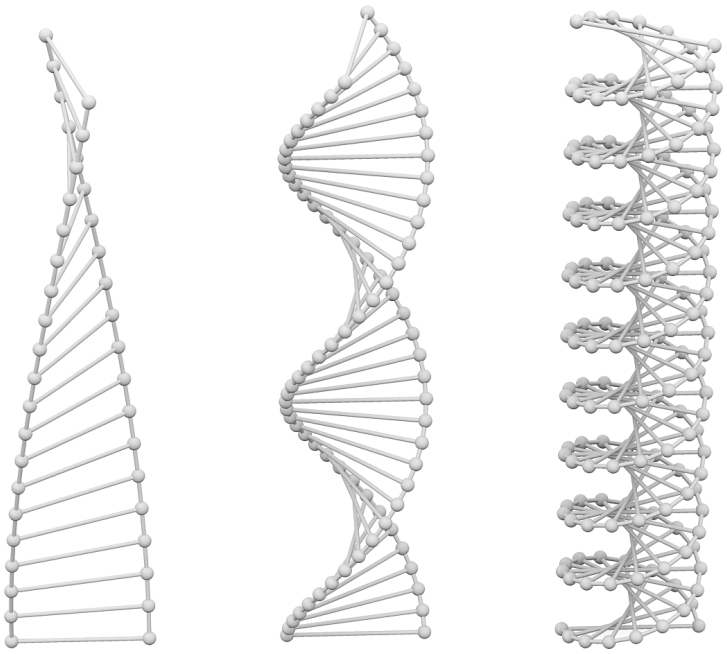


Exercício 5.6.6 Considere o desenho de uma escada de mão idêntica à que se apresenta na seguinte figura:



Repare que a escada de mão pode ser vista como uma versão (muito) simplificada de uma treliça composta por apenas duas sequências de nós. Defina a função `escada_de_mao` que recebe, como parâmetros, duas listas de pontos correspondentes aos centros das sequências de nós e que cria os nós nesses pontos e as barras que os unem. Considere, como pré-definidas, as funções `nos_trelica` que recebe uma lista de posições como argumento e `barras_trelica` que recebe duas listas de posições como argumentos.

Exercício 5.6.7 Defina uma função capaz de representar a dupla hélice do genoma, tal como se pode ver na seguinte imagem:



A função deverá receber uma posição referente ao centro da base do genoma, o raio da hélice do genoma, a diferença angular entre os nós, a diferença em altura entre nós e, finalmente, o número de nós em cada hélice. Os genomas apresentados na imagem anterior foram gerados pelas seguintes expressões:

```
genoma(xyz(0, 0, 0), 1.0, pi/32, 0.5, 20)
genoma(xyz(4, 0, 0), 1.0, pi/16, 0.25, 40)
genoma(xyz(8, 0, 0), 1.0, pi/8, 0.125, 80)
```

5.6.3 Treliças Espaciais

Vimos como é possível definir treliças a partir de três listas, cada uma contendo as coordenadas dos nós a que as barras das treliças se ligam. Ligando entre si várias destas treliças é possível produzir uma estrutura ainda maior a que se dá o nome de *treliça espacial*. A Figura 5.18 mostra um exemplo onde são visíveis três treliças espaciais.

Para podermos definir um algoritmo que gere treliças espaciais é importante termos em conta que embora este tipo de treliças aglomere várias treliças simples, estas estão interligadas de tal modo que cada treliça partilha um conjunto de nós e barras com a treliça que lhe é adjacente, tal como é visível na Figura 5.19 que apresenta um esquema de uma treliça espacial. Assim, se uma treliça espacial for constituída por duas treliças simples interligadas, a treliça espacial será gerada, não por seis listas de coordenadas, mas apenas por cinco listas de coordenadas. No caso geral, uma treliça espacial constituída por n treliças simples será definida por um $2n + 1$ listas

Figura 5.18: Treliças espaciais no estádio Al Ain nos Emiratos Árabes Unidos. Fotografia de Klaus Knebel.

de pontos, i.e., por um número ímpar de listas de pontos (no mínimo, três listas).

A definição da função que contrói uma treliça espacial segue a mesma lógica da função que constrói uma treliça simples só que agora, em vez de operar com apenas três listas, opera com um número impar delas. Assim, a partir de uma lista contendo um número ímpar de listas de coordenadas, iremos processar essas listas de coordenadas duas a duas, sabendo que a "terceira" lista de coordenadas $c_{i,j}$ da treliça i é também a "primeira" lista de coordenadas $a_{i+1,j}$ da treliça seguinte $i+1$.

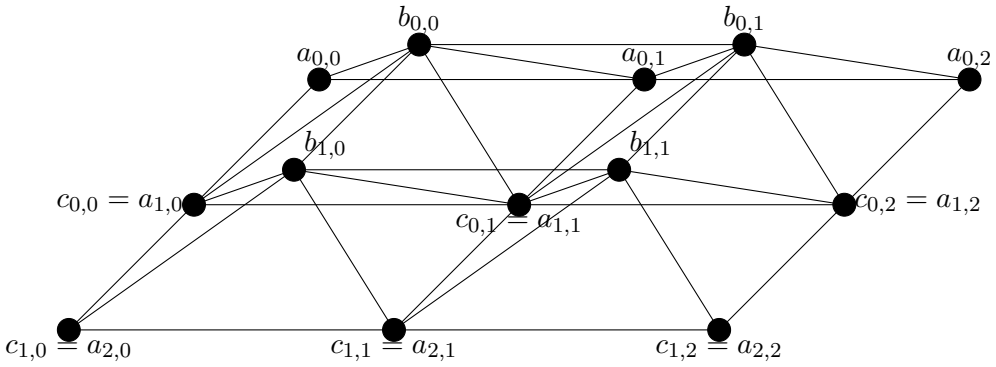


Figura 5.19: Esquema de ligação de barras de uma treliça espacial.

Uma vez que processamos duas listas de cada vez e partimos de um número ímpar de listas, no caso "final" restará apenas uma lista de coordenadas que deverá "fechar" a construção da treliça.

Há, no entanto, uma dificuldade adicional: para que a treliça plana tenha rigidez transversal é ainda necessário interligar entre si os nós centrais das várias treliças. Estes travamentos correspondem a ligar cada nó $b_{i,j}$ ao nó $b_{i+1,j}$. Assim, à medida que formos processando as listas de coordenadas, iremos também estabelecer os travamentos entre as listas correspondentes. Todo este processo é implementado pela seguinte função:

```
trelica_espacial(ptss) =
  let ais = ptss[1],
      bis = ptss[2],
      cis = ptss[3]
    nos_trelica(ais)
    nos_trelica(bis)
    barras_trelica(ais, cis)
    barras_trelica(bis, ais)
    barras_trelica(bis, cis)
    barras_trelica(bis, ais[2:end])
    barras_trelica(bis, cis[2:end])
    barras_trelica(ais[2:end], ais)
    barras_trelica(bis[2:end], bis)
    if ptss[4:end] == []
      nos_trelica(cis)
      barras_trelica(cis[2:end], cis)
    else
      barras_trelica(bis, ptss[4])
      trelica_espacial(ptss[3:end])
    end
  end
```

Exercício 5.6.8 Na realidade, uma treliça simples é um caso particular de uma treliça espacial. Redefina a função `trelica` de modo a que ela use a função `trelica_espacial`.

Agora que já sabemos construir treliças espaciais a partir de uma lista de listas de coordenadas, podemos pensar em mecanismos para gerar esta lista de listas. Um exemplo simples é o de uma treliça espacial horizontal, tal como a que se apresenta na Figura 5.20.

Para gerar as coordenadas dos nós desta treliça, podemos definir uma função que, com base no número de pirâmides pretendidas e na largura da base da pirâmide, gera os nós ao longo de uma das dimensões, por exemplo, a dimensão X:

```
coordenadas_x(p, l, n) =
  [p+vx(i*l) for i in 0:n-1]
```

Em seguida, basta-nos definir uma outra função que itera a anterior ao longo da outra dimensão Y, de modo a gerar uma linha de nós a_i, seguida

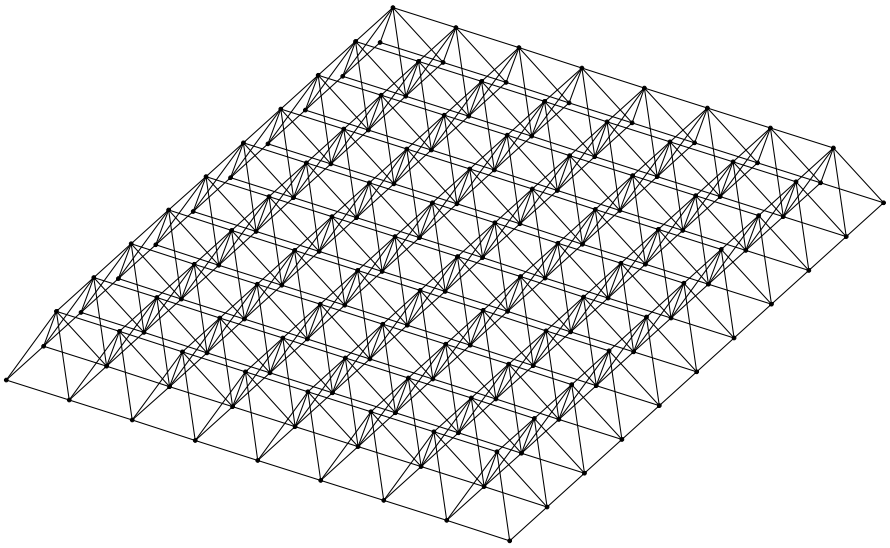


Figura 5.20: Uma treliça espacial horizontal, composta por oito treliças simples com dez pirâmides cada uma.

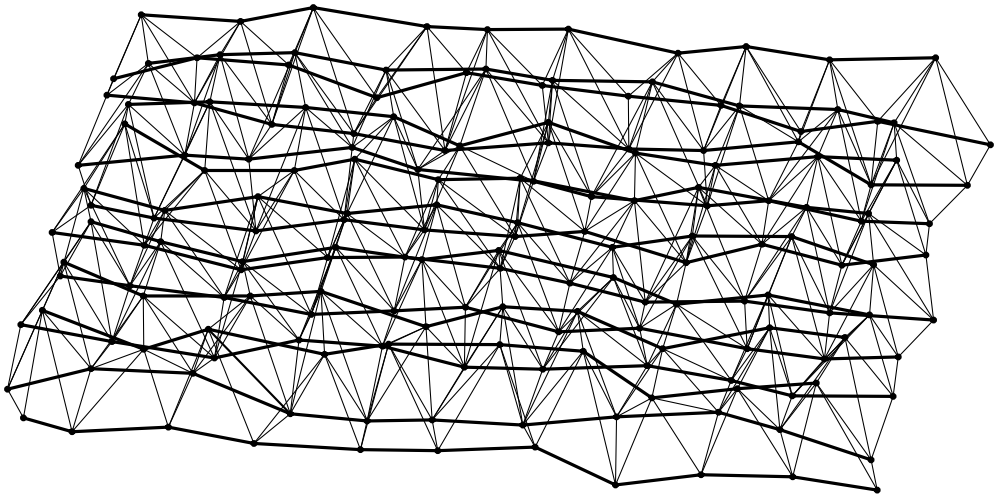
de outra linha b_i deslocada para o centro da pirâmide e à altura desta, seguida das restantes linhas, até o final, em que teremos de produzir mais uma linha a_i. É ainda necessário termos em conta que as linhas b_i têm menos um nó do que as linhas a_i. Com base nestas considerações, podemos escrever:

```
coordenadas_trelica_horizontal(p, h, l, m, n) =
  if m == 0
    [coordenadas_x(p, l, n)]
  else
    [coordenadas_x(p, l, n),
     coordenadas_x(p+vxyz(l/2, l/2, h), l, n-1),
     coordenadas_trelica_horizontal(p+vy(l), h, l, m-1, n)...]
  end
```

Podemos agora combinar a lista de listas de coordenadas produzidas pela função anterior com a que constrói uma treliça espacial. A título de exemplo, a seguinte expressão produz a treliça apresentada na Figura 5.20:

```
trelica_espacial(
  coordenadas_trelica_horizontal(xyz(0, 0, 0), 1, 1, 8, 10))
```

Exercício 5.6.9 Considere a construção de uma treliça espacial *aleatória*. Esta treliça caracteriza-se por as coordenadas dos seus nós estarem posicionados a uma distância aleatória das coordenadas dos nós correspondentes de uma treliça espacial horizontal, tal como é exemplificado pela seguinte figura onde, para facilitar a visualização, se assinalou a traço mais grosso as barras que unem os nós a_i, b_i e c_i do esquema apresentado na Figura 5.19.

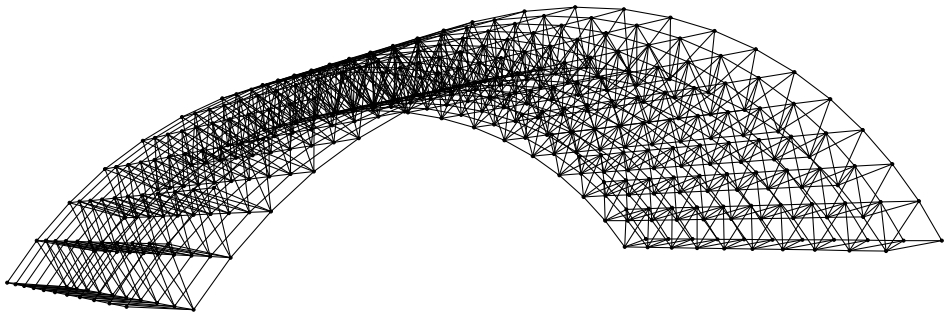


Defina a função `coordenadas_trelica_aleatoria` que, para além dos parâmetros da função `coordenadas_trelica_horizontal`, recebe ainda a distância máxima r a que cada nó da treliça aleatória pode ser colocado relativamente ao nó correspondente da treliça horizontal. A título de exemplo, considere que a treliça apresentada na figura anterior foi gerada pela avaliação da expressão:

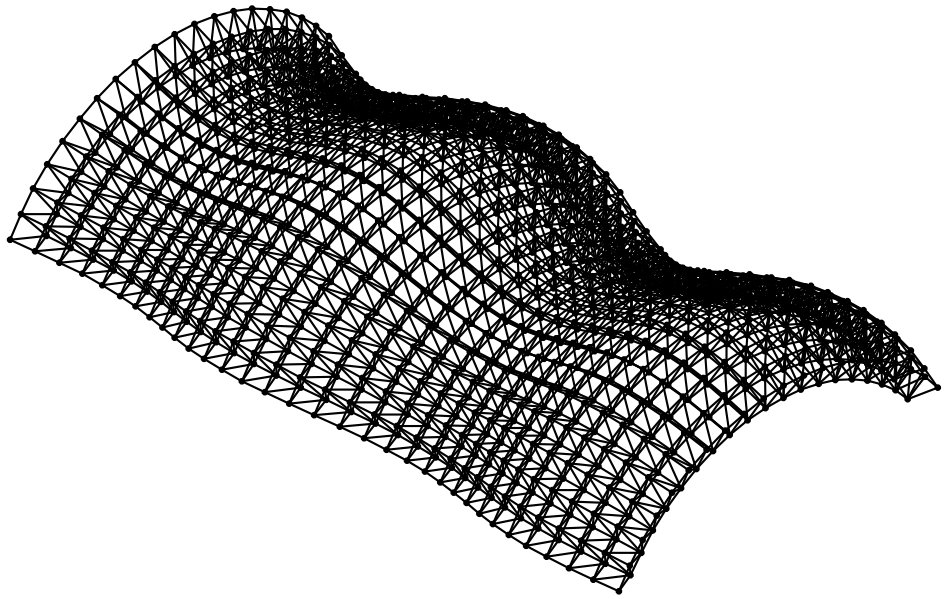
```
trelica_espacial(
  coordenadas_trelica_aleatoria(xyz(0, 0, 0), 1, 1, 8, 10, 0.2))
```

Exercício 5.6.10 Considere a construção de uma treliça espacial em arco, tal como a que se apresenta na imagem seguinte (em perspectiva). Defina a função `trelica_espacial_arco` que, para além dos parâmetros da função `trelica_arco`, possui ainda um parâmetro adicional que indica o número de treliças simples que constituem a treliça espacial. A título de exemplo, considere que a treliça apresentada na imagem seguinte foi gerada pela expressão:

```
trelica_espacial_arco(xyz(0, 0, 0), 10, 9, 1.0, 20, 0, pi/3, pi/3, 10)
```



Exercício 5.6.11 Considere a treliça apresentada na imagem seguinte:



Esta treliça é semelhante à treliça espacial em arco mas com uma *nuance*: os raios exterior `rac` e interior `rb` variam ao longo do eixo do arco. Esta variação corresponde a uma sinusoide de amplitude Δ_r a variar desde um valor inicial α_0 até um valor final α_1, em incrementos de Δ_α.

Defina a função `trelica_ondulada` que constrói este tipo de treliças, a partir dos mesmos parâmetros da função `trelica_espacial_arco` e ainda dos parâmetros α_0, α_1, Δ_α e Δ_r. Como exemplo, considere que a figura anterior foi gerada pela invocação seguinte:

```
trelica_ondulada(xyz(0, 0, 0), 10, 9, 1.0, 20, 0, -pi/3, pi/3, 0, 4*pi, pi/8, 1)
```

Capítulo 6

Formas

6.1 Introdução

Até agora temos lidado apenas com curvas e sólidos simples. Nesta secção vamos discutir formas mais complexas criadas a partir da união, intersecção e subtracção de formas mais simples. Como iremos ver, estas operações são muito utilizadas em arquitectura.

A Figura 6.1 ilustra o templo de *Portuna* (também conhecido, erradamente, por templo de *Fortuna Virilis*), uma construção do século um antes de Cristo caracterizada por usar colunas, não como elemento eminentemente estrutural, mas sim como elemento decorativo. Para isso, os arquitectos consideraram uma *união* entre uma parede estrutural e um conjunto de colunas, de modo a que as colunas ficassem embebidas na parede. Esta mesma abordagem é visível noutros monumentos como, por exemplo, o Coliseu de Roma.

Já no caso do edifício visivel na Figura 6.2, da autoria do atelier de Michel Rojkind, o arquitecto criou uma forma inovadora através da *subtracção* de esferas a formas paralelipipédicas.

A Figura 6.3 demonstra um terceiro exemplo. Neste caso, trata-se da Catedral da Sagrada Família, do arquitecto Catalão Antoni Gaudí. Como iremos ver na secção 6.7, algumas das colunas idealizadas por Gaudí para esta obra resultam da *intersecção* de prismas torcidos.

6.2 Geometria Construtiva

A *geometria construtiva de sólidos* é uma das técnicas mais usadas para a modelação de sólidos. Esta abordagem baseia-se na combinação de sólidos simples, como paralelipípedos, esferas, pirâmides, cilindros, toros, etc. Cada um destes sólidos pode ser visto como um conjunto de pontos no espaço e a sua combinação é feita usando operações de conjuntos como a

Figura 6.1: Templo de Portuna em Roma, Itália. Fotografia de Rickard Lamré.

Figura 6.2: Nestlé Application Group em Querétaro, México. Fotografia de Paúl Rivera - archphoto.

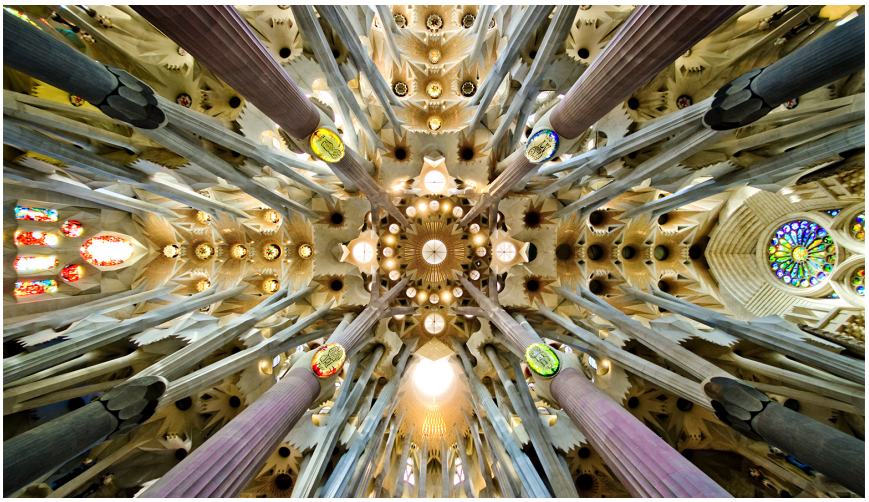


Figura 6.3: Colunas da Catedral da Sagrada Família em Barcelona, Espanha. Fotografia de Salvador Busquets Artigas.

união, a intersecção e a subtracção desses conjuntos de pontos. Por simplicidade de expressão, vamos denominar o conjunto de pontos no espaço por *região*.

Comecemos por considerar a operação de *união*. Dadas as regiões R_0 e R_1, a sua união $R_0 \cup R_1$ é o conjunto de pontos que pertence a R_0 ou que pertence a R_1. Esta operação é implementada no Khepri pela função `union`. A Figura 6.4 mostra, à esquerda, a união entre um cubo e uma esfera, produzida pelas seguintes expressões:

```
cubo = box(xyz(0, 0, 0), xyz(1, 1, 1))
esfera = sphere(xyz(0, 1, 1), 0.5)
union(cubo, esfera)
```

Uma outra operação é a *intersecção* de regiões $R_0 \cap R_1$, que produz o conjunto de pontos que pertencem, simultaneamente, aos conjuntos R_0 e R_1 e que, no Khepri, se obtém através da função `intersection`. A segunda imagem da Figura 6.4 mostra a intersecção entre um cubo e uma esfera e foi produzida pelas seguintes expressões:

```
cubo = box(xyz(2, 0, 0), xyz(3, 1, 1))
esfera = sphere(xyz(2, 1, 1), 0.5)
intersection(cubo, esfera)
```

Finalmente, existe ainda a operação de *subtracção* de regiões $R_0 \setminus R_1$ que corresponde ao conjunto de pontos que pertence a R_0 mas que não pertence a R_1. Ao contrário das anteriores, esta operação não é comutativa. Assim, é diferente subtrair uma esfera a um cubo de subtrair um cubo a uma esfera. Essa diferença é visivel na duas imagens à direita da Figura 6.4 que foram produzidas pelas expressões:

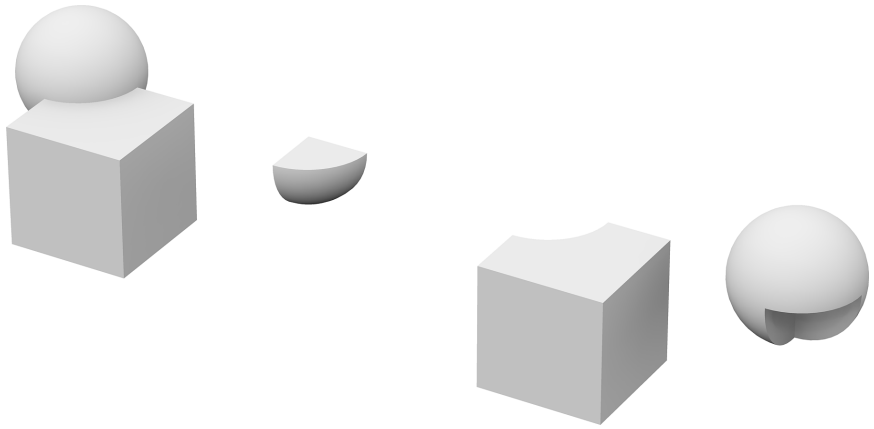


Figura 6.4: Combinações de volumes. A imagem da esquerda representa a união de um cubo com uma esfera, a imagem do centro representa a intersecção do cubo com a esfera e as imagens da direita representam a subtracção do cubo pela esfera e a subtracção da esfera pelo cubo.

```
cubo = box(xyz(4, 0, 0), xyz(5, 1, 1))
esfera = sphere(xyz(4, 1, 1), 0.5)
subtraction(cubo, esfera)
```

e pelas expressões:

```
cubo = box(xyz(6, 0, 0), xyz(7, 1, 1))
esfera = sphere(xyz(6, 1, 1), 0.5)
subtraction(esfera, cubo)
```

À semelhança de outras funções já discutidas, como `line` e `spline`, as funções `union`, `intersection`, e `subtraction` recebem qualquer número de argumentos ou, alternativamente, uma lista com todos os argumentos. Como exemplo de utilização destas operações, consideremos três cilindros dispostos segundo os eixos X, Y e Z. A união destes cilindros está visível no lado esquerdo da Figura 6.5 e foi gerada pela expressão:

```
union(cylinder(xyz(-1, 0, 0), 1, xyz(1, 0, 0)),
      cylinder(xyz(0, -1, 0), 1, xyz(0, 1, 0)),
      cylinder(xyz(0, 0, -1), 1, xyz(0, 0, 1)))
```

Este objecto tem a propriedade de, quando visto em planta, alçado lateral e alçado frontal, se reduzir a um quadrado. No lado direito da mesma figura encontra-se um sólido ainda mais interessante, gerado pela intersecção dos mesmos três cilindros, produzindo um objecto que, obviamente, não é uma esfera mas que, tal como a esfera, projecta um círculo em planta, em alçado frontal e em alçado lateral.

Para melhor percebermos a diferença entre o objecto construído pela intersecção dos cilindros e uma verdadeira esfera, podemos subtrair a esfera

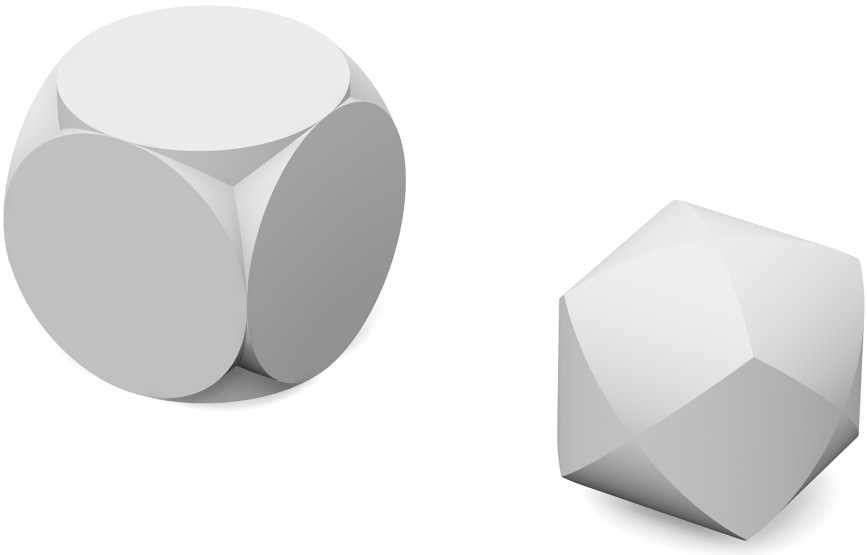


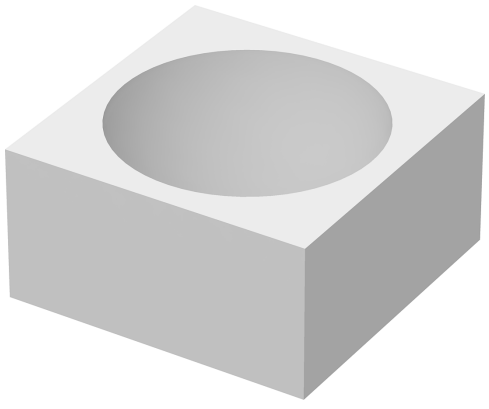
Figura 6.5: A união e intersecção de três cilindros dispostos ortogonalmente.

ao objecto. Para podermos "espreitar" o interior do objecto, vamos subtrair uma esfera com um raio (muito) ligeiramente maior que o dos cilindros:

```
subtraction(
    intersection(cylinder(xyz(-1, 0, 0), 1, xyz(1, 0, 0)),
                 cylinder(xyz(0, -1, 0), 1, xyz(0, 1, 0)),
                 cylinder(xyz(0, 0, -1), 1, xyz(0, 0, 1))),
    sphere(xyz(0, 0, 0), 1.01))
```

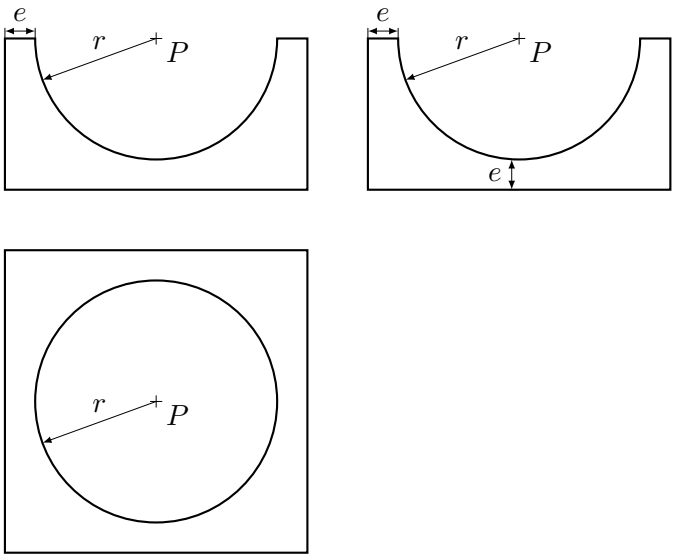
O resultado encontra-se representado na Figura 6.6.

Exercício 6.2.1 Pretende-se modelar uma bacia de pedra idêntica à que se apresenta na imagem seguinte:



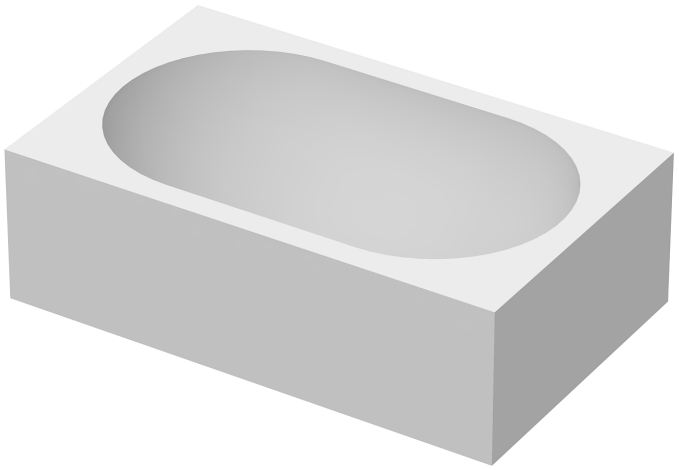
Os parâmetros relevantes para a bacia encontram-se representados no alçado frontal, planta e alçado lateral apresentados na imagem seguinte:

Figura 6.6: Subtracção de uma esfera à intersecção de três cilindros disposto ortogonalmente. A esfera tem um raio 1% superior ao dos cilindros por forma a permitir visualizar o interior.

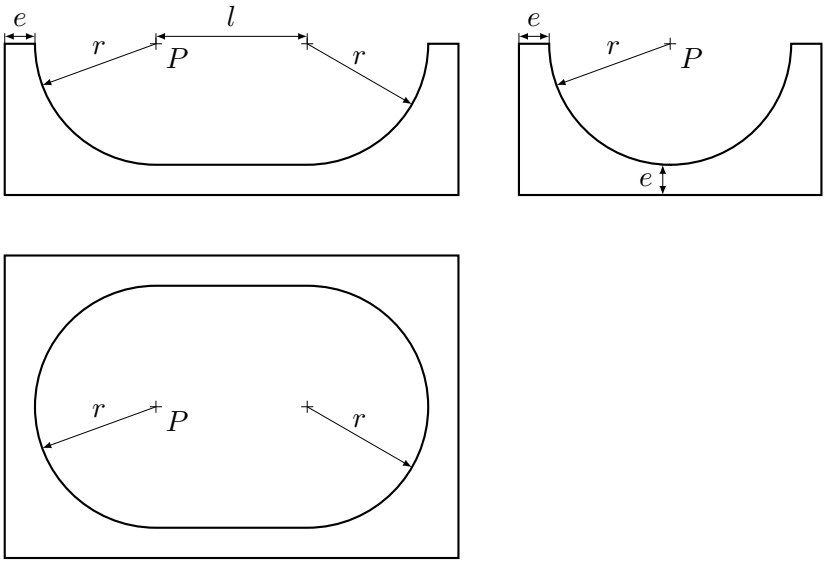


Defina uma função denominada `bacia` que constroi uma bacia idêntica à da figura anterior.

Exercício 6.2.2 Pretende-se modelar uma banheira de pedra idêntica à que se apresenta na imagem seguinte:



Os parâmetros relevantes para a banheira encontram-se representados no alçado frontal, planta e alçado lateral apresentados na imagem seguinte:



Defina uma função denominada `banheira` que constroi uma banheira idêntica à da figura anterior.

6.3 Superfícies

Até agora, temos usado o Khepri para a criação de curvas, que visualizamos normalmente em duas dimensões, por exemplo, no plano XY, ou para a criação de sólidos, que visualizamos a três dimensões. Agora, vamos ver que o Khepri também permite a criação de *superfícies*.

Existem muitas formas de o Khepri criar uma superfície. Várias das funções que produzem curvas fechadas possuem uma versão com os mesmos parâmetros mas com a diferença de produzirem a superfície delimitada por essas curvas. As funções `surface_circle`, `surface_rectangle`, `surface_polygon` e `surface_regular_polygon` recebem exactamente os mesmos argumentos que, respecivamente, as funções `circle`, `rectangle`, `polygon`, e `regular_polygon`, mas produzem superfícies em vez de curvas. Para além destas, existe ainda a função `surface_arc` que produz uma superfície limitada por um arco e pelos raios que as extremidades desse arco fazem com o centro do arco. Finalmente, existe ainda a função `surface` que recebe uma curva ou lista de curvas e produz uma superfície delimitada por essa curva ou curvas. Na verdade, temos que:

```
surface_circle(...)≡surface(circle(...))

surface_rectangle(...)≡surface(rectangle(...))

surface_polygon(...)≡surface(polygon(...))
```

e, de forma idêntica, se estabelecem as equivalências para as restantes funções.

Tal como os sólidos, também as superfícies podem ser combinadas com as operações de união, intersecção e subtracção para criar superfícies mais complexas. Por exemplo, consideremos a seguinte união entre uma superfície triangular e um círculo

```
union(surface_polygon(xy(0, 0), xy(2, 0), xy(1, 1)),
      surface_circle(xy(1, 1), 0.5))
```

cujo resultado se encontra no canto superior esquerdo da Figura 6.7.

Se, no lugar da união, tivessemos optado pela intersecção, já o resultado seria o representado no canto superior direito da Figura 6.7. A subtracção do círculo ao triângulo está representada no canto inferior esquerdo da Figura 6.7 e, como a subtracção não é uma operação comutativa, podemos ainda fazer a subtracção inversa visível no canto inferior direito da Figura 6.7.

6.3.1 Trifólios, Quadrifólios e Outros Fólios

O trifólio é um elemento arquitectural que teve enorme divulgação durante o período Gótico. Trata-se de uma ornamentação constituída por três círculos tangentes dispostos em torno de um centro, usualmente associada ao topo das janelas Góticas mas que pode ocorrer em vários outros locais. Para além do trifólio, a arquitectura Gótica também explorou o quadrifólio, o pentafólio e outros "fólios." Na Figura 6.8 apresentamos vários exemplos deste elemento, a par de algumas suas derivações.

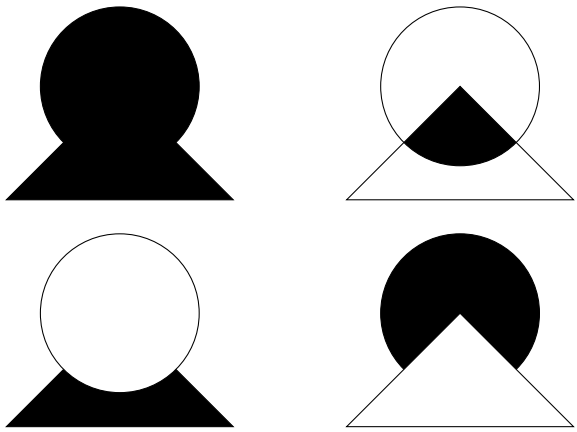


Figura 6.7: Operações de combinação de superfícies aplicadas a uma superfície triangular e a uma superfície circular. No canto superior esquerdo está representada a união e no direito a intersecção. No canto inferior esquerdo está representada a subtraccão do primeiro pelo segundo e no direito a subtraccão do segundo pelo primeiro.

Figura 6.8: Trifólios, quadrifólios, pentafólios e outros "fólios" numa janela da Catedral de São Pedro, em Exeter, Inglaterra. Fotografia de Chris Last.

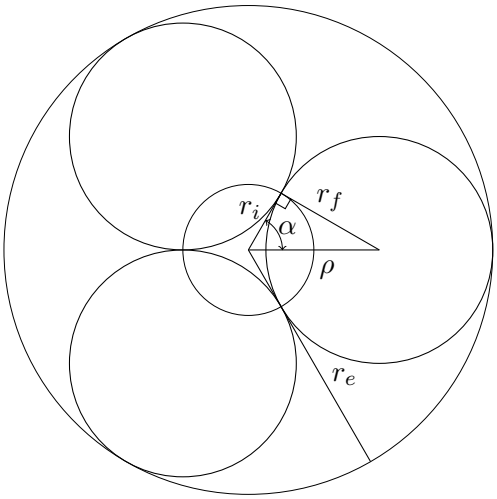


Figura 6.9: Os parâmetros de um trifólio.

Nesta secção vamos usar as operações que permitem criar e combinar superfícies para construir trifólios, quadrifólios e, mais genericamente, n-fólios. Por agora vamos considerar apenas o trifólio.

Para determinarmos os parâmetros de um trifólio vamos reportar-nos à Figura 6.9 onde apresentamos um trifólio "deitado." Nessa figura, podemos identificar r_e como o raio exterior do trifólio, r_f como o raio de cada "folha" do trifólio, ρ a distância do centro de cada folha ao centro do trifólio e r_i o raio do círculo interior do trifólio. Uma vez que o trifólio divide a circunferência em três partes iguais, o ângulo α corresponderá necessariamente a metade de um terço da circunferência, ou seja, $\alpha = \frac{\pi}{3}$. No caso de um quadrifólio teremos, obviamente, $\alpha = \frac{\pi}{4}$ e no caso geral de um n-fólio teremos $\alpha = \frac{\pi}{n}$.

Empregando as relações trigonométricas podemos relacionar os parâmetros do trifólio uns com os outros:

$$r_f = \rho \sin \alpha$$

$$r_i = \rho \cos \alpha$$

$$\rho + r_f = r_e$$

Se assumirmos que o parâmetro fundamental é o raio exterior do trifólio, r_e, então podemos deduzir que:

$$\rho = \frac{r_e}{1 + \sin \frac{\pi}{n}}$$

$$r_f = \frac{r_e}{1 + \frac{1}{\sin \frac{\pi}{n}}}$$

$$r_i = r_e \frac{\cos\frac{\pi}{n}}{1+\sin\frac{\pi}{n}}$$

A partir destas equações, podemos decompor o processo de criação de um n-fólio como a união de uma sucessão de círculos de raio r_f dispostos circularmente com um último círculo central de raio r_i. Transcrevendo para Julia, temos:

```
n_folio(p, re, n) =
  union(folhas_folio(p, re, n), circulo_interior_folio(p, re, n))
```

A função `circulo_interior_folio` define-se directamente a partir da fórmula do raio do círculo interior:

```
circulo_interior_folio(p, re, n) =
  surface_circle(p, re*cos(pi/n)/(1+sin(pi/n)))
```

Para a função `folhas_folio`, responsável por criar as folhas dispostas circularmente, vamos considerar o emprego de coordenadas polares. Cada folha (de raio r_i) será colocada numa coordenada polar determinada pelo raio ρ e por um ângulo ϕ que vamos recursivamente incrementando de $\Delta_\phi = \frac{2\pi}{n}$. Para isso, vamos definir uma nova função que implemente este processo:

```
uniao_circulos(p, ro, fi, d_fi, rf, n) =
  if n == 0
    ???
  else
    union(surface_circle(p+vpol(ro, fi), rf),
          uniao_circulos(p, ro, fi+d_fi, d_fi, rf, n-1))
  end
```

O problema que se coloca agora é saber o que é que a função devolve quando `n` é zero. Para melhor percebermos o problema, imaginemos que nomeamos os `n` círculos a unir de c_1, c_2, ..., c_n. Uma vez que, durante a recursão, a função vai deixando as uniões pendentes, quando atingimos a condição de paragem temos:

```
union(c1,
      union(c2,

                 ...

                      union(cn,
                            ???)))
```

Torna-se agora claro que `???` tem de ser algo que possa ser usado como argumento de uma união e que, para além disso, não afecta as uniões pendentes. Ora isto implica que `???` tem de ser o elemento neutro da união

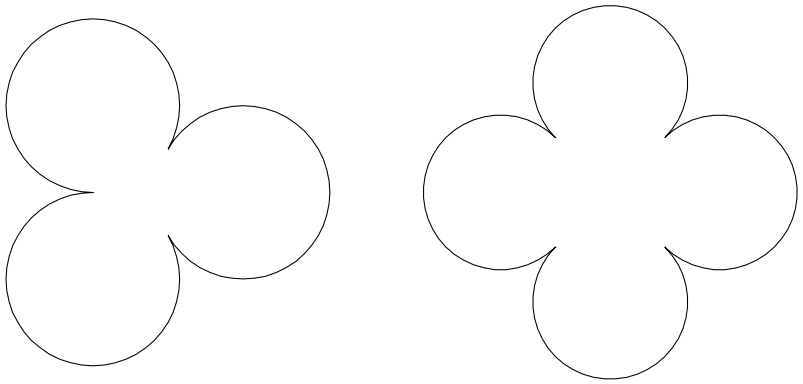


Figura 6.10: Um trifólio e um quadrifólio.

$\varnothing$. É precisamente para esse propósito que o Khepri providencia a função `empty_shape`. Quando invocada, a função `empty_shape` produz uma região vazia, i.e., um conjunto *vazio* de pontos no espaço, literalmente representando uma forma vazia que, consequentemente, é elemento neutro da união de formas.

Usando esta função, já podemos escrever o algoritmo completo:

```
uniao_circulos(p, ro, fi, d_fi, rf, n) =
  if n == 0
    empty_shape()
  else
    union(surface_circle(p+vpol(ro, fi), rf),
          uniao_circulos(p, ro, fi+d_fi, d_fi, rf, n-1))
  end
```

Estamos agora em condições de definir a função `folhas_folio` que invoca a anterior com os valores pré-calculados de ρ, Δ_ϕ e r_f. Para simplificar, vamos considerar um ângulo ϕ inicial de zero. Assim, temos:

```
folhas_folio(p, re, n) =
  uniao_circulos(p, re/(1+sin(pi/n)),
                 0,
                 2*pi/n,
                 re/(1+1/sin(pi/n)),
                 n)
```

Com base nestas funções, podemos criar um trifólio ao lado de um quadrifólio, tal como apresentamos na Figura 6.10, simplesmente avaliando as seguintes expressões:

```
n_folio(xy(0, 0), 1, 3)
n_folio(xy(2.5, 0), 1, 4)
```

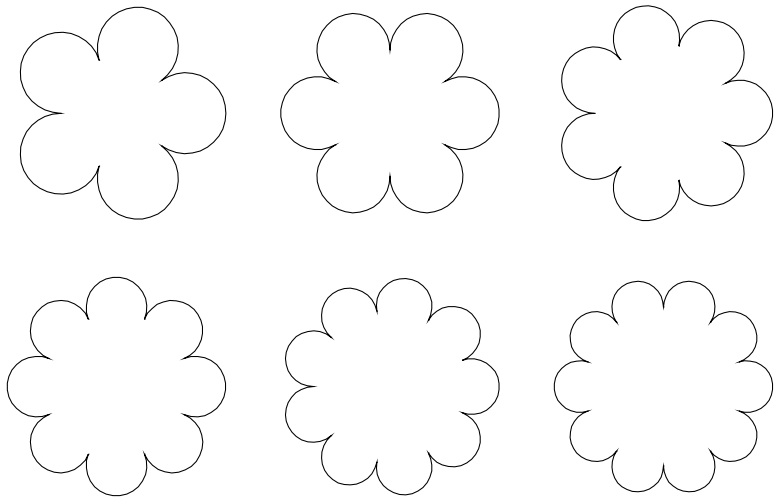


Figura 6.11: Fólios com número de folhas crescente. De cima para baixo e da esquerda para a direita estão representados um pentafólio, um hexafólio, um heptafólio, um octofólio, um eneafólio e um decafólio.

Naturalmente, podemos usar a função para gerar outros "fólios." A Figura 6.11 mostra um pentafólio, um hexafólio, um heptafólio, um octofólio, um eneafólio e um decafólio.

Tendo um mecanismo genérico para a construção de n-fólios, é tentador explorar os seus limites, em particular quando reduzimos o número de folhas. O que será um fólio de duas folhas?

Infelizmente, quando experimentamos criar um destes "bifólios," encontramos um erro provocado pela função `circulo_interior_folio`. O erro ocorre porque, na verdade, à medida que diminui o número de fólios, o raio do círculo interior vai-se reduzindo até se tornar simplesmente zero. De facto, para $n = 2$, o raio do círculo interior fica

$$r_i = r_e \frac{\cos \frac{\pi}{2}}{1 + \sin \frac{\pi}{2}}$$

ou seja,

$$r_i = r_e \frac{0}{2} = 0$$

Não é difícil corrigir o problema inserindo um teste na função `n_folio` que evite fazer a união das folhas com o círculo interior quando este tem raio zero. Contudo, essa não é a melhor opção pois, do ponto de vista matemático, o algoritmo de construção de fólios que tínhamos idealizado continua perfeitamente correcto: ele une um círculo interior a um conjunto de folhas. Acontece que quando o círculo interior tem raio zero, ele representa um conjunto vazio de pontos, ou seja, uma forma vazia, e a união de uma forma vazia aos círculos correspondentes às folhas não altera o resultado.

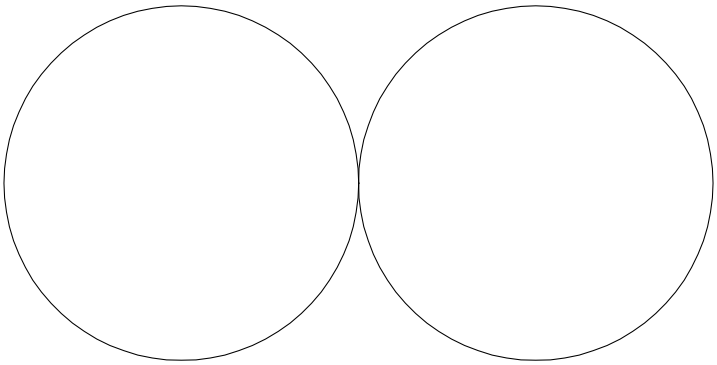


Figura 6.12: Um bifólio.

Com base na possibilidade de criar formas vazias, podemos agora redefinir a função `circulo_interior_folio` de modo a que, quando o número de folhas for dois, a função devolva uma forma vazia:

```
circulo_interior_folio(p, re, n) =
  if n == 2
    empty_shape()
  else
    surface_circle(p, re*cos(pi/n)/(1+sin(pi/n)))
  end
```

Empregando esta redefinição da função, já é possível avaliar a expressão `n_folio(xy(0, 0), 1, 2)` sem gerar qualquer erro e produzindo o bifólio correcto, visível na Figura 6.12.

Sendo possível gerar bifólios, trifólios, quadrifólios e n-fólios, é legítimo pensar também nos conceitos de *unifólio* e de *zerofólio* e tentar imaginar as suas formas. Infelizmente, aqui chocamos com limites matemáticos mais difíceis de ultrapassar: quando o número de fólios é um, o raio r_i do círculo interior fica negativo, perdendo todo o sentido geométrico; quando o número de fólios é zero, a situação é igualmente absurda, pois o ângulo α torna-se uma impossibilidade devido à divisão por zero. Por estes motivos, o bifólio é o limite inferior dos n-fólios.

6.4 Álgebra de Formas

Na secção anterior vimos a necessidade de introdução do conceito de forma vazia. Essa necessidade torna-se mais evidente quando comparamos as operações de combinação de formas com as operações algébricas, como a soma e o produto. Consideremos, para começar, uma função que some uma lista de números:

```
soma(numeros) =
  if numeros == []
    0
  else
    numeros[1]+soma(numeros[2:end])
  end
```

Como podemos ver na função anterior, a soma de uma lista *vazia* de números é zero. Isto faz sentido pois, se não há números para somar, o total é necessariamente zero. Por outro lado, quando pensamos que, durante a recursão ficam somas pendentes então, quando se atinge o caso de paragem, é necessário que seja devolvido um valor que não afecte as somas pendentes e, mais uma vez, faz sentido que esse valor tenha de ser zero pois esse é o elemento neutro da soma.

Do mesmo modo, se pensarmos numa função que calcule a união de uma lista de regiões, devemos escrever:

```
unioes(regioes) =
  if regioes == []
    empty_shape()
  else
    union(regioes[1], unioes(regioes[2:end]))
  end
```

pois `empty_shape()` é o elemento neutro da união de regiões.

Consideremos agora uma função que multiplique uma lista de números, onde deixámos por decidir qual será o valor da função no caso básico:

```
produto(numeros) =
  if numeros == []
    ???
  else
    numeros[1]*produto(numeros[2:end])
  end
```

Embora a maior parte das pessoas tenha a tentação de usar para valor do caso básico o número zero, isso seria incorrecto pois esse zero, por ser o elemento absorvente do produto, seria propagado ao longo da sequência de produtos que ficaram pendentes até ao caso básico, produzindo um resultado final de zero. Assim, torna-se claro que o valor correcto do caso básico tem de ser `1`, i.e., o elemento neutro do produto:

```
produto(numeros) =
  if numeros == []
    1
  else
    numeros[1]*produto(numeros[2:end])
  end
```

Esta análise é crucial para agora podermos definir correctamente uma função que calcule a intersecção de uma lista de regiões. Embora seja fácil escrever um primeiro esboço

```
interseccoes(regioes) =
  if regioes == []
    ???
  else
    intersection(regioes[1], interseccoes(regioes[2:end]))
  end
```

já não é tão evidente saber o que colocar no lugar de `???`. O que sabemos é que esse valor tem de ser o elemento neutro da operação de combinação que, neste caso, é a intersecção. Felizmente, não é difícil concluir que esse elemento neutro é, necessariamente, a *forma universal* U, i.e., a forma que inclui todos os pontos do espaço. Essa forma é produzida pela função `universal_shape`. Assim, já podemos definir:

```
interseccoes(regioes) =
  if regioes == []
    universal_shape()
  else
    intersection(regioes[1], interseccoes(regioes[2:end]))
  end
```

É importante percebermos que a necessidade das funções `empty_shape` e `universal_shape` resulta de termos de assegurar o correcto comportamento matemático das operações de união, intersecção e subtracção de regiões. Esse comportamento matemático é ditado pelas seguintes equações da álgebra de regioes:

$$R \cup \varnothing = \varnothing \cup R = R$$

$$R \cup U = U \cup R = U$$

$$R \cup R = R$$

$$R \cap \varnothing = \varnothing \cap R = \varnothing$$

$$R \cap U = U \cap R = R$$

$$R \cap R = R$$

$$R \setminus \varnothing = R$$

$$\varnothing \setminus R = \varnothing$$

$$R \setminus R = \varnothing$$

Exercício 6.4.1 A álgebra de regiões que elaborámos encontra-se incompleta por não incluir a operação de *complemento* de uma região. O complemento R^C de uma região R define-se como a subtracção à região universal U da região R:

$$R^C = U \setminus R$$

A operação de complemento permite representar o conceito de *buraco*. Um buraco com a forma da região R obtém-se simplesmente através R^C. Um buraco não tem representação geométrica óbvia, pois é difícil imaginar um buraco sem sabermos de que região é que ele é buraco, mas, do ponto de vista matemático, o conceito fará sentido se as operações algébricas sobre regiões o souberem interpretar. Por exemplo, dado o buraco R_1^C, podemos aplicá-lo a uma região R_0 através da intersecção das duas regiões $R_0 \cap R_1^C$. Como as ferramentas de CAD não sabem lidar com complementos de regiões, a operação terá de ser traduzida em termos de outras operações já conhecidas. Neste caso, a subtracção é uma escolha óbvia, pois $R_0 \cap R_1^C = R_0 \setminus R_1$.

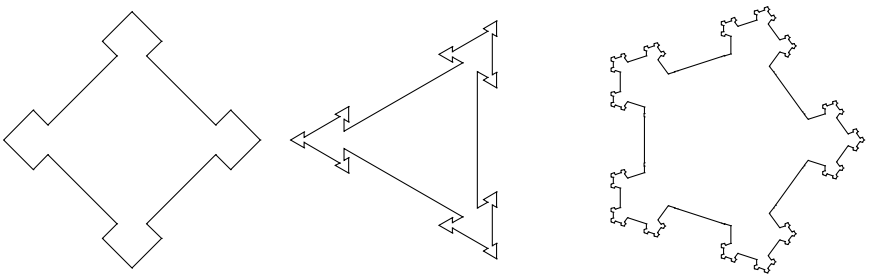
Para completar a álgebra de buracos é necessário definir o resultado das operações de união, intersecção e subtracção quando aplicadas a buracos. Defina matematicamente essas operações, bem como a combinação entre regiões "normais" e buracos.

Exercício 6.4.2 Uma vez que o Khepri não implementa o conceito de complemento, defina um construtor para complementos de regiões. O construtor deverá receber a região da qual se pretende o complemento e deverá devolver um objecto que represente simbolicamente o complemento da região. Não se esqueça que o complemento do complemento de uma região é a própria região.

Defina também um reconhecedor de complementos que poderá receber qualquer tipo de objecto e devolverá verdade apenas para aqueles que representam complementos de regiões.

Exercício 6.4.3 Defina as operações de união, intersecção e subtracção de regiões de modo a lidarem com complementos de regiões.

Exercício 6.4.4 Considere a construção de uma região composta por uma união recursiva de polígonos regulares sucessivamente mais pequenos e centrados nos vértices do polígono imediatamente maior, tal como se pode ver nos três exemplos apresentados na seguinte figura:



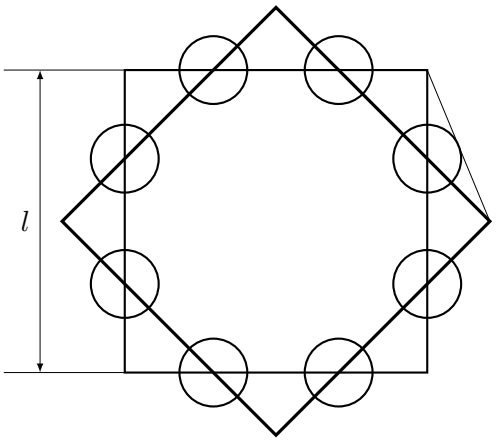
Tal como acontecia com a função `vertices_poligono_regular` definida na Secção 5.4.2, cada um destes polígonos é caracterizado por estar inscrito numa circunferência de raio r centrada num ponto p, por ter um "primeiro" vértice que faz um ângulo ϕ com o eixo X e por ter um determinado número de lados n. Para além disso, a construção dos polígonos

regulares é feita de modo a que cada vértice seja o centro de um novo polígono idêntico, mas inscrito num círculo cujo raio é uma fracção (dada por α_r) do raio r, sendo este processo repetido um determinado número de níveis. Por exemplo, na figura anterior, a imagem mais à esquerda foi feita com $p = (0, 0)$, $r = 1$, $\phi = 0$, $\alpha_r = 0.3$, $n = 4$ e com um número de níveis igual a 2. No conjunto, as imagens foram produzidas pela avaliação das seguintes expressões:

```
poligonos_recursivos(xy(0, 0), 1, 0, 0.3, 4, 2)
poligonos_recursivos(xy(3, 0), 1, pi/3, 0.3, 3, 3)
poligonos_recursivos(xy(6, 0), 1, 0, 0.3, 5, 4)
```

Exercício 6.4.5 As torres Petronas foram desenhadas pelo arquitecto Argentino César Pelli e foram consideradas os edifícios mais altos do mundo desde 1998 até 2004. A Figura 6.13 mostra uma perspectiva do topo de uma das torres.

A secção das torres, fortemente inspirada em motivos islâmicos, é composta por dois quadrados rodados de um quarto de círculo que se intersectam e ao qual se adicionam círculos nos pontos de intersecção, tal como se pode ver no seguinte esquema construtivo.

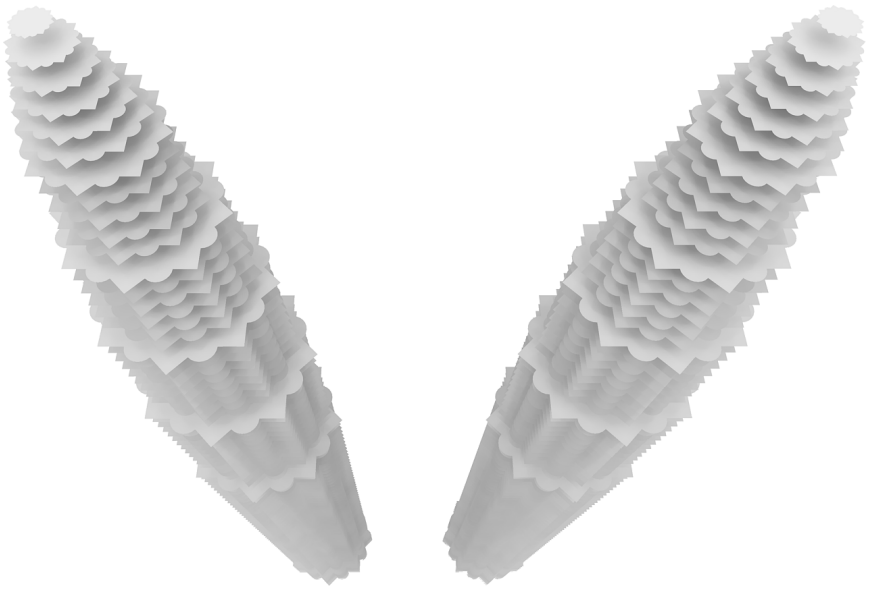


Note-se que os círculos são tangentes às arestas imaginárias que ligam os vértices.

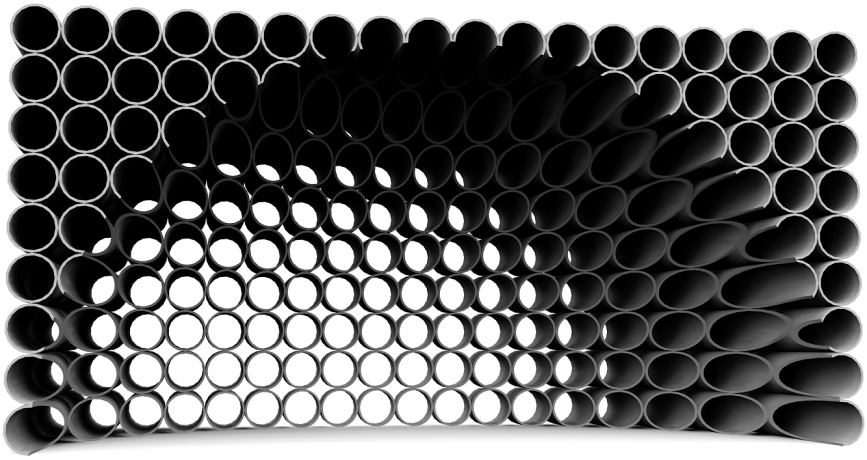
Defina uma função Julia denominada `seccao_petronas` que recebe o centro da secção e a largura l e produzem a secção da torre Petronas. Sugestão: use as funções `surface_regular_polygon` e `uniao_circulos` para gerar as regiões relevantes.

Exercício 6.4.6 Defina a função `torre_petronas` que, a partir do centro da base, constrói uma sucessão de secções da torre Petronas de modo a reproduzir a geometria real da torre Petronas, tal como se ilustra na seguinte perspectiva onde a função foi invocada duas vezes em posições diferentes:

Figura 6.13: As torres Petronas, localizadas em Kuala Lumpur, na Malasia. Fotografia de Mel Starrs.



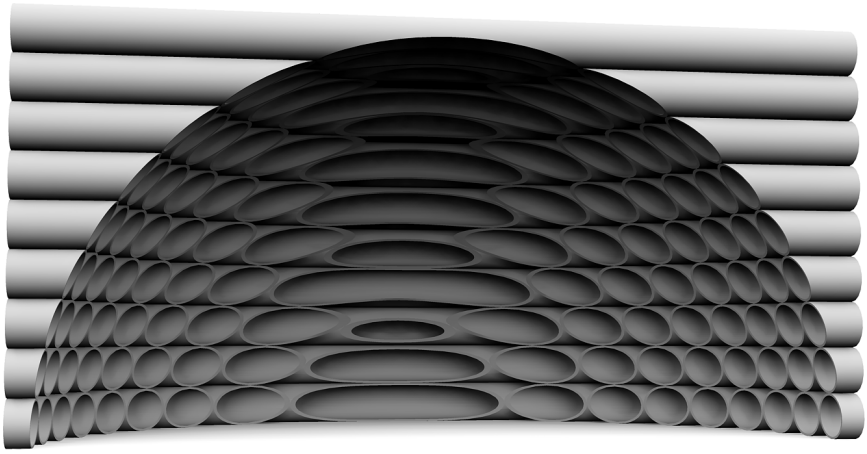
Exercício 6.4.7 Considere a imagem seguinte:



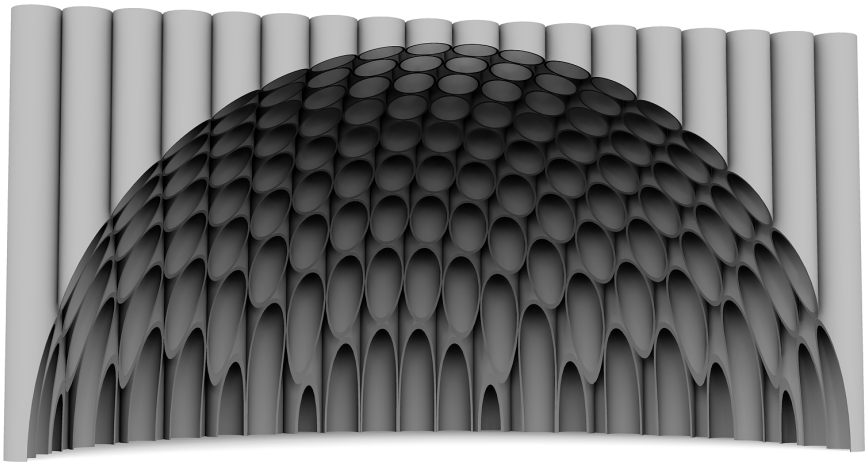
A imagem representa um abrigo de forma paralelipipédica construído com tubos cilíndricos cortados de modo a que o espaço interior tenha a forma de um quarto de esfera. Note que os tubos cilíndricos possuem uma espessura que é 10% do raio. Note ainda a relação entre o raio do quarto de esfera e o dos cilindros.

Defina uma função que constrói o abrigo a partir do centro da esfera, da altura do paralelipípedo e do número de tubos a colocar ao longo da altura.

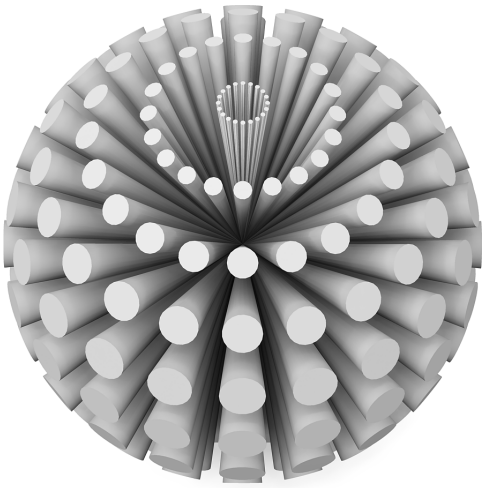
Exercício 6.4.8 Refaça o exercício anterior mas agora considerando a orientação dos tubos visivel na imagem seguinte:



Exercício 6.4.9 Refaça o exercício anterior mas agora considerando a orientação dos tubos visivel na imagem seguinte:



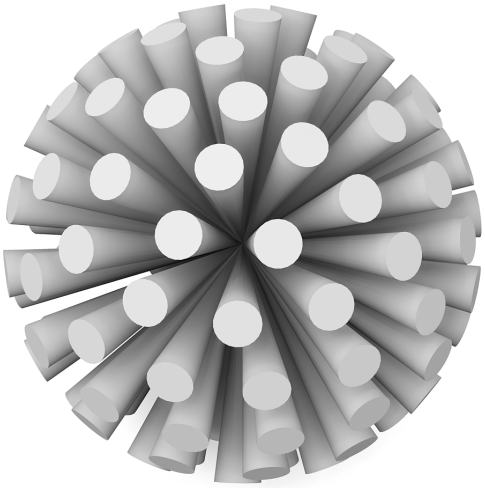
Exercício 6.4.10 Considere a construção de uma "esfera" de cones tal como se apresenta na seguinte imagem:



Note que todos os cones têm o seu vértice no mesmo ponto e estão orientados de modo a que o centro da base fica assente numa esfera virtual. Note que todos os "meridianos" e "paralelos" possuem o mesmo número de cones e note também que o raio da base dos cones diminui à medida que nos aproximamos dos "pólos" para que os cones não interfiram uns com os outros.

Escreva um programa em Julia capaz de construir a "esfera" de cones.

Exercício 6.4.11 Considere uma variação sobre a "esfera" de cones do exercício anterior em que, ao invés de diminuirmos o raio da base dos cones à medida que nos aproximamos dos pólos, diminuímos o número de cones, tal como se apresenta na seguinte imagem:



Escreva um programa em Julia capaz de construir esta "esfera" de cones.

Exercício 6.4.12 Defina uma função em Julia capaz de criar cascas esféricas perfuradas, tal como as que se apresentam em seguida:

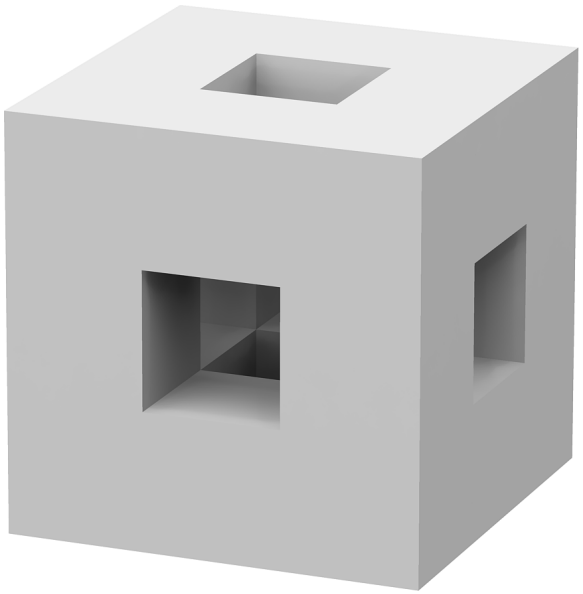
A função deverá ter como argumentos o centro, raio e espessura da casca esférica, e o raio e número de perfurações a realizar ao longo do "equador." Este número deverá diminuir à medida que nos aproximamos dos "pólos."

Exercício 6.4.13 Considere a seguinte construção feita a partir de um arranjo aleatório de esferas perfuradas:

Para modelar a imagem anterior, defina uma função que recebe dois pontos que definem os limites de um volume imaginário (paralelo aos eixos coordenados) dentro do qual estão localizados os centros das esferas e ainda o valor mínimo e máximo do raio de cada esfera, a espessura da casca, o número de esferas a criar e, finalmente, os parâmetros necessários para realizar o mesmo tipo de perfurações em cada esfera.

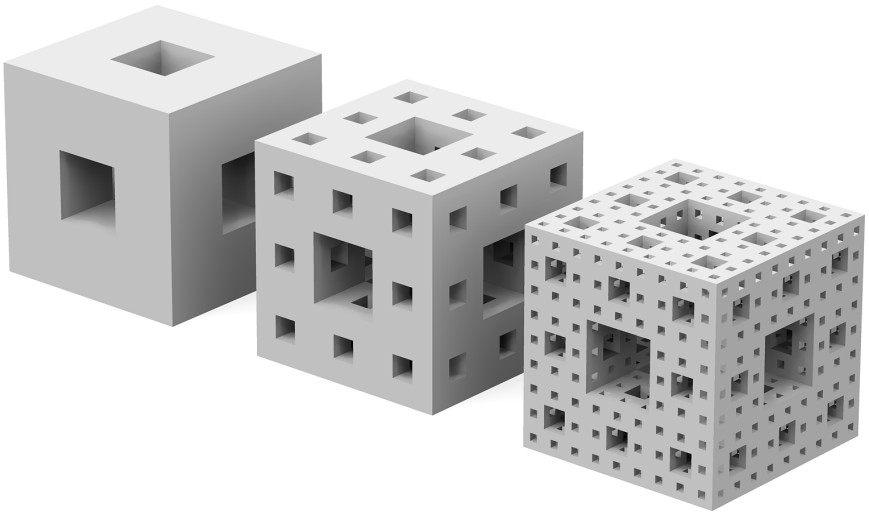
Note que o interior da construção deverá estar desimpedido, i.e., tal como se ilustra no seguinte corte:

Exercício 6.4.14 Considere um cubo construído a partir de um arranjo de cubos mais pequenos, tal como se apresenta na imagem seguinte:



Como é fácil de ver, os cubos pequenos possuem, cada um, um terço do lado do cubo construído. Defina uma função que, a partir das coordenadas de um vértice do cubo e do comprimento do lado do cubo a construir, constrói este cubo respeitando o arranjo de cubos pequenos que aparece na imagem anterior.

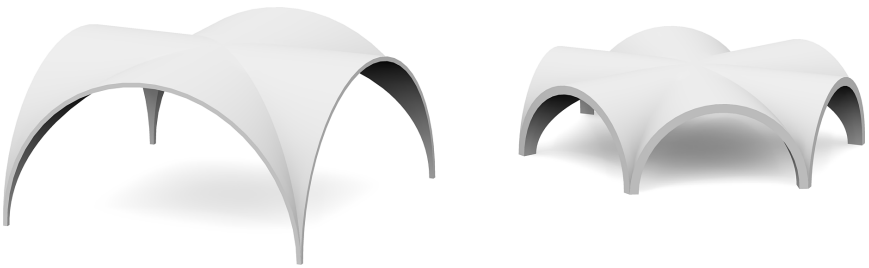
Exercício 6.4.15 A *esponja* de Menger é um conhecido fractal descoberto pelo matemático Karl Menger. A seguinte imagem mostra vários estágios da construção de uma esponja de Menger.



À semelhança do exercício anterior, a construção de uma esponja de Menger pode ser feita através da composição de cubos mais pequenos, com a *nuance* de se substituirem os cubos pequenos por (sub-)esponjas de Menger. Naturalmente, no caso de uma implementação em computador, esta recursão infinita é impraticável e, por isso, há que estabelecer uma condição de paragem, a partir da qual não se cria uma (sub-)esponja de Menger, mas sim um cubo.

Defina a função `esponja_menger` que recebe as coordenadas de um vértice da esponja, a dimensão da esponja e o grau de profundidade desejado para a recursão.

Exercício 6.4.16 Considere a seguinte cobertura feita de abóbadas em arco romano:



Defina uma função denominada `cobertura_arcos_romanos` que tem como parâmetros o centro da cobertura, o raio do círculo que circunscreve a cobertura, a espessura da cobertura e o número de arcos a colocar e que produz coberturas como as apresentadas acima. Assegure-se que a sua função consegue gerar a seguinte cobertura de apenas três arcos:

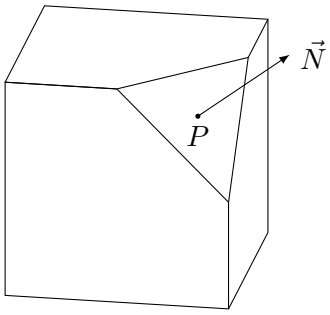
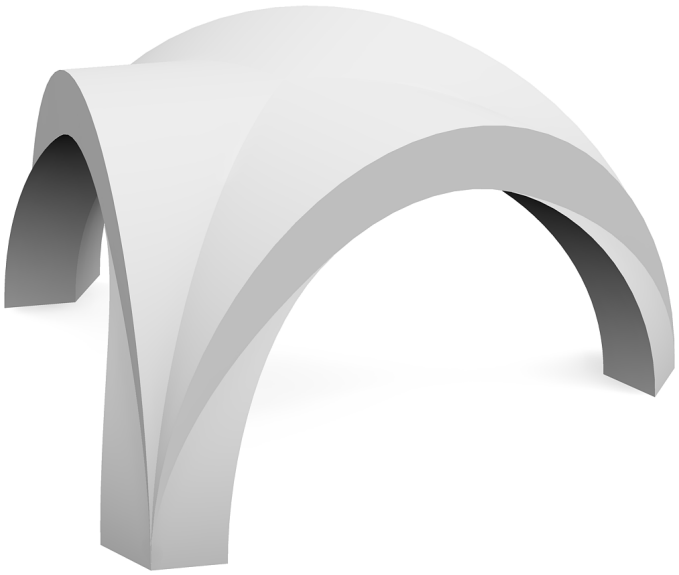


Figura 6.14: Corte de um sólido por intermédio de um plano de corte definido por um ponto P pertencente ao plano e por um vector N normal ao plano.



6.5 Corte de Regiões

Para além das operações de união, intersecção e subtracção, o Khepri disponibiliza a operação de *corte*, implementada pela função `slice`. Esta operação permite modificar uma região através do seu corte por um plano. A especificação do plano é feita através de um ponto contido nesse plano e um vector normal ao plano. A direcção do vector indica qual a região que se pretende descartar. A Figura 6.14 ilustra o corte de um cubo por um plano (virtual) definido pelo ponto P e pelo vector $\vec{N}$. A sintaxe desta operação é `slice(`*`região`*`, P, N)`.

Como exemplo de utilização desta operação consideremos a criação de um "gomo" (de ângulo de abertura ϕ) de uma esfera de raio r centrada em

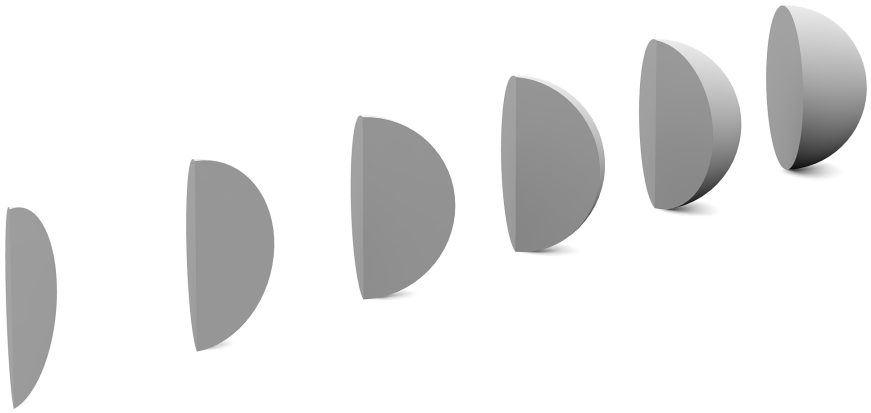


Figura 6.15: Gomos de uma esfera. Da direita para a esquerda, o ângulo entre os planos de corte varia desde 0 até π em incrementos de $\pi/6$.

p. O gomo é obtido através de dois cortes verticais na esfera.

```
gomo_esfera(p, r, fi) =
  slice(slice(sphere(p, r),
              p,
              vcyl(1, 0, 0)),
        p,
        vcyl(1, fi, 0))
```

A Figura 6.15 apresenta gomos com diferentes aberturas gerados pela função `gomo_esfera`.

Frequentemente, os planos de corte que pretenderemos empregar terão normais paralelas aos eixos coordenados. Para facilitar esses casos, vamos definir funções apropriadas:

```
u0() =
  xyz(0, 0, 0)

vux() =
  vxyz(1, 0, 0)

vuy() =
  vxyz(0, 1, 0)

vuz() =
  vxyz(0, 0, 1)
```

Usando estas funções, podemos facilmente construir sólidos com formas complexas. Por exemplo, um oitavo de uma esfera obtém-se através de três cortes:

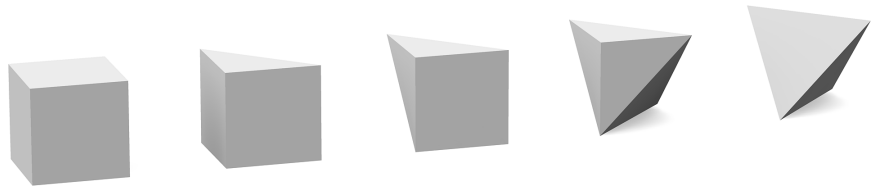


Figura 6.16: A criação de um tetraedro por sucessivos cortes num paralelilípedo envolvente.

```
slice(slice(slice(sphere(u0(), 2),
                  u0(), vux()),
            u0(), vuy()),
      u0(), vuz())
```

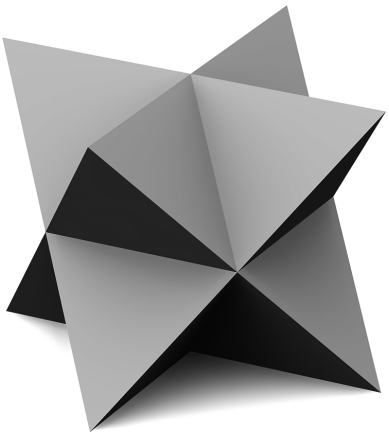
Para terminarmos, consideremos a modelação de um tetraedro. O tetraedro é o mais simples dos sólidos platónicos e caracteriza-se por ser um poliedro de quatro lados triangulares. Esses quatro lados unem-se em quatro vértices que especificam totalmente o tetraedro. Embora o Khepri disponibilize várias operações para modelar alguns dos sólidos fundamentais, como o paralelipípedo ou a pirâmide regular, não disponibiliza uma operação específica para criar tetraedros. Para implementar esta operação, podemos produzir o tetraedro através de cortes num paralelipípedo que envolva totalmente os quatro vértices do tetraedro, tal como apresentamos na Figura 6.16.

Para especificarmos o paralelipípedo envolvente, basta-nos calcular os valores máximo e mínimo das coordenadas dos vértices do tetraedro. Em seguida, cortamos o paralelipípedo usando os quatro planos correspondentes às faces do tetraedro, planos esses definidos pelas combinações dos quatro vértices tomados três a três. Assim, admitindo que o tetraedro tem os vértices P_0, P_1, P_2 e P_3, o primeiro corte será definido pelo plano contendo os pontos P_0, P_1 e P_2 e preservando a parte que contém P_3, o segundo pelos pontos P_1, P_2 e P_3 e preservando P_0, o terceiro pelos pontos P_2, P_3 e P_0 e preservando P_1, e o quarto pelos pontos P_3, P_0 e P_1 e preservando P_2. Cada um destes planos será especificado na operação de corte por um ponto e pelo vector normal ao plano. Para o cálculo deste vector normal temos de empregar o produto externo de dois vectores (implementado pela função pré-definida `cross`) e temos de verificar se a normal aponta na direcção do ponto a preservar, o que podemos fazer verificando o sinal do produto interno (implementado pela função pré-definida `dot`) entre o vector normal e o vector que termina no ponto a preservar.

```
normal_pontos(p0, p1, p2, p3) =
  let v0 = p1-p0,
      v1 = p2-p0,
      n = cross(v0, v1)
    if dot(n, p3-p0) < 0
      n
    else
      -n
    end
  end


tetraedro(p0, p1, p2, p3) =
  let pmin = xyz(min(p0.x, p1.x, p2.x, p3.x),
                 min(p0.y, p1.y, p2.y, p3.y),
                 min(p0.z, p1.z, p2.z, p3.z)),
      pmax = xyz(max(p0.x, p1.x, p2.x, p3.x),
                 max(p0.y, p1.y, p2.y, p3.y),
                 max(p0.z, p1.z, p2.z, p3.z)),
      solido = box(pmin, pmax)
    solido = slice(solido, p0, normal_pontos(p0, p1, p2, p3))
    solido = slice(solido, p1, normal_pontos(p1, p2, p3, p0))
    solido = slice(solido, p2, normal_pontos(p2, p3, p0, p1))
    solido = slice(solido, p3, normal_pontos(p3, p0, p1, p2))
    solido
  end
```

A Figura 6.16 apresenta as várias fases do processo de "lapidação" do paralelipípedo até se obter o tetraedro.

Exercício 6.5.1 Em 1609, Johannes Kepler[1] concebeu o poliedro que se ilustra na imagem seguinte, que baptizou de *stella octangula*. Esta estrela de oito braços, também conhecida por octaedro estrelado é, na verdade, uma composição de dois tetraedros.



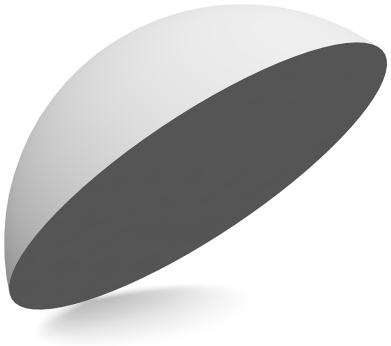
[1]Johannes Kepler foi um famoso matemático e astrónomo que, entre muitas outras contribuições, estableceu as leis do movimento planetário.

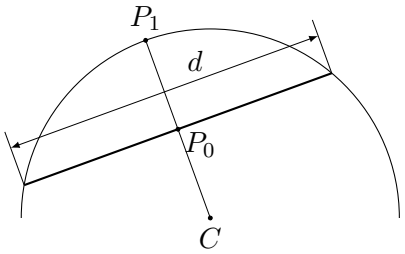
Defina a função `octaedro_estrelado` que, dado o centro do cubo envolvente ao octaedro e o comprimento da aresta desse cubo, produz o octaedro imaginado por Kepler.

Exercício 6.5.2 A imagem seguinte representa sucessivas iterações da pirâmide de Sierpiński,[2] também denominada *tetrix*. A pirâmide de Sierpiński é um fractal tridimensional que se pode produzir de forma recursiva a partir de um tetraedro imaginário cujos pontos médios de cada aresta constituem os vértices de sub-pirâmides de Sierpiński.

Defina a função `sierpinski` que, a partir das coordenadas dos quatro vértices e do nível de recursão pretendido, cria a pirâmide de Sierpiński correspondente.

Exercício 6.5.3 Considere a calota esférica que se apresenta em seguida, caracterizada pelos dois pontos P_0 e P_1 e pelo diâmetro d da base da calota.





Defina a função `calota_esferica` que recebe os pontos P_0 e P_1 e o diâmetro d da base da calota. Sugestão: determine a posição e dimensão da esfera e empregue um plano de corte apropriado.

6.6 Extrusões

Vimos nas secções anteriores que o Khepri disponibiliza um conjunto de sólidos pré-definidos tais como esferas, paralelipípedos, pirâmides, etc. Através da composição desses sólidos é possível modelar formas bastante mais

[2] Wacław Sierpiński foi um matemático polaco que deu enormes contribuições à teoria dos conjuntos e à topologia. Sierpiński descreveu uma versão bidimensional desta pirâmida em 1915.

Figura 6.17: Um detalhe do Kunst- und Ausstellungshalle. Fotografia de Hanneke Kardol.

complexas mas, em qualquer caso, essas formas serão sempre decomponíveis nas formas básicas que estão na sua génese.

Infelizmente, muitas das formas que a nossa imaginação consegue conceber não são fáceis de construir apenas através da composição dos sólidos pré-definidos. A título de exemplo, consideremos o Kunst- und Ausstellungshalle, um edifício desenhado pelo arquitecto vienense Gustav Peichl e destinado a ser um centro de exposições e de comunicações. Este edifício é de forma quadrada mas um dos "cantos" deste quadrado é cortado por uma sinusóide, representada na Figura 6.17.

Como é óbvio, se pretendermos modelar a parede ondulada do Kunst- und Ausstellungshalle não haverá nenhum sólido pré-definido que possamos usar como ponto de partida para a sua construção.

Felizmente, o Khepri fornece um conjunto de funcionalidades que permite resolver facilmente alguns destes problemas de modelação. Nesta secção vamos considerar duas dessas funcionalidades, em particular a extrusão simples e a extrusão ao longo de um caminho.

6.6.1 Extrusão Simples

Mecanicamente falando, a extrusão é um processo de fabrico que consiste em fazer passar uma quantidade de material moldável através de uma matriz com a forma desejada de modo a produzir peças cuja secção é determi-

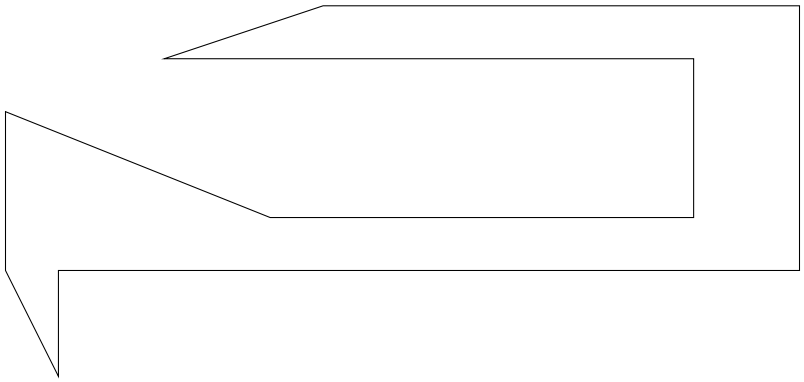


Figura 6.18: Um polígono arbitrário.

nada por essa matriz.

No Khepri, a extrusão é uma operação metaforicamente idêntica à que é usada em fabrico mecânico, embora se parta apenas da matriz que é "estendida" na direcção desejada, produzindo assim uma forma geométrica cuja secção é igual à matriz. A extrusão é realizada pela função `extrusion` que recebe, como argumentos, a secção que se pretende extrudir e a direcção da extrusão pretendida ou a altura se essa direcção for a vertical. Se a região a extrudir for uma curva, a extrusão produz necessariamente uma superfície. Se a região a extrudir for uma superfície, a extrusão produz um sólido. Convém ter presente que existem limitações no processo de extrusão que impedem a extrusão de formas excessivamente complexas.

A título de exemplo, consideremos a Figura 6.18 onde se mostra um polígono arbitrário assente no plano XY. Para construirmos um sólido de altura unitária a partir da extrusão desse polígono na direcção Z podemos usar a seguinte expressão:

```
extrusion(surface_polygon(xy(0, 2), xy(0, 5), xy(5, 3), xy(13, 3),
                          xy(13, 6), xy(3, 6), xy(6, 7), xy(15, 7),
                          xy(15, 2), xy(1, 2), xy(1, 0)),
          vz(1))
```

A forma resultante está representada à esquerda, na Figura 6.19

Na mesma figura, à direita, está representada uma extrusão alternativa, que no lugar de produzir sólidos produz superfícies. Esta extrusão pode ser obtida substituindo a expressão anterior por:

```
extrusion(polygon(xy(0, 2), xy(0, 5), xy(5, 3), xy(13, 3),
                  xy(13, 6), xy(3, 6), xy(6, 7), xy(15, 7),
                  xy(15, 2), xy(1, 2), xy(1, 0)),
          vz(1))
```

Finalmente, a Figura 6.20 ilustra duas "flores," a da esquerda produzida por um conjunto de cilindros com a base num mesmo ponto e os topos posicionados ao longo da superfície de uma esfera, a da direita produzida por

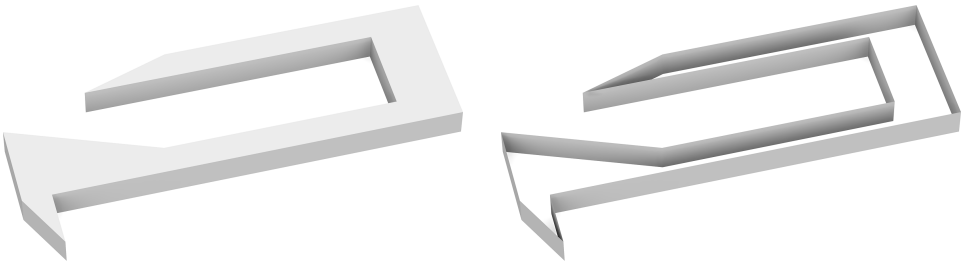


Figura 6.19: Um sólido (à esquerda) e uma superfície (à direita) produzidas pela extrusão de um polígono.

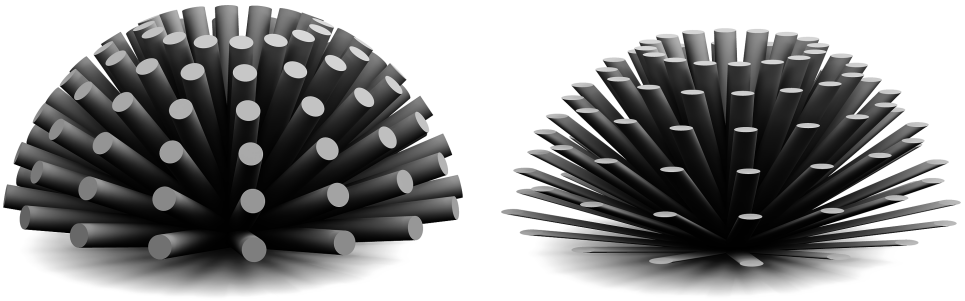


Figura 6.20: Uma "flor" (à esquerda) produzida por um conjunto de cilindros inclinados e (à direita) produzida por um conjunto de extrusões inclinadas.

um conjunto de extrusões a partir de uma mesma superfície circular horizontal, usando os vectores correspondentes aos topos dos cilindros anteriores. Como podemos ver, as extrusões da superfície circular em direcções que não são perpendiculares a essa superfície produz cilindros distorcidos.

Naturalmente, é possível extrudir formas planas mais complexas, o que permite modelar, por exemplo, a fachada ilustrada na Figura 6.21, pertencente a um edifício em Tucson, Arizona, onde é evidente a forma sinusoidal. Para modelarmos essa fachada, podemos "traçar" duas sinusóides paralelas no plano XY, juntamos-lhes dois segmentos de recta nas extremidades para as transformarmos numa superfície e, de seguida, extrudimo-las até à altura pretendida.

Matematicamente, a equação da sinusóide é:

$$y(x) = a\,\mathrm{sin}(\omega x + \phi)$$

em que a é a *amplitude* da sinusóide, ω é a *frequência angular*, i.e., o número de ciclos por comprimento e ϕ é a *fase*, i.e., o "avanço" ou "atraso" da curva em relação ao eixo dos Y. A Figura 6.22 mostra a curva sinusóide, ilustrando o significado destes parâmetros.

A tradução da definição anterior para Julia é a seguinte:

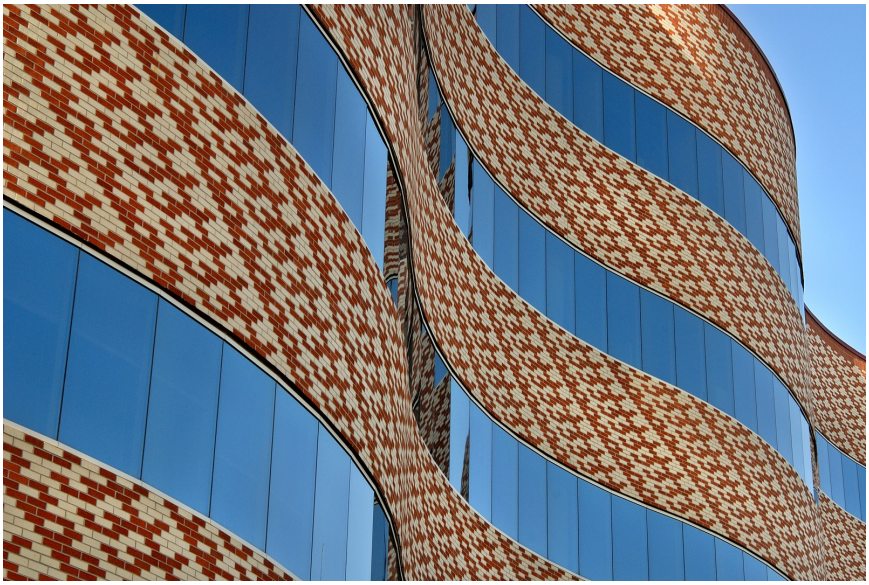


Figura 6.21: Fachada sinusoidal de um edifício em Tucson, Arizona. Fotografia de Janet Little.

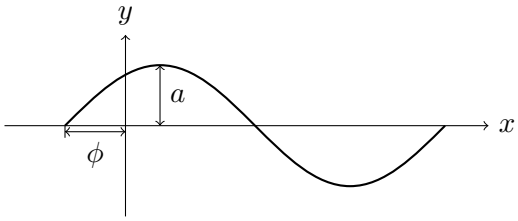


Figura 6.22: A curva sinusóide.

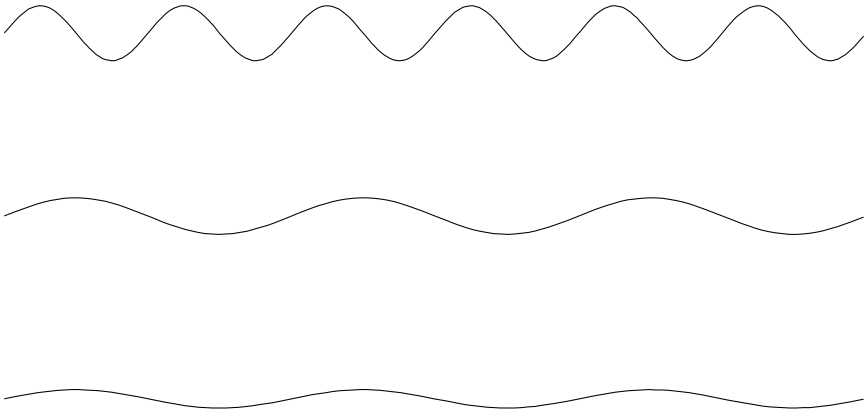


Figura 6.23: Três curvas sinusóides.

```
sinusoide(a, omega, fi, x) =
  a*sin(omega*x+fi)
```

A partir da função sinusóide, o passo seguinte para a modelação de curvas sinusóides é a definição de uma função que produza uma lista de coordenadas correspondentes a pontos da curva sinusóide entre os limites de um intervalo $[x_0, x_1]$ e separados de um incremento Δ_x. Uma vez que podemos estar interessados em produzir curvas sinusóides cuja "origem" esteja localizada num determinado ponto do espaço p, é conveniente que a função que constrói os pontos da curva sinusóide inclua também esse ponto como parâmetro:

```
pontos_sinusoide(p, a, omega, fi, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p+vy(sinusoide(a, omega, fi, x0)),
     pontos_sinusoide(p+vx(dx), a, omega, fi, x0+dx, x1, dx)...]
  end
```

Note-se que os pontos gerados estão todos assentes num plano paralelo ao plano XY, com a sinusóide a evoluir numa direcção paralela ao eixo X, a partir de um ponto p. A Figura 6.23 mostra três dessas curvas obtidas a partir das seguintes expressões:

```
spline(pontos_sinusoide(xy(0, 0), 0.2, 1, 0, 0, 6*pi, 0.4))
spline(pontos_sinusoide(xy(0, 4), 0.4, 1, 0, 0, 6*pi, 0.4))
spline(pontos_sinusoide(xy(0, 8), 0.6, 2, 0, 0, 6*pi, 0.2))
```

Como se pode ver pela análise da figura, as sinusóides possuem diferentes amplitudes e diferentes períodos.

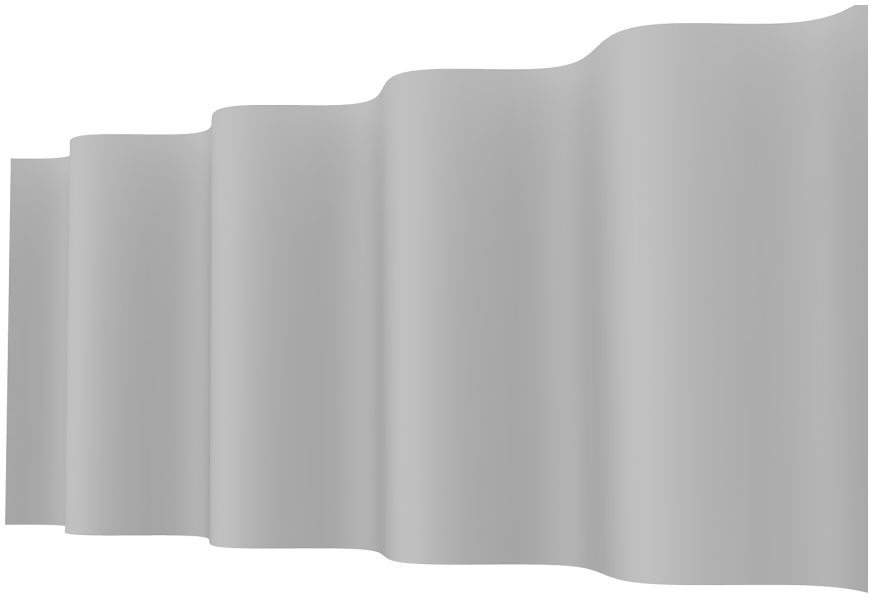


Figura 6.24: Uma parede sinusoidal produzida por extrusão simples a partir de uma região formada por duas sinusóides unidas nas pontas de forma a fazerem uma curva fechada.

Para construir uma parede sinusoidal basta-nos agora criar duas sinusóides afastadas de uma determinada distância igual à espessura e da parede, curvas essas que iremos fechar para formar uma região que, depois, extrudimos para ficar com uma dada altura h:

```
parede_sinusoidal(p, a, omega, fi, x0, x1, dx, e, h) =
  let pts_1 = pontos_sinusoide(p, a, omega, fi, x0, x1, dx),
      pts_2 = pontos_sinusoide(p+vy(e), a, omega, fi, x0, x1, dx)
    extrusion(surface(spline(pts_1),
                      spline(pts_2),
                      spline([pts_1[1], pts_2[1]]),
                      spline([pts_1[end], pts_2[end]])),
              vz(h))
  end
```

Usando a função `pontos_sinusoide` torna-se possível construir uma parede de forma sinusoidal, tal como ilustramos na Figura 6.24.

Exercício 6.6.1 Considere a extrusão de uma curva fechada aleatória tal como exemplificamos na seguinte figura:

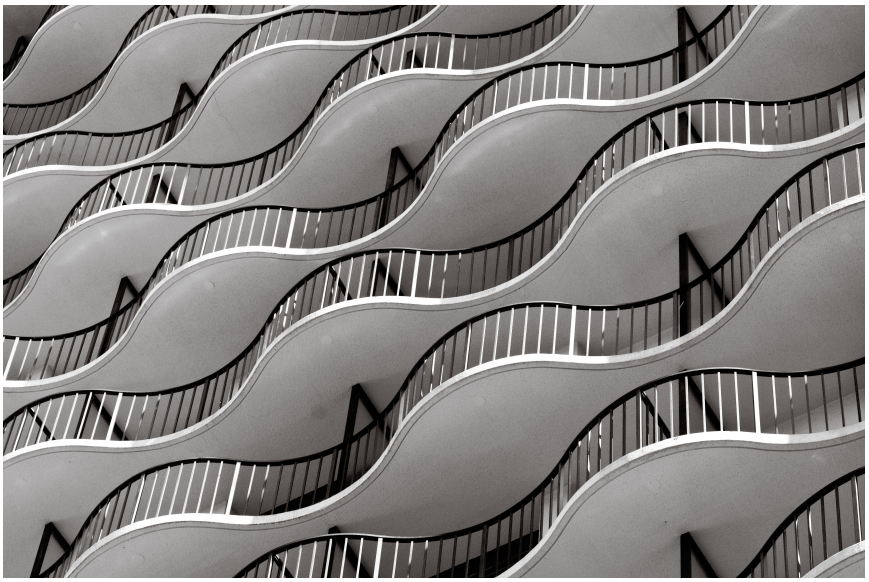
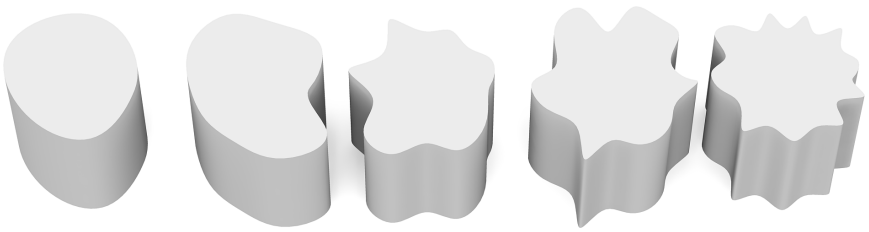


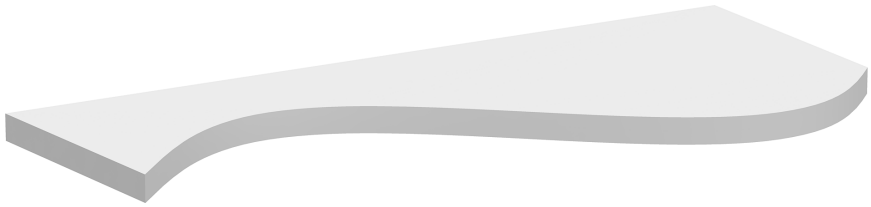
Figura 6.25: Pormenor do Hotel Marriott de Anaheim. Fotografia de Abbie Brown.



Cada uma das curvas é uma *spline* produzida a partir de pontos gerados pela função `pontos_circulo_raio_aleatorio` definida no exercício 5.5.2. Escreva uma expressão capaz de gerar um sólido semelhante aos apresentados na imagem anterior.

Exercício 6.6.2 A Figura 6.25 apresenta um detalhe da fachada do Hotel Marriott em Anaheim. É fácil vermos que as varandas da fachada do hotel são sinusóides de igual amplitude e frequência mas que, entre cada dois pisos, as sinusóides apresentam fases opostas. Para além disso, cada varanda corresponde a uma sinusóide localizada a uma determina cota z.

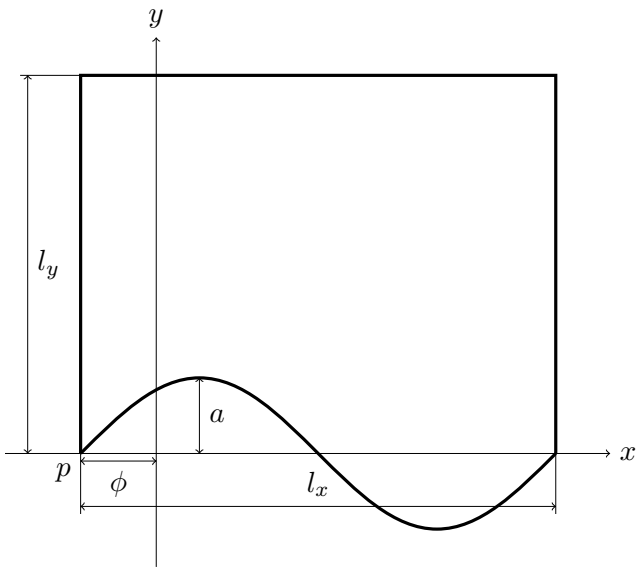
Crie uma função denominada `laje` que, recebendo todos os parâmetros necessários para a geração de uma sinusóide, cria uma laje rectangular mas com a fachada de forma sinusoidal, tal como se apresenta na seguinte imagem:



A função deverá ter os seguinte parâmetros:

```
laje(p, a, omega, fi, lx, dx, ly, lz) =
  ...
```

que estão explicados na seguinte figura:



O parâmetro `dx` representa o incremento Δ_x a considerar para a geração de pontos da sinusóide. O último parâmetro `lz` representa a espessura da laje.

A título de exemplo, considere a laje representada anteriormente como tendo sido gerada pela invocação

```
laje(xy(0, 0), 1.0, 1.0, 0, 4*pi, 0.5, 2, 0.2)
```

Exercício 6.6.3 Crie uma função denominada `corrimao` que, recebendo todos os parâmetros necessários para a geração de uma sinusóide, cria um corrimão de secção rectangular, mas ao longo de uma curva sinusoidal, tal como se apresenta na seguinte imagem:

A função deverá ter os seguinte parâmetros:

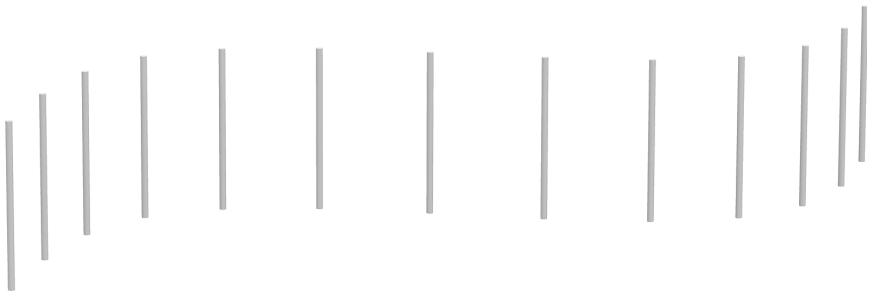
```
corrimao(p, a, omega, fi, lx, dx, l_corrimao, a_corrimao) =
  ...
```

Os primeiros parâmetros são os necessários para completamente especificar a curva sinusóide. Os dois últimos `l_corrimao` e `a_corrimao` correspondem à largura e altura da secção rectangular do corrimão.

Por exemplo, a imagem anterior poderia ser gerada pela invocação

```
corrimao(xy(0, 0), 1.0, 1.0, 0, 4*pi, 0.5, 0.5, 0.2)
```

Exercício 6.6.4 Crie uma função denominada `prumos` que, recebendo todos os parâmetros necessários para a geração de uma sinusóide, cria os prumos de apoio a um corrimão de curva sinusoidal, tal como se apresenta na seguinte imagem:



Note que os prumos são de secção circular e, consequentemente, correspondem a cilindros com uma determinada altura e um determinado raio. Note também que os prumos possuem um determinado afastamento horizontal `dx`.

A função deverá ter os seguinte parâmetros:

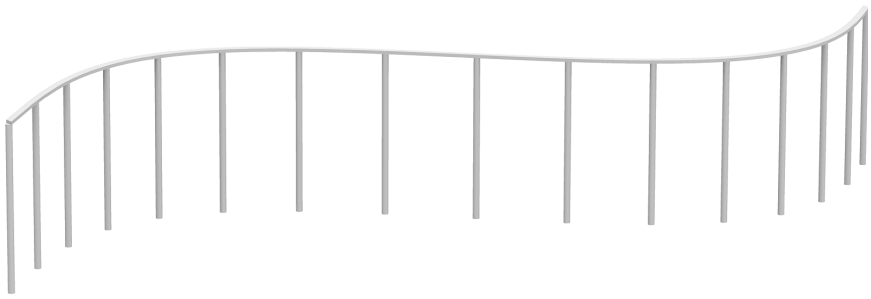
```
prumos(p, a, omega, fi, lx, dx, altura, raio) =
  ...
```

Os primeiros parâmetros são os necessários para completamente especificar a curva sinusóide. Os dois últimos `altura` e `raio` correspondem à altura e raio de cada cilindro.

Por exemplo, a imagem anterior poderia ser gerada pela invocação

```
prumos(xy(0, 0), 1.0, 1.0, 0, 4*pi, 0.8, 1, 0.1)
```

Exercício 6.6.5 Crie uma função denominada `guarda` que, recebendo todos os parâmetros necessários para a criação do corrimão e dos prumos, cria uma guarda, tal como se apresenta na imagem seguinte:



Para simplificar, considere que os prumos deverão ter como diâmetro exactamente a mesma largura que o corrimão.

A função deverá ter os seguinte parâmetros:

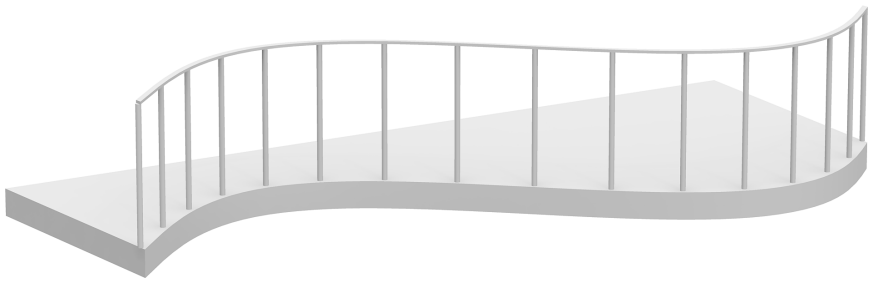
```
guarda(p, a, omega, fi, lx, dx,
       a_guarda, l_corrimao, a_corrimao, d_prumos) =
  ...
```

Os primeiros parâmetros são os necessários para completamente especificar a curva sinusóide. Os parâmetros `a_guarda`, `l_corrimao`, `a_corrimao` e `d_prumos` são, respectivamente, a altura da guarda, a largura e altura da secção quadrada do corrimão e a separação horizontal entre os prumos.

Por exemplo, a imagem anterior poderia ser gerada pela invocação

```
guarda(xy(0, 0), 1.0, 1.0, 0, 4*pi, 0.5, 1, 0.1, 0.04, 0.4)
```

Exercício 6.6.6 Crie uma função denominada `piso` que, recebendo todos os parâmetros necessários para a criação da laje e da guarda, cria um piso, tal como se apresenta na imagem seguinte:



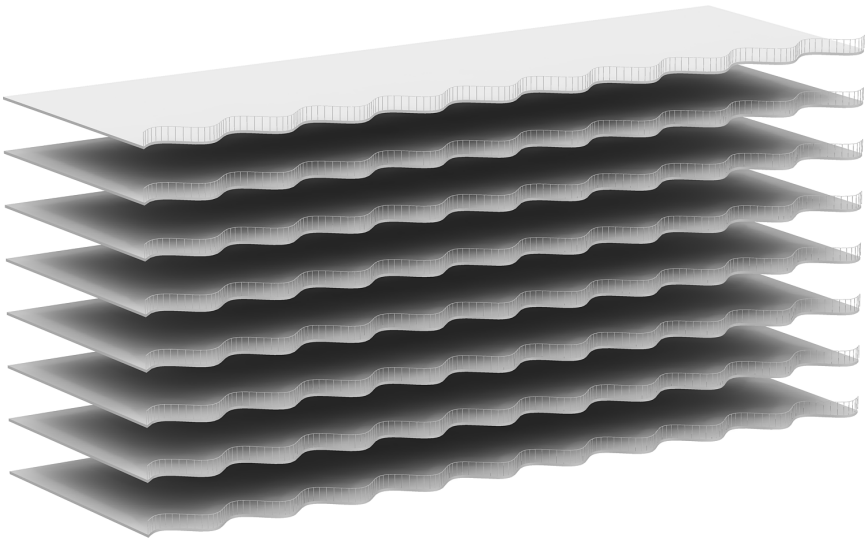
Para simplificar, considere que a guarda fica assente na extremidade da laje. A função `piso` deverá ter os seguinte parâmetros:

```
piso(p, a, omega, fi, lx, dx, ly,
     a_laje, a_guarda, l_corrimao, a_corrimao, d_prumos) =
  ...
```

Por exemplo, a imagem anterior poderia ser gerada pela invocação:

```
piso(xy(0, 0), 1.0, 1.0, 0, 2*pi, 0.5, 2, 0.2, 1, 0.04, 0.02, 0.4)
```

Exercício 6.6.7 Crie uma função denominada `predio` que recebe todos os parâmetros necessários para a criação de um andar, incluindo a altura de cada andar `a_andar` e o número de andares `n_andares`, e cria um prédio com esses andares, tal como se apresenta na imagem seguinte:



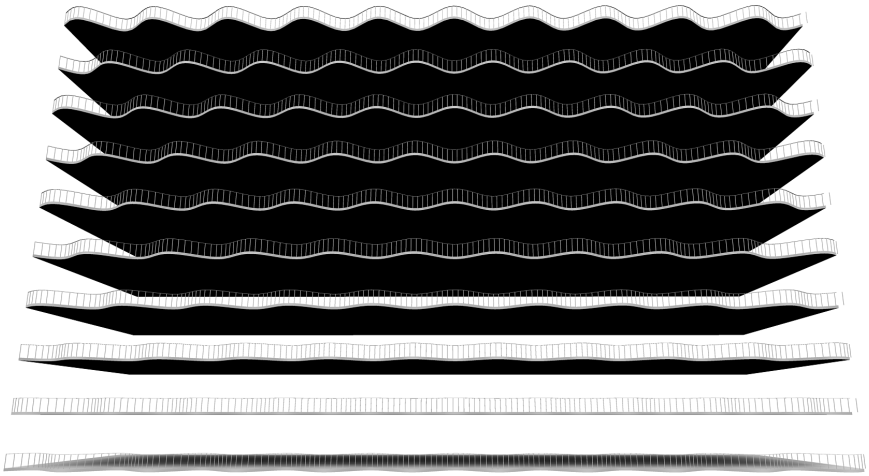
A função `predio` deverá ter os seguinte parâmetros:

```
predio(p, a, omega, fi, lx, dx, ly,
       a_laje, a_guarda,
       l_corrimao, a_corrimao,
       d_prumos, a_andar, n_andares) =
  ...
```

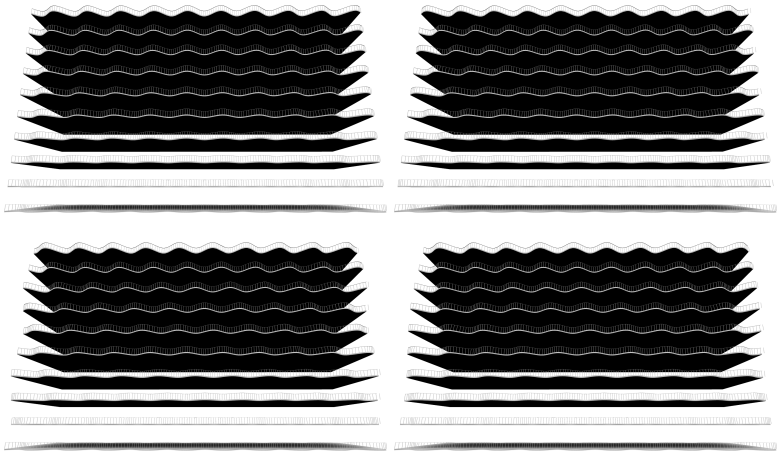
Por exemplo, a imagem anterior poderia ser gerada pela invocação:

```
predio(xy(0, 0), 1.0, 1.0, 0, 20*pi, 0.5, 20, 0.2, 1, 0.04, 0.02, 0.4, 4, 8)
```

Exercício 6.6.8 Modifique a função `predio` de modo a receber um parâmetro adicional que representa um incremento de fase a considerar em cada andar. Através desse parâmetro é possível gerar edifícios com fachadas mais interessantes, em que cada piso tem uma forma sinusoidal diferente. Por exemplo, compare o seguinte edifício com o anterior:



Exercício 6.6.9 As imagens seguintes apresentam a visualização de algumas variações em torno do incremento de fase. Identifique qual foi o incremento empregue em cada imagem.



6.6.2 Extrusão ao Longo de um Caminho

As extrusões realizadas nos exemplos anteriores correspondem a criar sólidos (ou superfícies) através da deslocação de uma secção assente no plano XY ao longo de uma direcção. Através da função `sweep`, é possível generalizar o processo de modo a que essa deslocação seja feita ao longo de uma curva arbitrária. Tal como acontecia com a extrusão simples, convém ter presente que existem algumas limitações no processo de extrusão que impedem a extrusão de formas excessivamente complexas ou ao longo de caminhos excessivamente complexos.

A título de exemplo consideremos a criação de um tubo cilíndrico de forma sinusoidal. É relativamente evidente que a secção a extrudir é um círculo, mas é também óbvio que a extrusão não pode ser feita segundo

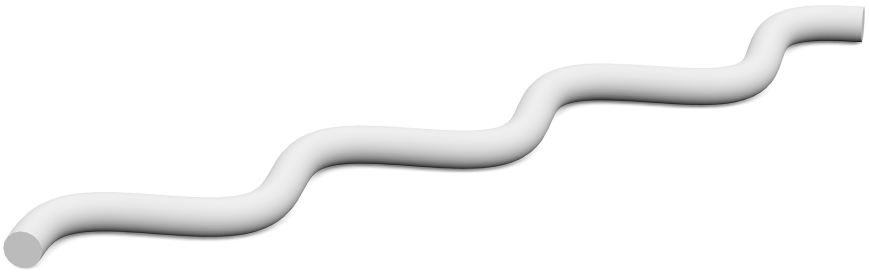


Figura 6.26: Um tubo cilíndrico de forma sinusoidal.

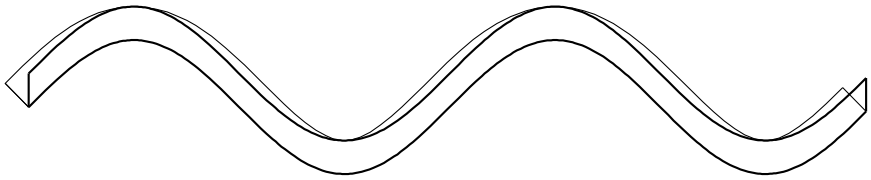


Figura 6.27: Vista de topo de duas paredes sinusoidais sobrepostas. O perfil a traço mais grosso foi gerado a partir da extrusão de uma região delimitada por duas sinusóides. O perfil a traço mais fino foi gerado pela extrusão de uma região rectangular ao longo de uma sinusóide.

uma direcção fixa pois o resultado não teria a forma sinusoidal pretendida. Para resolvermos este problema, ao invés de extrudirmos o círculo segundo uma recta, podemos simplesmente fazer deslocar esse círculo ao longo de uma sinusóide, obtendo a imagem da Figura 6.26. Esta imagem foi produzida pelas expressões:

```
let seccao = surface_circle(xy(0, 0), 0.4),
    curva = spline(pontos_sinusoide(xy(0, 0), 0.2, 1, 0, 0, 7*pi, 0.2))
  sweep(curva, seccao)
end
```

Quer a extrusão ao longo de um caminho (`sweep`), quer a extrusão simples (`extrusion`), permitem construir inúmeros sólidos, mas é importante saber qual das operações é mais adequada em cada caso. A título de exemplo, consideremos novamente a criação de uma parede sinusoidal. Na secção 6.6.1 idealizámos um processo de o fazer a partir de uma região delimitada por duas sinusóides, que extrudimos na direcção vertical, para formarmos uma parede sólida. Infelizmente, quando as sinusóides usadas como ponto de partida possuem curvaturas acentuadas, as paredes resultantes ficam com uma espessura manifestamente não-uniforme, tal como a imagem apresentada na Figura 6.27 o demonstra.

Para darmos à parede uma espessura uniforme, é preferível pensar na modelação da parede como um rectângulo, com a altura e a espessura da parede, que se desloca ao longo da curva que define a parede. Assim, para

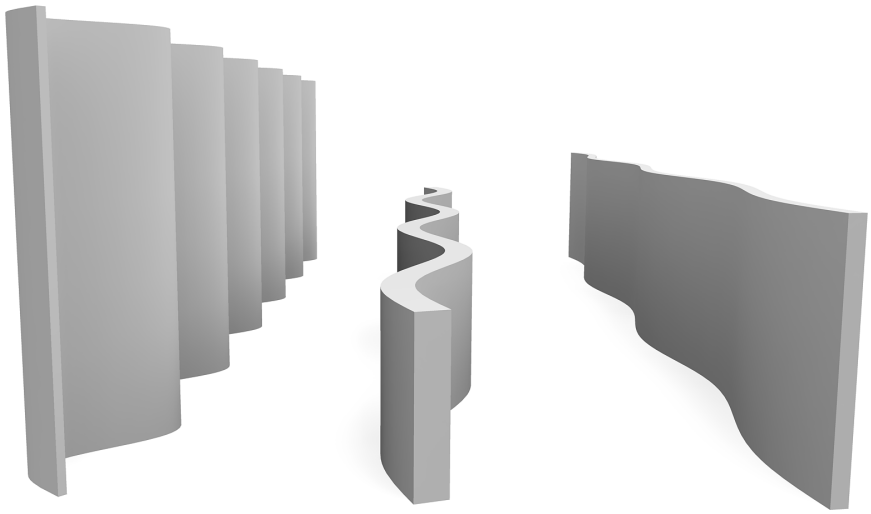


Figura 6.28: Três paredes com diferentes alturas e espessuras construídas sobre curvas sinusóides.

construirmos uma parede sinusoidal, basta-nos criar a linha de "guia" com a forma de uma sinusóide onde deslocaremos o perfil rectangular da parede:

```
parede_sinusoidal(p, a, omega, fi, x0, x1, dx, altura, largura) =
  sweep(
    spline(
      pontos_sinusoide(p+vxyz(0, largura/2.0, altura/2.0),
                       a, omega, fi, x0, x1, dx)),
      surface_rectangle(xy(-largura/2.0, -altura/2.0), largura, altura))
```

A função anterior permite-nos facilmente construir as paredes pretendidas. As seguinte expressões foram usadas para a geração da imagem representada na Figura 6.28.

```
parede_sinusoidal(xy(0, 0), 0.2, 1, 0, 0, 6*pi, 0.4, 3, 0.2)
parede_sinusoidal(xy(0, 4), 0.4, 1, 0, 0, 6*pi, 0.4, 2, 0.4)
parede_sinusoidal(xy(0, 8), 0.6, 2, 0, 0, 6*pi, 0.2, 5, 0.1)
```

Exercício 6.6.10 Refaça o exercício 6.6.3 mas usando uma extrusão ao longo de um caminho.

Exercício 6.6.11 Defina uma função `parede_pontos` que, dada a espessura e altura de uma parede e dada uma curva descrita por uma lista de pontos, constrói uma parede através do deslocamento de uma secção rectangular com a espessura e altura dadas ao longo de uma *spline* que passa pelas coordenadas dadas. Se precisar, pode usar a função `spline` que, a partir de uma lista de pontos, constrói uma spline que passa por esses pontos.

6.6.3 Extrusão com Transformação

A operação de extrusão ao longo de um caminho disponibilizada pelo Khepri permite ainda aplicar uma transformação à secção à medida que esta vai sendo extrudida. A transformação consiste numa rotação e num factor de escala, sendo a rotação particularmente útil para a modelação de formas com torções. Quer a rotação quer o factor de escala são parâmetros opcionais da função `sweep`.

A título de exemplo, consideremos a modelação das colunas presentes na fachada do edifício representado na Figura 6.29.

Uma vez que as colunas presentes na fachada correspondem a paralelipípedos que sofreram uma torção de 90^o, podemos simular essa forma através do uso de uma secção rectangular que é rodada esses mesmos 90^o durante o seu processo de extrusão:

```
sweep(line(u0(), z(30)),
      surface_polygon(xy(-1, -1),
                      xy(1, -1),
                      xy(1, 1),
                      xy(-1, 1)),
      pi/2)
```

Para melhor percebermos o efeito da torção, a Figura 6.30 mostra o processo de torção de colunas de secção quadrada em sucessivos incrementos de ângulo de $\frac{\pi}{4}$, desde uma torção máxima de 2π no sentido dos ponteiros do relógio até à torção máxima de 2π no sentido oposto.

6.7 Colunas de Gaudí

Antoni Gaudí, um dos maiores arquitectos de todos os tempos, liderou o Modernismo Catalão com uma interpretação muito pessoal do movimento Arte Nova, que combinava elementos góticos, elementos surrealistas e influências orientais. Gaudí procurava que as suas obras tivessem uma forte relação com a natureza que era a sua maior fonte de inspiração. Na sua arquitectura é frequente encontrarem-se referências a elementos naturais, como superfícies semelhantes a ondas e estruturas de suporte fortemente inspiradas nas formas das árvores.

Em 1883, com apenas 31 anos, Gaudí começou a trabalhar no Templo Expiatório da Sagrada Família, em Barcelona, onde explorou fantásticas combinações de formas que fazem deste templo, ainda inacabado, uma das obras mais singulares de toda a Arquitectura.

Nesta secção vamos debruçar-nos sobre uma ínfima parte desta obra, nomeadamente, as colunas idealizadas por Gaudí e que se encontram parcialmente visíveis na Figura 6.31.

Como se pode constatar pela observação da figura, Gaudí imaginou colunas cuja forma varia ao longo da coluna. O seu objectivo era sugerir

Figura 6.29: Colunas "torcidas" na fachada de um edifício em Atlanta. Fotografia de Pauly.

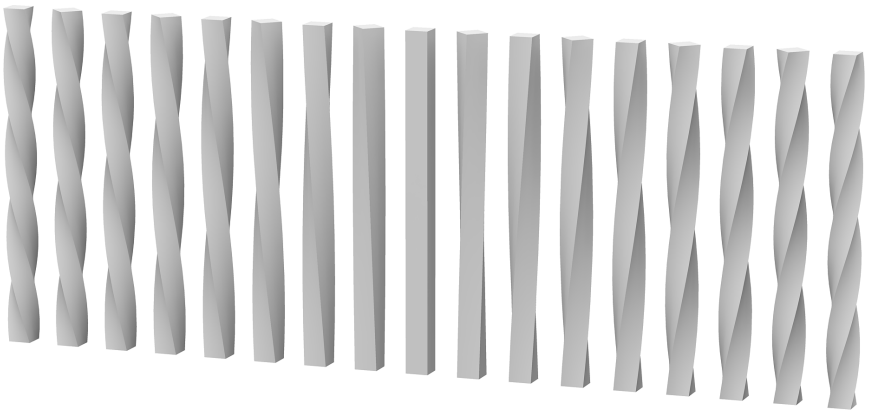


Figura 6.30: Modelação da torção de colunas de secção quadrada. Da esquerda para a direita, as colunas foram torcidas em incrementos de $\frac{\pi}{4}$ radianos.

uma "floresta" de pedra e, para isso, modelou colunas que se ramificam de outras colunas, como se de ramos de uma árvore se tratassem, sendo que os pontos de união entre as colunas se assemelham aos "nós" dos troncos de uma árvore. A variação das formas das colunas é perfeitamente visível nalguns destes "ramos," cuja base é quadrada, mas que, no topo, assumem uma forma quase circular. Noutros casos, a coluna termina com uma secção em forma de estrela.

Para modelar estas colunas, Gaudí dispendeu dois anos a elaborar um esquema construtivo baseado na intersecção e união de formas helicoidais produzidas pela "torção" de prismas. No caso mais simples, estes prismas possuem uma secção quadrada e sofrem uma torção de dezasseis avos de círculo em ambos os sentidos, i.e., $2\pi/16 = \pi/8$.

Para generalizarmos a abordagem de Gaudi, vamos implementar uma função que lide com o caso geral de um prisma de n lados que é torcido um ângulo arbitrário. Para isso, vamos usar a função `surface_regular_polygon` que constrói um polígono regular a partir do número de vértices n, do centro do polígono p, da distância r dos vértices ao ponto p, e do ângulo ϕ do primeiro vértice com o eixo X.

Para modelarmos o prisma torcido, vamos criar uma superfície com a forma do polígono regular e vamos extrudir essa superfície até uma altura h ao mesmo tempo que a torcemos de um determinado ângulo Δ_ϕ e lhe aplicamos um factor de escala e:

```
prisma_torcido(p, r, n, h, fi, dfi, e) =
  sweep(line(p, p+vz(h)),
        surface_regular_polygon(n, u0(), r, fi), dfi, e)
```

Para reproduzirmos a abordagem de Gaudí, vamos agora produzir a intersecção de dois destes prismas, ambos torcidos um ângulo de $\pi/8$, mas o

Figura 6.31: Colunas de suporte do Templo da Sagrada Família em Barcelona. Fotografia de Piqui Cuervo.

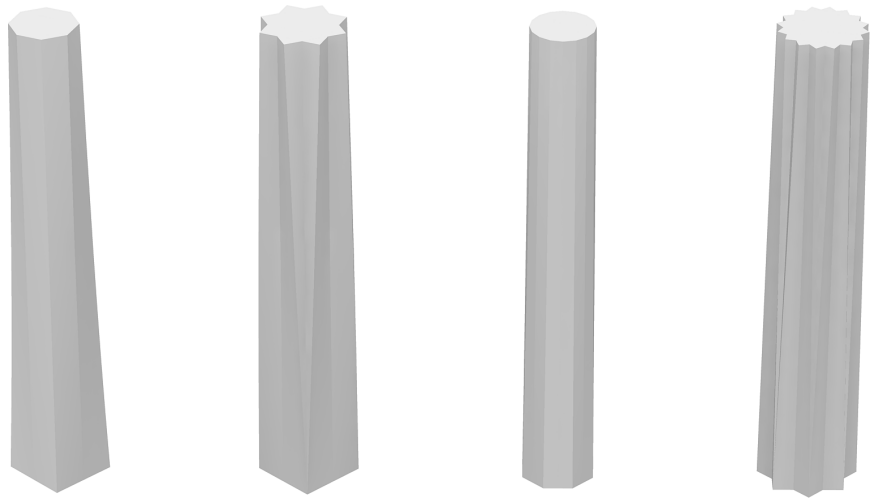


Figura 6.32: Colunas obtidas por intersecção e união de prismas torcidos de secção quadrada.

primeiro num sentido e o segundo no outro. Para obtermos mais realismo, vamos também aplicar um factor de escala de 0.9, de modo a que a coluna se vá estreitando à medida que sobe:

```
intersection(
  prisma_torcido(xy(0, 0), 1, 4, 10, 0, +pi/8, 0.9),
  prisma_torcido(xy(0, 0), 1, 4, 10, 0, -pi/8, 0.9))
```

O resultado é a coluna mais à esquerda na Figura 6.32. A união destes prismas produz outra das formas empregues por Gaudí, visível imediatamente à direita da coluna anterior:

```
union(
  prisma_torcido(xy(5, 0), 1, 4, 10, 0, +pi/8, 0.9),
  prisma_torcido(xy(5, 0), 1, 4, 10, 0, -pi/8, 0.9))
```

Tal como Gaudí o fez, podemos complementar as colunas anteriores com a sua extensão natural, duplicando o número de lados dos prismas e reduzindo o ângulo de torção para metade:

```
intersection(
  prisma_torcido(xy(10, 0), 1, 4, 10, 0, +pi/16, 0.9),
  prisma_torcido(xy(10, 0), 1, 4, 10, 0, -pi/16, 0.9),
  prisma_torcido(xy(10, 0), 1, 4, 10, pi/4, +pi/16, 0.9),
  prisma_torcido(xy(10, 0), 1, 4, 10, pi/4, -pi/16, 0.9))
```

```
union(
  prisma_torcido(xy(15, 0), 1, 4, 10, 0, +pi/16, 0.9),
  prisma_torcido(xy(15, 0), 1, 4, 10, 0, -pi/16, 0.9),
  prisma_torcido(xy(15, 0), 1, 4, 10, pi/4, +pi/16, 0.9),
  prisma_torcido(xy(15, 0), 1, 4, 10, pi/4, -pi/16, 0.9))
```

Os resultados estão visíveis nas duas colunas à direita, na Figura 6.32.

6.8 Revoluções

Uma superfície de revolução é uma superfície gerada pela rotação de uma curva bi-dimensional em torno de um eixo. Um sólido de revolução é um sólido gerado pela rotação de uma região bidimensional em torno de um eixo.

Sendo a rotação um processo muito simples de criação de superfícies e sólidos, é natural que o Khepri disponibilize uma operação para o fazer. A função `revolve` destina-se precisamente a esse fim. A função `revolve` recebe, como argumentos, a região a "revolver" e, opcionalmente, um ponto no eixo de rotação (por omissão, a origem), um vector paralelo ao eixo de rotação (por omissão, o vector na direcção do eixo Z), o ângulo inicial de revolução (por omissão, zero) e o incremento de ângulo para a revolução (por omissão, $2 \cdot \pi$). Naturalmente, se se omitir o incremento ou este for de $2 \cdot \pi$ radianos, obtém-se uma revolução completa.

6.8.1 Superfícies de Revolução

Usando estas funções torna-se agora mais fácil criar superfícies ou sólidos de revolução. Consideremos, por exemplo, a *spline* apresentada na Figura 6.33, que foi gerada a partir da seguinte expressão:

```
spline(xyz(0, 0, 2), xyz(1, 0, -1),
       xyz(2, 0, 1), xyz(3, 0, -1),
       xyz(4, 0, 0), xyz(5, 0, -1))
```

Reparemos que a spline está assente no plano XZ e, por isso, pode ser usada como base para uma superfície cujo eixo de revolução é o eixo Z. Para visualizarmos melhor o "interior" da superfície, vamos considerar uma abertura de $\frac{2 \cdot \pi}{4}$. A expressão Julia correspondente é:

```
revolve(spline(xyz(0, 0, 2), xyz(1, 0, -1),
               xyz(2, 0, 1), xyz(3, 0, -1),
               xyz(4, 0, 0), xyz(5, 0, -1)),
        u0(),
        vz(),
        1/4*2*pi,
        3/4*2*pi)
```

O resultado da avaliação da expressão anterior está representado na Figura 6.34.

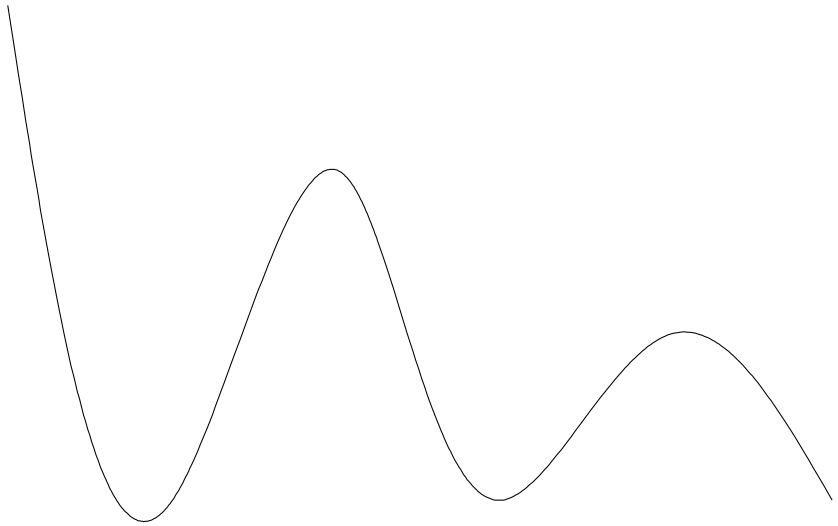


Figura 6.33: Uma *spline* usada como base para a construção de uma superfície de revolução.

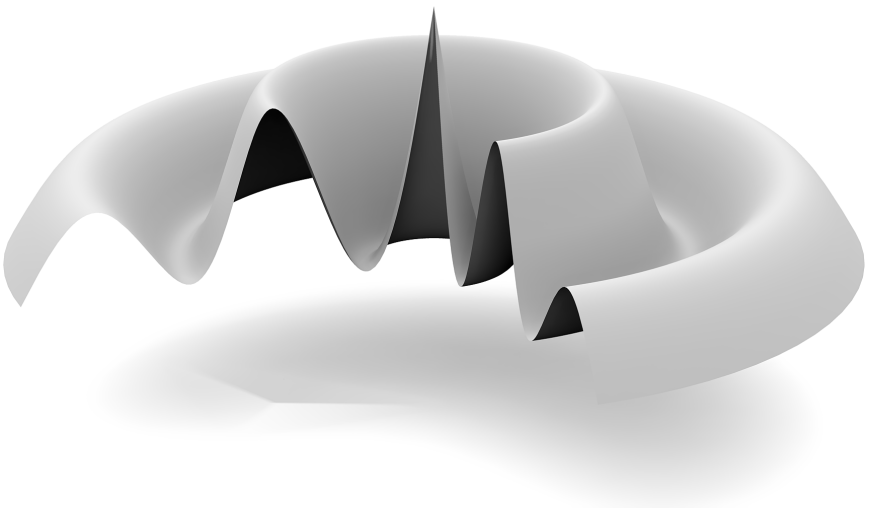


Figura 6.34: Uma superfície de revolução gerada pela *spline* representada na Figura 6.33.

Figura 6.35: Cúpulas da Catedral de Verkho o Salvador, em Moscovo, Rússia. Fotografia de Amanda Graham.

As superfícies de revolução são frequentemente usadas em Arquitectura, nomeadamente, para a construção de cúpulas. A cúpula em forma de cebola, por exemplo, é um elemento muito explorado quer na arquitectura russa, quer na arquitectura Mughal, quer noutras arquitecturas influenciadas por estas. A imagem da Figura 6.35 mostra as cúpulas da Catedral de Verkho o Salvador, em Moscovo, Rússia.

As cúpulas em forma cebola possuem uma simetria axial que permite modelá-las como se de superfícies de revolução se tratassem. Para isso, vamos definir uma função denominada `cupula_revolucao` que, a partir de uma lista de coordenadas pertencentes ao perfil da cúpula, constrói a superfície de revolução que modela a cúpula. Para simplificar vamos admitir que a cúpula estará fechada no topo e que o topo é dado pelo primeiro elemento da lista de coordenadas. Esta simplificação permite estabelecer o eixo de rotação da superfície a partir do primeiro elemento da lista de coordenadas:

```
cupula_revolucao(pontos) =
  revolve(spline(pontos), pontos[1])
```

Para testarmos se a função modela correctamente uma cúpula, podemos obter algumas coordenadas do perfil de cúpulas reais e usá-las para invocar a função. Foi precisamente isso que fizemos nas seguintes expressões

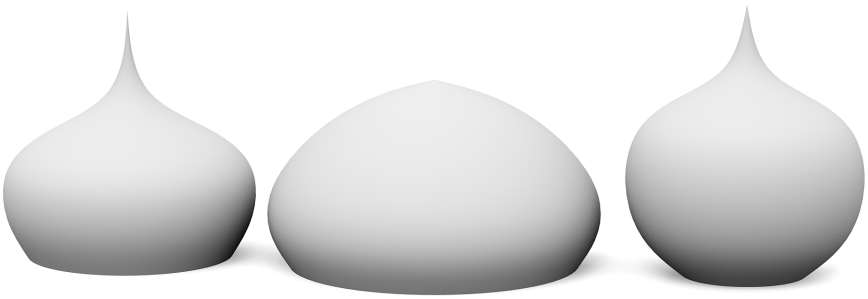


Figura 6.36: Cúpulas geradas em Khepri.

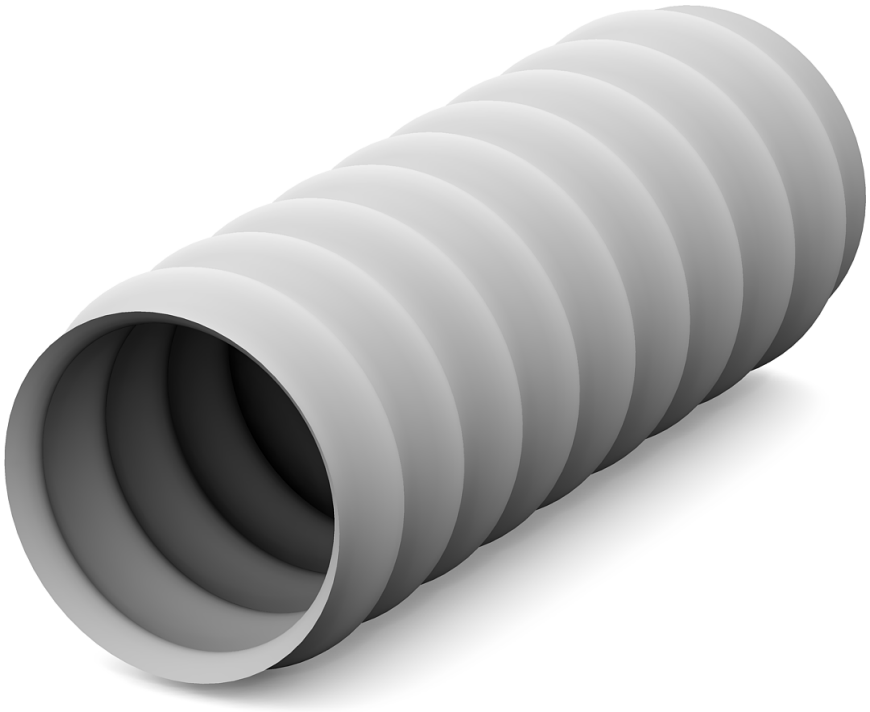
```
cupula_revolucao([xyz(0, 0, 12),
                  xyz(0, 1, 8),
                  xyz(0, 6, 4),
                  xyz(0, 6, 2),
                  xyz(0, 5, 0)])
```

```
cupula_revolucao([xyz(15, 0, 9),
                  xyz(15, 3, 8),
                  xyz(15, 7, 5),
                  xyz(15, 8, 3),
                  xyz(15, 8, 2),
                  xyz(15, 7, 0)])
```

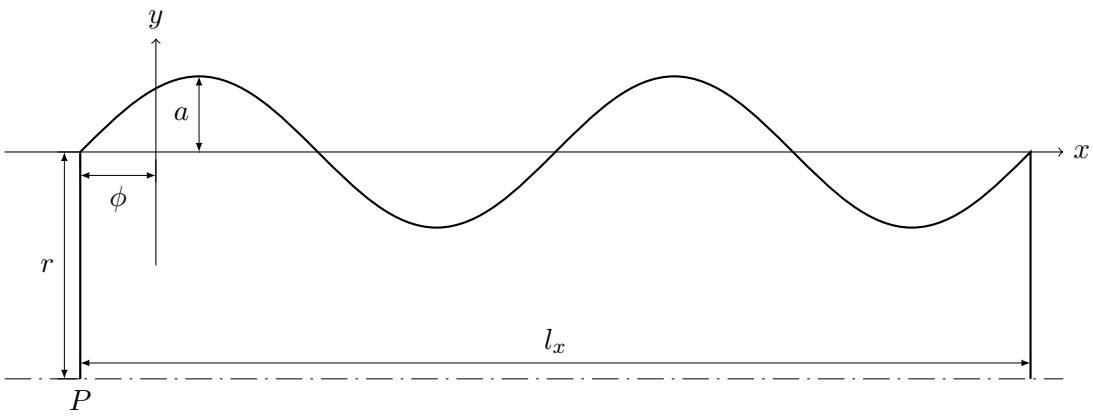
```
cupula_revolucao([xyz(30, 0, 13),
                  xyz(30, 1, 10),
                  xyz(30, 5, 7),
                  xyz(30, 5, 2),
                  xyz(30, 3, 0)])
```

que criam, da esquerda para a direita, as superfícies apresentadas na Figura 6.36.

Exercício 6.8.1 Considere o tubo de perfil sinusoidal apresentado na imagem seguinte.

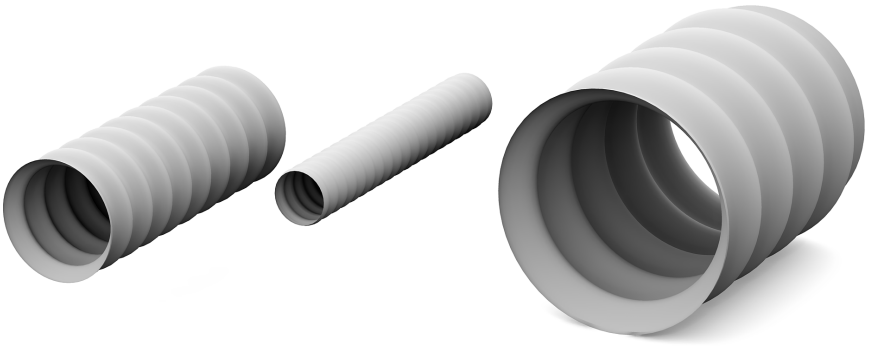


O tubo foi produzido tendo em conta os parâmetros geométricos descritos no perfil apresentado em seguida:

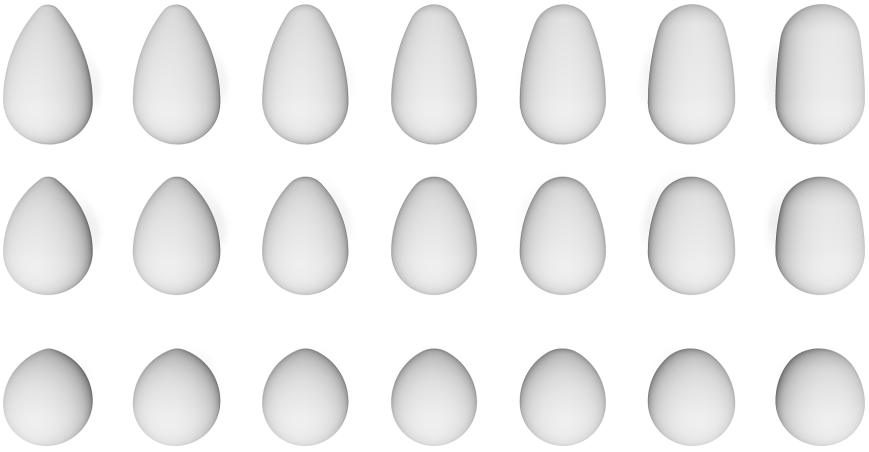


Defina a função `tubo_sinusoidal` que, a partir dos parâmetros anteriores P, r, a, $omega$, fi, lx e, finalmente, da separação entre pontos de interpolação Δ_x, gera o tubo sinusoidal pretendido. Por exemplo, os tubos apresentados na imagem seguinte foram gerados pela avaliação das seguintes expressões:

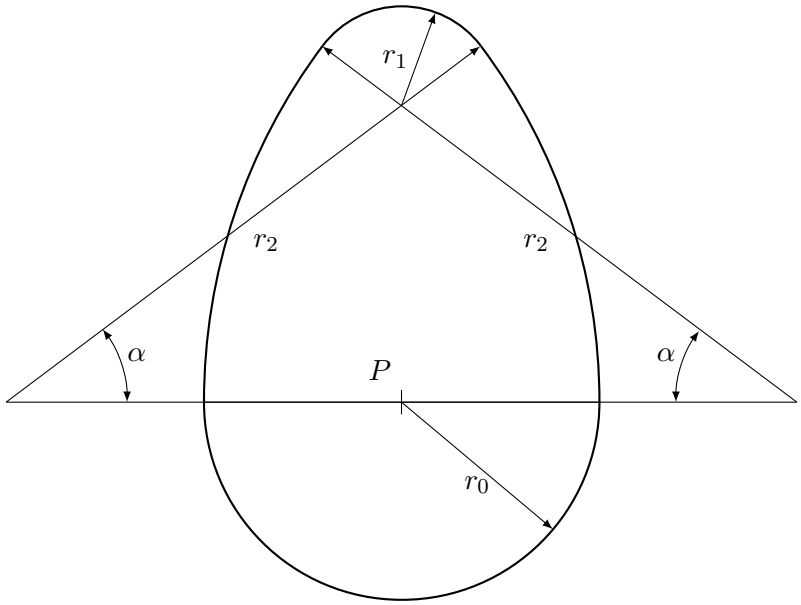
```
tubo_sinusoidal(xyz(-20, 80, 0), 20, 2.0, 0.5, 0, 60, 0.2)
tubo_sinusoidal(xyz(0, 30, 0), 5, 0.2, 2.0, 0, 60, 0.1)
tubo_sinusoidal(xyz(30, 0, 0), 10, 1.0, 1.0, pi/2, 60, 0.2)
```



Exercício 6.8.2 Considere um ovo como generalização de ovos de diferentes proporções, tal como se ilustra na seguinte imagem:



O ovo, embora seja um objecto tridimensional, tem a secção que se esquematiza na imagem seguinte:



Defina uma função `ovo` que constrói um ovo. A função deverá receber, apenas, as coordenadas do ponto P, os raios r_0 e r_1 e, finalmente, a altura h do ovo.

6.8.2 Sólidos de Revolução

Se, ao invés de usarmos uma curva para produzir a superfície de revolução, usarmos uma região, produziremos um sólido de revolução. A título de exemplo, vamos explorar a utilização de um trifólio para gerar um arco. A imagem da esquerda da Figura 6.10 mostra a região usada como ponto de partida e a Figura 6.37 mostra o sólido criado. Para este exemplo, usámos a seguinte expressão:

```
revolve(n_folio(xy(3, 0), 1, 3),
        u0(),
        vy(),
        0,
        pi)
```

Através da combinação de extrusões com revoluções é possível produzir modelos bastante sofisticados. Consideremos, por exemplo, a modelação de uma arcada cujas colunas e arcos possuem secções em forma de n-fólio. Para modelar estas formas é necessário saber as coordenadas p do centro do fólio usado para gerar a coluna ou o arco, o raio r_c das colunas e o número de fólios a usar n_f. Para as colunas é ainda necessário sabermos a altura h e, finalmente, para o arco é necessário saber o seu raio r_a. As seguintes funções implementam essas formas:

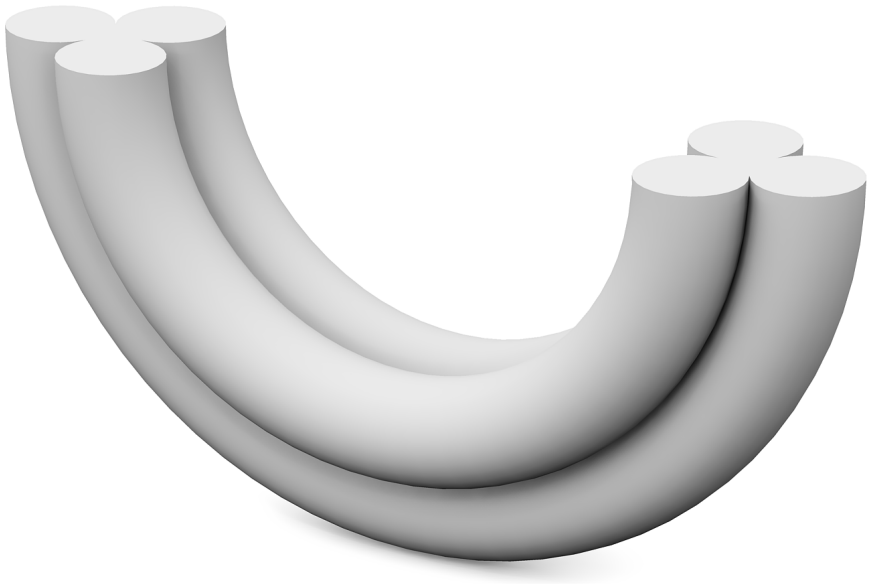


Figura 6.37: Um sólido de revolução gerado pelo trifólio representado na Figura 6.10.

```
coluna_folio(p, rc, nf, h) =
  extrusion(n_folio(p, rc, nf), h)

arco_folio(p, rc, ra, nf) =
revolve(n_folio(p, rc, nf),
        p+vx(ra),
        vy(),
        0,
        pi)
```

Para construirmos a arcada, o mais simples será definir uma função que constrói uma coluna e um arco e que, recursivamente, constrói as restantes arcadas até não ter mais arcadas para construir, caso em que constrói a última coluna que suporta o último arco. A tradução deste algoritmo para Julia é:

```
arcadas_folio(p, rc, nf, ra, h, n) =
  if n == 0
    coluna_folio(p, rc, nf, h)
  else
    union(coluna_folio(p, rc, nf, h),
          arco_folio(p+vxyz(0, 0, h), rc, ra, nf),
          arcadas_folio(p+vxyz(2*ra, 0, 0), rc, nf, ra, h, n-1))
  end
```

A imagem da Figura 6.38 apresenta uma perpectiva sobre uma série de arcadas geradas pelas seguintes expressões:

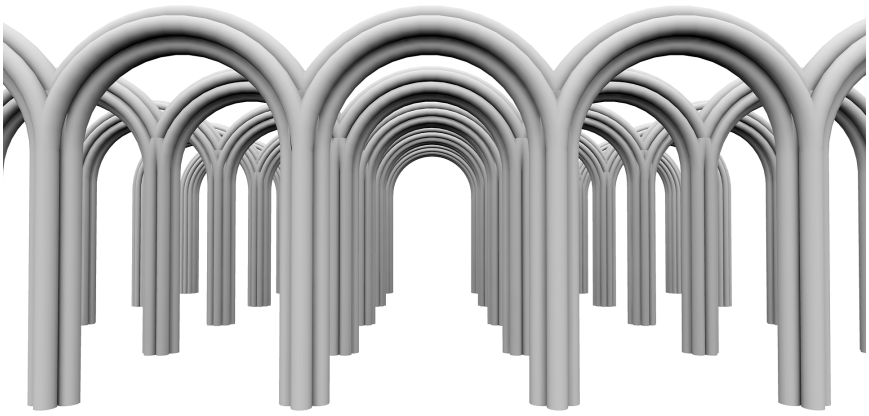
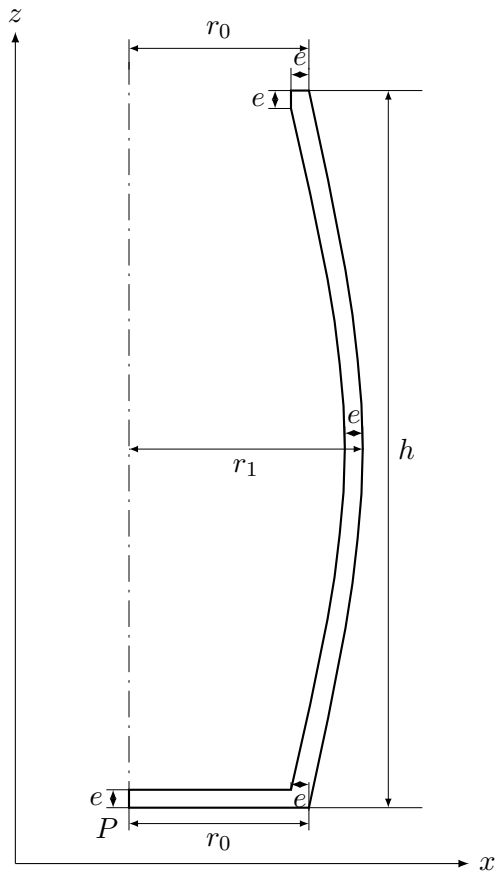


Figura 6.38: Arcadas geradas pela função `arcadas_folio`. Da frente para trás, as arcadas possuem uma secção em forma de quadrifólio, exafólio, octafólio, decafólio e dodecafólio.

```
arcadas_folio(xy(0, 0), 1, 4, 5, 10, 5)
arcadas_folio(xy(0, 10), 1, 6, 5, 10, 5)
arcadas_folio(xy(0, 20), 1, 8, 5, 10, 5)
arcadas_folio(xy(0, 30), 1, 10, 5, 10, 5)
arcadas_folio(xy(0, 40), 1, 12, 5, 10, 5)
```

Exercício 6.8.3 Pretende-se criar um programa capaz de gerar um barril com a base assente no plano XY. Considere que a base do barril está fechada mas o topo está aberto.

Crie uma função denominada `perfil_barril` que recebe o ponto P, os raios r_0 e r_1, a altura h e a espessura e e que devolve a região que define o perfil do barril, tal como se apresenta na imagem seguinte:

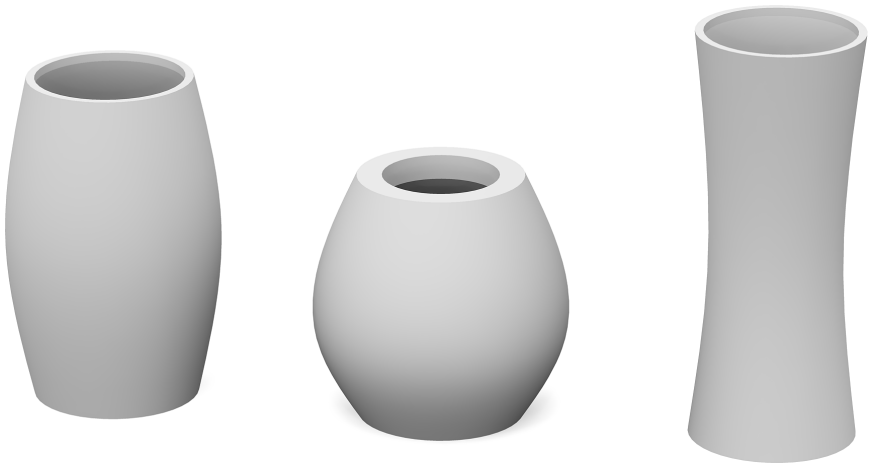


Exercício 6.8.4 Usando a função `perfil_barril`, defina a função `barril` que, tendo os mesmos parâmetros da anterior, cria o barril tridimensional. A função deverá ter a seguinte forma:

```
barril(p, r0, r1, h, e) =
  ...
```

A título de exemplo, considere que a imagem seguinte foi gerada pela avaliação das expressões:

```
barril(xyz(0, 0, 0), 1, 1.3, 4, 0.1)
barril(xyz(4, 0, 0), 1, 1.5, 3, 0.3)
barril(xyz(8, 0, 0), 1, 0.8, 5, 0.1)
```

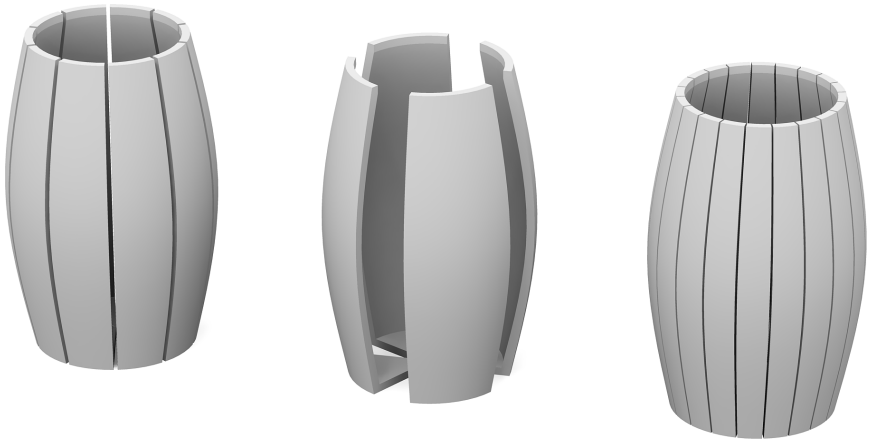


Exercício 6.8.5 O barril criado no exercício anterior é excessivamente "moderno," não representando os barris tradicionais de madeira que são constituídos por múltiplas tábuas de madeira encostadas umas às outras. A imagem seguinte mostra alguns exemplos destas tábuas, com dimensões e posicionamentos angulares diferentes:

Defina a função `tabua_barril` que, para além dos mesmos parâmetros que definem um barril, dado o ângulo de rotação inicial α, a que se inicia a tábua, e dado o incremento de ângulo Δ_α, que corresponde à âmplitude angular da tábua, constrói a secção tridimensional do barril correspondente à tábua em questão.

Exercício 6.8.6 Defina uma função denominada `tabuas_barril` que recebe como parâmetros o ponto P, os raios r_0 e r_1, a altura h, a espessura e, o número de tábuas n e a "folga" angular entre tábuas s, e que cria um barril tridimensional com esse número de tábuas. A imagem seguinte mostra alguns exemplos destes barris e foi criada pela avaliação das seguintes expressões:

```
tabuas_barril(xyz(0, 0, 0), 1, 1.3, 4, 0.1, 10, 0.05)
tabuas_barril(xyz(4, 0, 0), 1, 1.3, 4, 0.1, 4, 0.5)
tabuas_barril(xyz(8, 0, 0), 1, 1.3, 4, 0.1, 20, 0.01)
```



6.9 Interpolação de Secções

A interpolação de secções, também conhecida por metamorfose de formas, permite a criação de um objecto tridimensional através da interpolação de secções planas desse objecto. Em Khepri, a interpolação de secções é obtida através da função `loft` ou de uma das suas variantes. Estas funções operam a partir de uma lista ordenada das secções planas (que poderão ser curvas ou regiões), que, para minimizar incorrecções no processo de interpolação, pode opcionalmente ser complementada com linhas de guiamento. As linhas de guiamento são particularmente importantes no caso de secções planas com partes rectilíneas.

Nas próximas secções iremos analizar separadamente cada uma destas formas de interpolação.

6.9.1 Interpolação por Secções

Comecemos pela interpolação usando apenas secções. Há duas formas fundamentais de fazer esta interpolação: usando uma superfície regrada (i.e., superfícies geradas por um segmento de recta que se desloca) entre cada secção plana, ou usando uma superfície "suave" (i.e., sem mudanças bruscas de inclinação) que se aproxima das várias secções. A interpolação por superfície regrada é implementada pela função `loft_ruled`, enquanto que a interpolação por superfície suave é realizada pela função `loft`.

Para compararmos os efeitos destas duas funções, consideremos as seguintes expressões que criam a interpolação por superfície regrada a partir de três círculos, representada à esquerda na Figura 6.39:

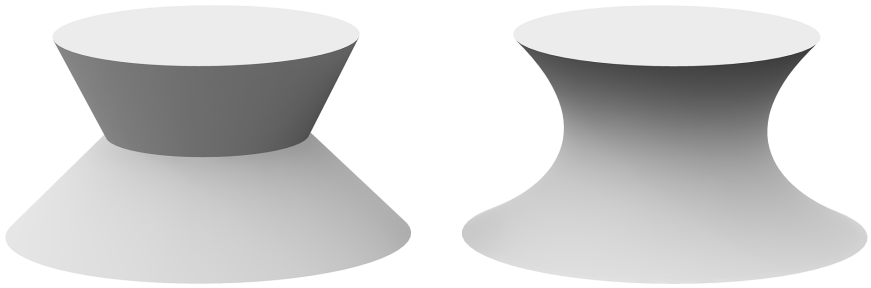


Figura 6.39: Interpolação de superfícies a partir de três círculos. À esquerda, usando interpolação com superfícies regradas. À direita, usando interpolação com superfícies suaves.

```
circ0 = surface_circle(xyz(0, 0, 0), 4)
circ1 = surface_circle(xyz(0, 0, 2), 2)
circ2 = surface_circle(xyz(0, 0, 4), 3)
loft_ruled([circ0, circ1, circ2])
```

Na mesma Figura 6.39, à direita, encontra-se a interpolação por superfície suave dos mesmos três círculos, gerada pelas seguintes expressões:

```
circ0 = surface_circle(xyz(0, 9, 0), 4)
circ1 = surface_circle(xyz(0, 9, 2), 2)
circ2 = surface_circle(xyz(0, 9, 4), 3)
loft([circ0, circ1, circ2])
```

6.9.2 Interpolação com Guiamento

É relativamente óbvio que existe um número ilimitado de diferentes superfícies que interpolam duas dadas secções. Os algoritmos de interpolação disponibilizados pelo Khepri têm assim que tomar algumas opções para decidir qual a interpolação real que vão produzir. Infelizmente, nem sempre as opções tomadas são as que o utilizador pretende e é perfeitamente possível (e até frequente) que a interpolação produzida não corresponda ao resultado desejado, em especial quando é necessário fazer uma interpolação entre duas secções muito diferentes.

Este comportamento é visível na Figura 6.40 onde apresentamos a interpolação entre um hexágono e um triângulo e onde é perfeitamente visível a falta de uniformidade da interpolação.[3] A figura do lado esquerdo foi gerada pela expressão:

[3] A interpolação exacta que é produzida depende não só da aplicação de CAD que se está a usar mas também da versão dessa aplicação.

```
loft([surface_regular_polygon(6, xyz(0, 0, 0), 1.0, 0, true),
      surface_regular_polygon(3, xyz(0, 0, 5), 0.5, pi/6, true)])
```

Note-se, no lado esquerdo da Figura 6.40, que um dos lados do triângulo é directamente mapeado num dos lados do hexágono, forçando a interpolação a ter de distorcer os restantes dois lados do triângulo para os mapear nos restantes cinco lados do hexágono.

Felizmente, o Khepri disponibiliza outras formas de fazer a interpolação de secções que minimizam as possibilidades de erro, em particular, através da função `loft`. Esta função recebe não só a lista das secções a interpolar mas também uma lista de curvas de guiamento que restringem a interpolação. Esta função permite-nos resolver o problema reportado na Figura 6.40 através do estabelecimento de linhas de guiamento entre os vértices relevantes de cada um dos polígonos. Estas linhas de guiamento vão restringir o Khepri a gerar uma interpolação que contenha essas linhas na sua superfície e, consequentemente, servem para controlar o emparelhamento de pontos entre as secções a interpolar. Para isso, vamos simplesmente gerar três vértices do hexágono que, juntamente com os três vértices do triângulo, permitirão construir as linhas de guiamento:

Uma vez que precisamos de construir as linhas de guiamento a partir dos vértices dos polígonos, vamos definir uma função que, com duas listas de pontos, cria uma lista de linhas que unem os pontos dois a dois:

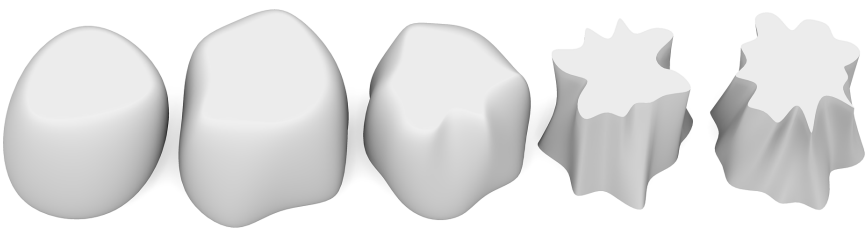
```
guias(p0s, p1s) =
  [line(p0, p1) for (p0, p1) in zip(p0s, p1s)]
```

Em seguida, vamos simplesmente gerar três vértices do hexágono que, juntamente com os três vértices do triângulo, permitirão construir as linhas de guiamento:

```
loft([surface_regular_polygon(6, xyz(3, 0, 0), 1.0, 0, true),
      surface_regular_polygon(3, xyz(3, 0, 5), 0.5, pi/6, true)],
     guias(regular_polygon_vertices(3, xyz(3, 0, 0), 1.0, 0, true),
           regular_polygon_vertices(3, xyz(3, 0, 5), 0.5, pi/6, true)))
```

A avaliação da expressão anterior permite ao Khepri produzir uma interpolação mais correcta entre o hexágono e o triângulo, tal como é visível no lado direito da Figura 6.40.

Exercício 6.9.1 Considere a interpolação entre duas curvas fechadas aleatórias posicionadas a alturas diferentes apresentadas na seguinte figura:



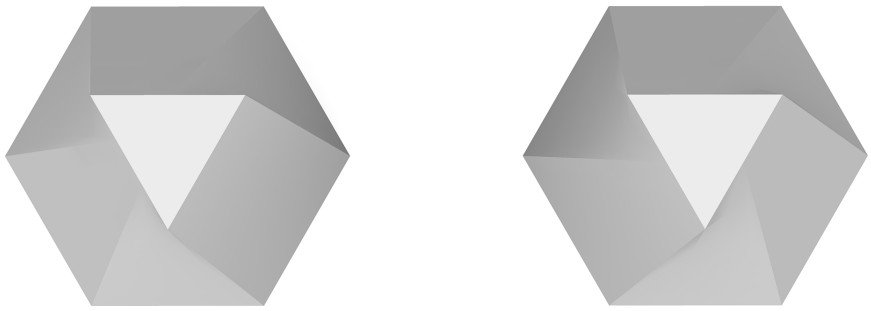


Figura 6.40: Vista de topo da interpolação entre uma secção hexagonal e uma secção triangular. À esquerda, a interpolação foi realizada sem linhas de guiamento. À direita, a interpolação foi realizada com linhas de guiamento que associam três vértices igualmente espaçados do hexágono aos três vértices do triangulo.

Cada uma das curvas é uma *spline* produzida a partir de pontos gerados pela função `pontos_circulo_raio_aleatorio` definida no exercício 5.5.2. Escreva uma expressão capaz de gerar um sólido semelhante aos apresentados na imagem anterior.

Capítulo 7

Transformações

7.1 Introdução

Até agora, todos os objectos que construímos possuem um carácter definitivo: os parâmetros usados para os construir determinam univocamente a forma desses objectos e tudo o que podemos fazer é invocar as funções que constroem esses objectos com diferentes parâmetros para construirmos objectos diferentes.

Embora essa forma de construção de objectos seja suficientemente poderosa, existem alternativas potencialmente mais práticas que se baseiam na *modificação* de objectos previamente criados. Essa modificação é realizada através de operações de *translação*, *rotação*, *reflexão* e *escala*.

É importante salientarmos que estas operações não criam novos objectos, apenas afectam aqueles a que se aplicam. Por exemplo, após a aplicação de uma translação a um objecto, este objecto muda simplesmente de posição.

Contudo, por vezes queremos aplicar transformações a objectos que produzam novos objectos como resultado. Por exemplo, podemos querer produzir a translação de um objecto, mas deixando o objecto original no lugar onde estava. Uma forma de o conseguirmos será aplicar a operação de translação, não ao objecto original, mas sim a uma sua cópia. A possibilidade de criarmos cópias dos objectos pode então ser usada para distinguirmos o caso em que queremos que uma operação modifique um objecto do caso em que queremos que uma operação crie um novo objecto. Para este propósito, o Khepri disponibiliza a operação `copy_shape` que recebe um objecto como argumento e devolve uma cópia desse objecto, localizada exactamente no mesmo local do original. Naturalmente, na ferramenta de CAD irão aparecer dois objectos iguais sobrepostos. Tipicamente, o objecto que foi copiado será subsequentemente transformado, por exemplo, movendo-o para outro local.

De seguida iremos ver quais são as operações de transformação dispo-

níveis em Khepri.

7.2 Translação

A operação de *translação* move um objecto através da adição de um vector a todos os seus pontos, fazendo com que todos os pontos se movam uma determinada distância numa determinada direcção. Os componentes do vector indicam qual o deslocamento relativamente a cada um dos eixos coordenados.

Para realizar esta operação o Khepri disponibiliza a operação `move`, que recebe um objecto e um vector deslocamento para realizar a operação de translação. Como exemplo, temos:

```
julia> move(sphere(), vxyz(1, 2, 3))
Move(...)
```

Notemos que a função devolve o objecto que sofreu a translação para permitir o seu fácil encadeamento com outras operações.

É fácil de ver que, para o exemplo anterior, é mais simples especificar imediatamente qual é o centro da esfera, escrevendo:

```
julia> sphere(xyz(1, 2, 3))
Sphere(...)
```

No entanto, em casos mais complexos pode ser muito vantajoso considerar que os objectos se constroem na origem e sofrem posteriormente uma translação para a posição desejada.

Consideremos, por exemplo, uma *cruz papal*, que se define pela união de três cilindros horizontais de comprimento progressivamente decrescente dispostos ao longo de um cilindro vertical, tal como se pode ver na Figura 7.1. É de salientar que os cilindros têm todos o mesmo raio e que o seu comprimento e posicionamento é função desse raio. Em termos de proporção, o cilindro vertical da cruz papal tem um comprimento igual a 20 raios, enquanto que os cilindros horizontais possuem comprimentos iguais a 14, 10 e 6 raios e o seu eixo está posicionado a uma altura igual a 9, 13 e 17 raios. Estas proporções são implementadas pela seguinte função a partir de um ponto de referência p:

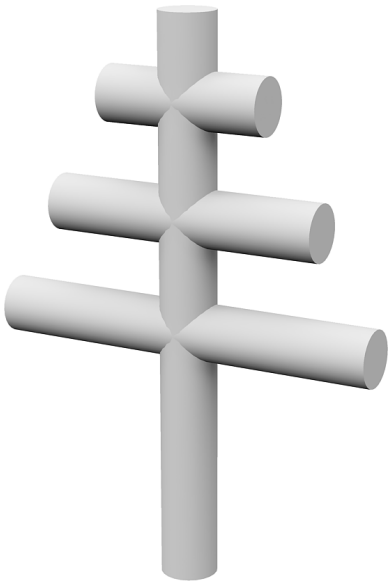


Figura 7.1: Uma cruz papal.

```
cruz_papal(p, raio) =
  union(cylinder(p,
                 raio,
                 p+vz(20*raio)),
        cylinder(p+vxz(-7*raio, 9*raio),
                 raio,
                 p+vxz(7*raio, 9*raio)),
        cylinder(p+vxz(-5*raio, 13*raio),
                 raio,
                 p+vxz(5*raio, 13*raio)),
        cylinder(p+vxz(-3*raio, 17*raio),
                 raio,
                 p+vxz(3*raio, 17*raio)))
```

Contudo, se assumirmos que a cruz se encontra inicialmente posicionada na origem, podemos simplificar ligeiramente a definição anterior:

```
cruz_papal(raio) =
  union(cylinder(u0(),
                 raio,
                 z(20*raio)),
        cylinder(xz(-7*raio, 9*raio),
                 raio,
                 xz(7*raio, 9*raio)),
        cylinder(xz(-5*raio, 13*raio),
                 raio,
                 xz(5*raio, 13*raio)),
        cylinder(xz(-3*raio, 17*raio),
                 raio,
                 xz(3*raio, 17*raio)))
```

Naturalmente se pretendermos colocar a cruz numa posição específica, por exemplo, $(1, 2)$, devemos escrever:

```
move(cruz_papal(1), xy(1, 2))
```

7.3 Escala

A *escala* consiste numa transformação que aumenta ou diminui a dimensão de uma entidade sem lhe modificar a forma. Esta operação é também denominada de *homotetia*. Embora seja concebível ter uma operação de escala que modifica independentemente cada uma das dimensões, é mais usual empregar-se uma escala uniforme que modifica simultaneamente as três dimensões, afectando-as de um mesmo factor. Se o factor é maior que um, o tamanho aumenta. Se o factor é menor que um, o tamanho diminui.

No caso do Khepri, apenas é disponibilizada uma operação de escala uniforme: `scale`. É fácil vermos que a escala, para além de alterar a dimensão do objecto, pode alterar também a sua posição. Se esse efeito não for pretendido, a solução óbvia é aplicar previamente uma translação para centrar o objecto na origem, aplicar de seguida a operação de escala e, finalmente, aplicar a translação inversa para "devolver" o objecto à sua posição original.

Usando a operação de escala, é possível simplificar ainda mais a definição anterior. Na verdade, como a dimensão da cruz depende apenas do raio, podemos arbitrar um raio unitário que depois alteramos através de uma operação de escala. Assim, podemos escrever:

```
cruz_papal() =
  union(cylinder(u0(), 1, 20),
        cylinder(xz(-7, 9), 1, xz(7, 9)),
        cylinder(xz(-5, 13), 1, xz(5, 13)),
        cylinder(xz(-3, 17), 1, xz(3, 17)))
```

Se quisermos construir uma cruz papal com um determinado raio r, por exemplo $r = 3$, basta-nos agora escrever:

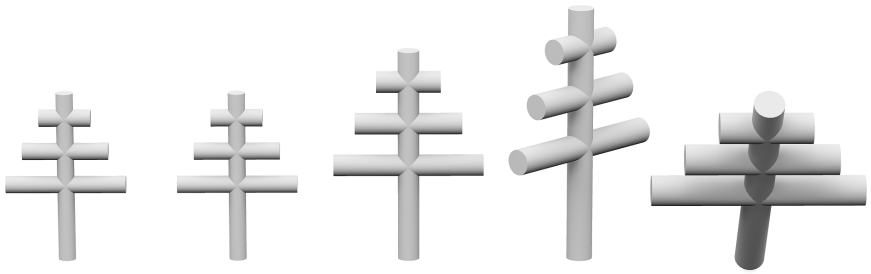


Figura 7.2: Da esquerda para a direita, uma cruz papal de raio unitário colocada na origem, seguida de uma translação, seguida de uma escala com translação, seguida de escala com rotação e translação e, finalmente, escala com rotação em torno do eixo X e translação.

```
scale(cruz_papal(), 3)
```

7.4 Rotação

Na *rotação*, todos os pontos de um objecto giram num movimento circular em torno de um ponto (a duas dimensões) ou eixo (a três dimensões). No caso geral de uma rotação a três dimensões é usual decompor-se esta em três rotações sucessivas em torno dos eixos coordenados. Estas rotações em torno dos eixos X, Y e Z designam-se *rotações principais*, por analogia com o conceito de *eixos principais* que se aplica a X, Y e Z. Cada uma destas rotações é realizada pela função `rotate` que recebe, como argumentos, o objecto sobre o qual se aplica a rotação, o ângulo de rotação, e dois pontos que definem o eixo de rotação. Por omissão, esses pontos são a origem e um ponto acima do anterior, o que implica que, por omissão, a rotação será feita em relação ao eixo Z.

A Figura 7.2 ilustra algumas combinações de translações, escalas e rotações, tendo sido gerada a partir do seguinte programa:

```
cruz_papal()
move(cruz_papal(), vx(20))
move(scale(cruz_papal(), 1.25), vx(40))
move(rotate(scale(cruz_papal(), 1.5), pi/4), vx(60))
move(rotate(scale(cruz_papal(), 1.75), pi/4, u0(), vx()), vx(80))
```

7.5 Reflexão

Para além da translação, escala, e rotação, o Khepri implementa ainda a operação de reflexão, disponibilizando, para isso, a função `mirror`. Esta função recebe, como argumentos, a forma a reflectir e um plano de reflexão

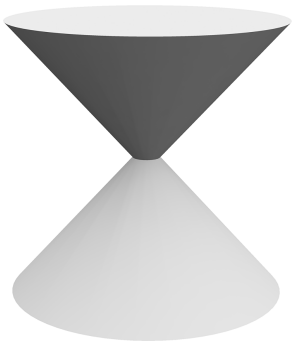


Figura 7.3: Uma ampulheta

descrito por um ponto contido no plano e por um vector normal ao plano. Por omissão, o plano de reflexão é o plano XY.

A título de exemplo, consideremos a ampulheta representada na Figura 7.3 e que tem como parâmetros o centro da base da ampulheta, o raio da base da ampulheta, o raio do estrangulamento da ampulheta e a altura da ampulheta.

Não é difícil conceber esta forma como uma união de dois troncos de cone:

```
ampulheta(p, rb, rc, h) =
  union(cone_frustum(p, rb, p+vz(h/2), rc),
        cone_frustum(p+vz(h/2), rc, p+vz(h), rb))
```

No entanto, é possível simplificarmos a definição anterior através do uso da operação `mirror`:

```
ampulheta(p, rb, rc, h) =
  mirror(cone_frustum(p, rb, p+vz(h/2), rc), p+vz(h/2))
```

Na secção seguinte iremos ver um exemplo em que estas operações irão ser utilizadas para a modelação de um dos edifícios mais famosos do mundo.

7.6 A Ópera de Sydney

A Ópera de Sydney resultou de um concurso internacional lançado em 1957 para a construção de um edifício destinado à realização de espectáculos. O vencedor foi o projecto de Jørn Utzon, um arquitecto dinamarquês pouco conhecido até então. O seu projecto, embora não cumprisse integralmente os requisitos do concurso, foi seleccionado pelo famoso arquitecto Eero Saarinen, então membro do júri, que o considerou desde logo como uma obra marcante. A proposta consistia num conjunto de estruturas em

Figura 7.4: A Ópera de Sydney. Fotografia de Brent Pearson.

forma de concha capazes de albergar várias salas de espectáculos. O resultado final desta proposta está representado na Figura 7.4.

Claramente inovador, o desenho de Utzon estava demasiado avançado para as tecnologias de projecto e construção da época e foi por muitos considerado como impossível. Nos três anos subsequentes à aprovação do projecto, Utzon, juntamente com a equipa de engenharia estrutural da empresa Ove Arup, tentou encontrar uma formulação matemática para as suas conchas desenhadas à mão, tendo experimentado uma grande variedade de diferentes abordagens, incluindo formas parabólicas, circulares e elípticas, mas todas as soluções encontradas tinham, para além de enormes dificuldades técnicas de construção, um custo elevadíssimo que era totalmente incompatível com o orçamento aprovado.

No verão de 1961, Utzon estava perto do limiar do desespero e decidiu desmantelar o modelo de perspex das conchas. Contudo, ao arrumar as conchas, descobriu que elas se ajustavam quase perfeitamente umas dentro das outras, o que apenas seria possível se as diferentes conchas tivessem a mesma curvatura em todos os pontos. Ora a superfície que possui a mesma curvatura em todos os pontos é, obviamente, a esfera, o que levou Utzon a pensar que talvez fosse possível modelar as suas conchas como triângulos "cortados" na superfície de uma esfera. Embora esta modelação não fosse exactamente idêntica aos desenhos originais, tinha a vantagem de ser calculável em computadores e, ainda mais importante, de permitir a sua construção económica. A colaboração da Ove Arup foi crucial para que a idéia de Utzon pudesse passar da teoria à prática mas o rasgo de génio que permitiu resolver os problemas de construção é, tal como o desenho origi-

Figura 7.5: Placa comemorativa que explica a idéia de Utzon para a modelação das conchas. Fotografia de Matt Prebble, Reino Unido, Dezembro 2006.

nal, de Utzon. Esta idéia de Utzon está explicada num modelo de bronze colocado junto ao edifício da Ópera, tal como se pode ver na Figura 7.5.

Infelizmente, os atrasos na construção, bem como os custos elevados que se estavam a acumular, levaram o governo a questionar a decisão política de construir a Ópera e forçaram Utzon a demitir-se quando ainda faltava a construção dos interiores. Utzon ficou destroçado e abandonou a Austrália em 1966 para nunca mais voltar. Contra a vontade da maioria dos arquitectos, a sua obra-prima foi completada por Peter Hall e inaugurada em 1973 sem que se fizesse uma única referência a Utzon. Infelizmente, o trabalho de Peter Hall não esteve ao mesmo nível do de Utzon e o contraste entre os magníficos exteriores e os banais interiores levou a que a obra fosse considerada uma "semi-obra-prima."[1]

Nesta secção, vamos modelar as conchas da Ópera de Sydney seguindo exactamente a mesma solução proposta por Utzon. Todas as conchas do edifício serão modeladas por triângulos esféricos obtidos através de três cortes numa esfera. A Figura 7.6 mostra duas esferas de igual raio onde

[1] Apesar do drama pessoal de Utzon, esta história acabou por ter um final feliz: trinta anos mais tarde, o governo Australiano remodelou a Ópera de Sydney para a transformar na verdadeira obra-prima que ela merecia ser e conseguiu que Utzon, que nunca chegou a ver a sua obra acabada, aceitasse dirigir os trabalhos que visavam devolver-lhe o aspecto original.

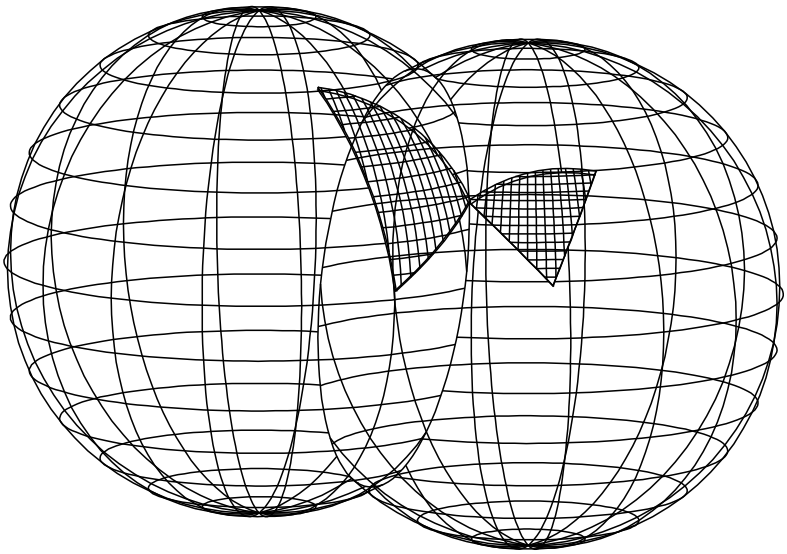


Figura 7.6: Duas das meia-conchas que constituem a Ópera de Sydney sobrepostas às esferas de onde foram obtidas por uma sucessão de cortes.

cortámos um triângulo em cada uma, de forma a obter duas das meias-conchas que constituem a Ópera de Sydney.

Para definir os cortes podemos considerar que o edifício irá ficar alinhado numa paralela ao eixo Y, pelo que o eixo de simetria corresponde a um plano de corte cuja normal é o eixo X, tal como se visualiza na Figura 7.7. Os dois restantes planos de corte terão normais determinadas de modo a aproximar, tão rigorosamente quanto nos é possível, a volumetria imaginada por Utzon. Como é também visível na Figura 7.7, cada concha é extraída de uma esfera com raio fixo mas que é centrada em diferentes pontos. Assim, para modelar estas meias-conchas vamos definir uma função que, a partir do centro p da esfera de raio r e das normais n_1 e n_2 dos planos de corte, produz uma casca esférica de espessura e com a forma pretendida.

```
meia_concha(p, r, e, n1, n2) =
  move(slice(slice(slice(subtraction(sphere(u0(), r),
                                     sphere(u0(), r - e)),
                         u0(), n2),
                  u0(), n1),
             x(-p.x), -vx()),
       p - u0())
```

A título de exemplo, a Figura 7.8 mostra uma meia concha produzida pela avaliação da expressão:

```
meia_concha(xyz(-45, 0, 0), 75, 2, vsph(1, 2.0, 4.6), vsph(1, 1.9, 2.6))
```

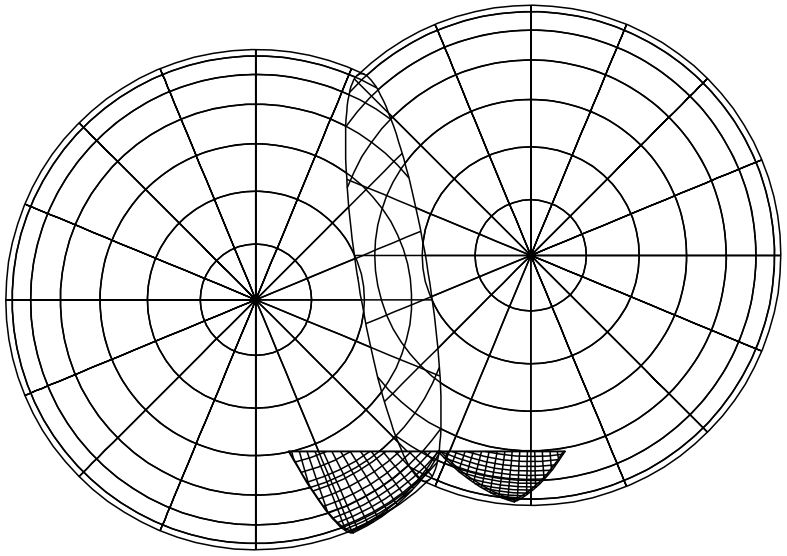


Figura 7.7: Vista de topo de duas das meia-conchas que constituem a Ópera de Sydney sobrepostas às esferas de onde foram obtidas por uma sucessão de cortes.

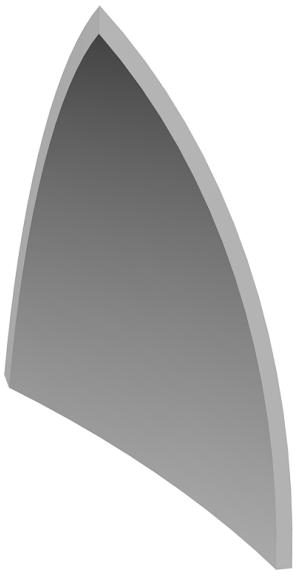


Figura 7.8: Uma meia concha da Ópera de Sydney.

Figura 7.9: Uma concha da Ópera de Sydney.

Para fazer uma concha completa temos apenas de aplicar uma reflexão à meia-concha, segundo o plano de corte vertical. Para juntar as duas partes, temos de fazer uma união, o que sugere a criação de uma função que combine a união com a reflexão:[2]

```
union_mirror(s, p, v) =
  union(s, mirror(s, p, v))
```

Usando esta operação, podemos agora definir:

```
concha(p, r, e, n1, n2) =
  union_mirror(meia_concha(p, r, e, n1, n2),
               u0(),
               -vx())
```

A Figura 7.9 mostra a concha produzida pela avaliação da expressão:

```
concha(xyz(-45, 0, 0), 75, 2, vsph(1, 2.0, 4.6), vsph(1, 1.9, 2.6))
```

Para definir o conjunto de conchas de um edifício vamos usar valores que aproximem as conchas produzidas do desenho original de Utzon:

[2]Na realidade, esta operação já existe pré-definida em Khepri.

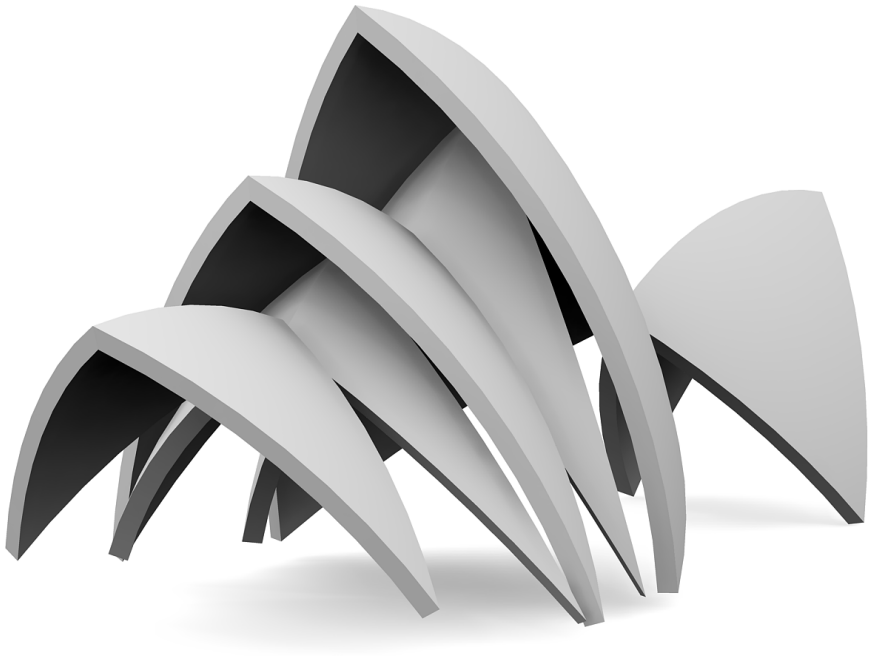


Figura 7.10: Uma fileira de conchas da Ópera de Sydney.

```
conchas_sydney() =
  union([concha(xyz(-45.01, 0.0, 0.0), 75, 2,
                vsph(1, 1.9701, 4.5693), vsph(1, 1.9125, 2.5569)),
         concha(xyz(-38.92, -13.41, -18.85), 75, 2,
                vsph(1, 1.9314, 4.5902), vsph(1, 1.7495, 1.9984)),
         concha(xyz(-38.69, -23.04, -29.89), 75, 2,
                vsph(1, 1.9324, 4.3982), vsph(1, 1.5177, 1.9373)),
         concha(xyz(-58.16, 81.63, -14.32), 75, 2,
                vsph(1, 1.6921, 3.9828), sph(1, 1.4156, 1.9618)),
         concha(xyz(-32.0, 73.0, -5.0), 75, 2,
                vsph(1, 0.91, 4.1888), vsph(1, 0.8727, 1.3439)),
         concha(xyz(-33.0, 44.0, -20.0), 75, 2,
                vsph(1, 1.27, 4.1015), vsph(1, 1.1554, 1.2217))])
```

A invocação desta função produz a fileira de conchas que apresentamos na Figura 7.10.

Para facilitar o posicionamento do edifício, vamos ainda incluir uma rotação em torno do eixo Z, uma escala e uma translação finais aplicadas a cada conjunto de conchas. Uma vez que o edifício possui dois conjuntos de conchas, vamos denominar esta função de `meia_opera_sydney`:

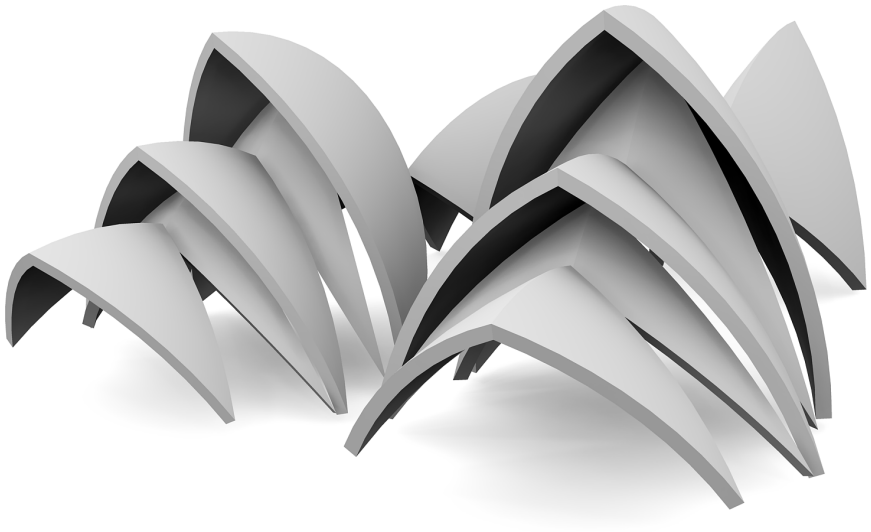


Figura 7.11: A modelação completa das conchas da Ópera de Sydney.

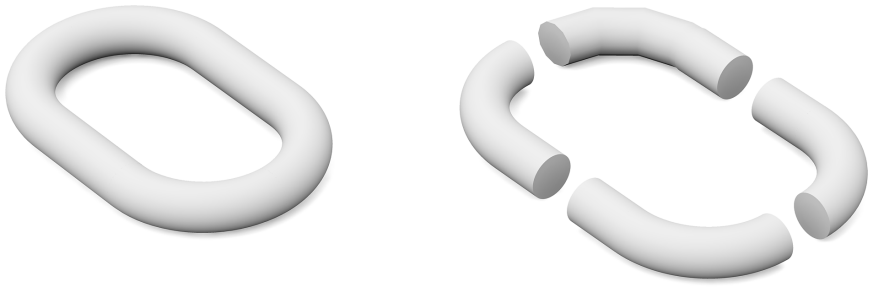
```
meia_opera_sydney(rot_z, esc, trans_x) =
  move(scale(rotate(conchas_sydney(),
                    rot_z),
             esc),
       vx(trans_x))
```

Finalmente, podemos modelar a ópera completa fazendo um edifício formado por um conjunto de conchas à escala 1.0 e um segundo edifício como uma versão reduzida, rodada e deslocada do primeiro. A escala usada por Utzon era de 80% e os ângulos de rotação e as translações que vamos usar são os que permitem que haja elevada semelhança com o edifício real tal como ele hoje se encontra:

```
opera_sydney() =
  begin
    meia_opera_sydney(0.1964, 1.0, 43)
    meia_opera_sydney(-0.1964, 0.8, -15)
  end
```

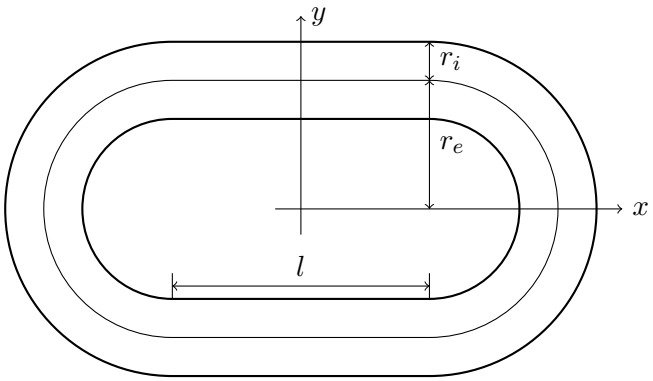
O resultado final da modelação das conchas encontra-se na Figura 7.11.

Exercício 7.6.1 Defina uma função que cria um *elo* de uma *corrente*, tal como se visualiza à esquerda da seguinte imagem.

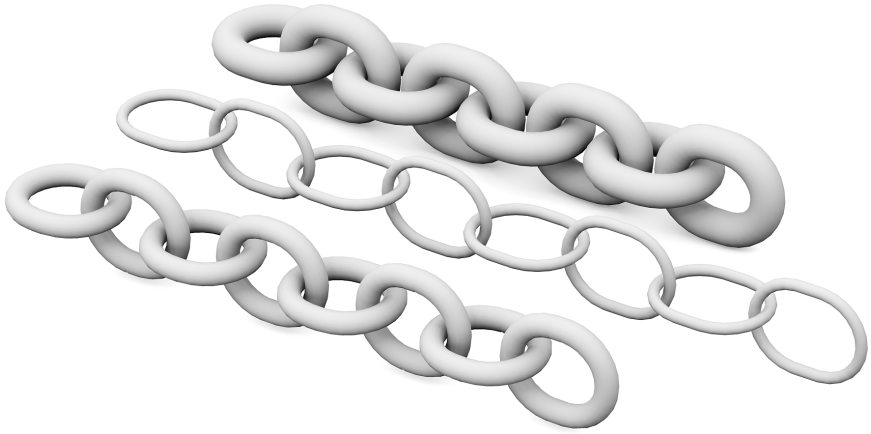


Para simplificar a modelação, considere o elo como decomponível em *quartos* de elo, tal como se visualiza à direita da imagem anterior. Deste modo, bastará definir uma função que cria um quarto de um elo (à custa de um quarto de um toro e de meio cilindro) e aplicar-lhe uma dupla reflexão nos eixos X e Y para compor o elo completo.

A função deverá receber, como parâmetros, o raio do elo r_e, o raio do "arame" r_i e o comprimento l entre semi-círculos, tal como se apresenta no seguinte esquema.

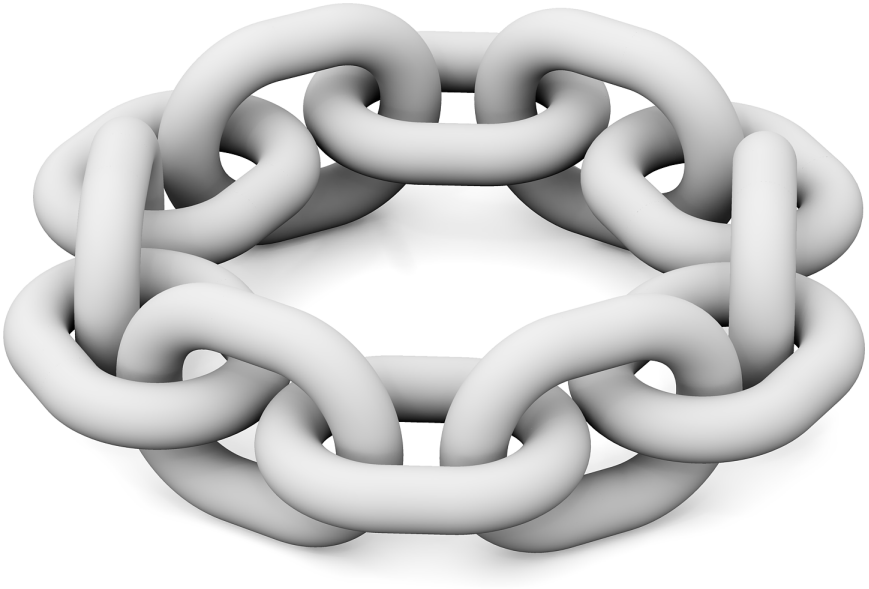


Exercício 7.6.2 Defina uma função capaz de criar *correntes* como as que se apresentam em seguida:



Note que, à medida que se vai construindo a corrente, os elos vão sofrendo sucessivas rotações em torno do eixo X.

Exercício 7.6.3 Defina uma função capaz de criar correntes fechadas como a que se apresenta em seguida:



Note que, à medida que se vai construindo a corrente, os elos vão sofrendo sucessivas rotações em torno do eixo X.

Capítulo 8

Ordem Superior

8.1 Introdução

Temos demonstrado como a linguagem Julia nos permite, através da definição de funções apropriadas, gerarmos as formas arquitecturais que temos em mente. Uma das vantagens da definição dessas funções é que muitas dessas formas arquitecturais ficam *parametrizadas*, permitindo-nos facilmente experimentar variações até atingirmos os valores numéricos dos parâmetros que nos satisfazem.

Apesar da "infinita" variabilidade que os parâmetros nos permitem, ainda assim existirão sempre limites naquilo que conseguimos fazer *apenas* com parâmetros numéricos e, para termos um grau superlativo de variabilidade, teremos de fazer variar as próprias funções.

Nesta secção vamos discutir funções de *ordem superior*. Uma função de ordem superior é uma função que recebe funções como argumentos ou que produz funções como resultados. À parte esta particularidade, uma função de ordem superior não tem nada de diferente de outra função qualquer.

8.2 Fachadas Curvilíneas

Para motivar a discussão vamos considerar um problema simples: pretendemos idealizar um edifício em que três dos seus lados são planos e o quarto—a fachada—é uma superfície vertical curvilínea. Para começar, podemos considerar que essa superfície curvilínea é de forma sinusoidal.

Tal como vimos na secção 6.6.1, uma sinusóide é determinada por um conjunto de parâmetros, como sejam, a *amplitude* a, o número de ciclos por unidade de comprimento ω e a *fase* ϕ da curva em relação ao eixo Y. Com estes parâmetros a equação da curva sinusóide fica com a forma:

$$y(x) = a\sin(\omega x + \phi)$$

Para calcularmos os pontos de uma sinusóide podemos usar a função `pontos_sinusoide` que computa uma lista de coordenadas (x, y) correspondentes à evolução da sinusóide entre os limites de um intervalo $[x_0, x_1]$, com um incremento Δ_x:

```
pontos_sinusoide(p, a, omega, fi, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p+vy(sinusoide(a, omega, fi, x0)),
     pontos_sinusoide(p+vx(dx), a, omega, fi, x0+dx, x1, dx)...]
  end
```

A função anterior gera uma lista com pontos da fachada. Para modelarmos o resto do edifício precisamos de unir esses pontos de modo a formar uma curva e precisamos de juntar a essa curva os três lados rectilineos de modo a formar uma região correspondente à planta do edifício. Para isso, vamos unir os pontos com uma *spline* e vamos formar uma região entre ela e as rectas que delimitam os outros três lados do edifício. O comprimento l_x do edifício será dado pela distância horizontal entre os dois extremos da curva da fachada. Já a largura l_y e a altura l_z terão de ser parâmetros. Assim, temos:

```
edificio(pontos, ly, lz) =
  let p0 = pontos[1],
      p1 = pontos[end],
      lx = p1.x-p0.x
    extrusion(surface(spline(pontos),
                      line(p0,
                           p0+vxy(0, ly),
                           p0+vxy(lx, ly),
                           p1)),
              lz)
  end
```

Para visualizarmos as capacidades da função anterior podemos construir vários edifícios usando valores diferentes dos parâmetros da sinusóide. Nas seguintes expressões considerámos duas filas de edifícios, todos com 15 metros de comprimento, 10 metros de largura e 20 ou 30 metros de altura, consoante a fila. A Figura 8.1 mostra esses edifícios para diferentes valores dos parâmetros `a`, `omega` e `fi`:

```
edificio(pontos_sinusoide(xy( 0,  0), 0.75, 0.5, 0, 0, 15, 0.4), 10, 20)
edificio(pontos_sinusoide(xy(25,  0), 0.55, 1.0, 0, 0, 15, 0.2), 10, 20)
edificio(pontos_sinusoide(xy(50,  0), 0.25, 2.0, 0, 0, 15, 0.1), 10, 20)
edificio(pontos_sinusoide(xy( 0, 20), 0.95, 1.5, 0, 1, 15, 0.4), 10, 30)
edificio(pontos_sinusoide(xy(25, 20), 0.85, 0.2, 0, 0, 15, 0.2), 10, 30)
edificio(pontos_sinusoide(xy(50, 20), 0.35, 1.0, 0, 1, 15, 0.1), 10, 30)
```

Infelizmente, embora a função `pontos_sinusoide` seja muito útil para modelar uma infinidade de diferentes edifícios de fachada sinusoidal, é to-

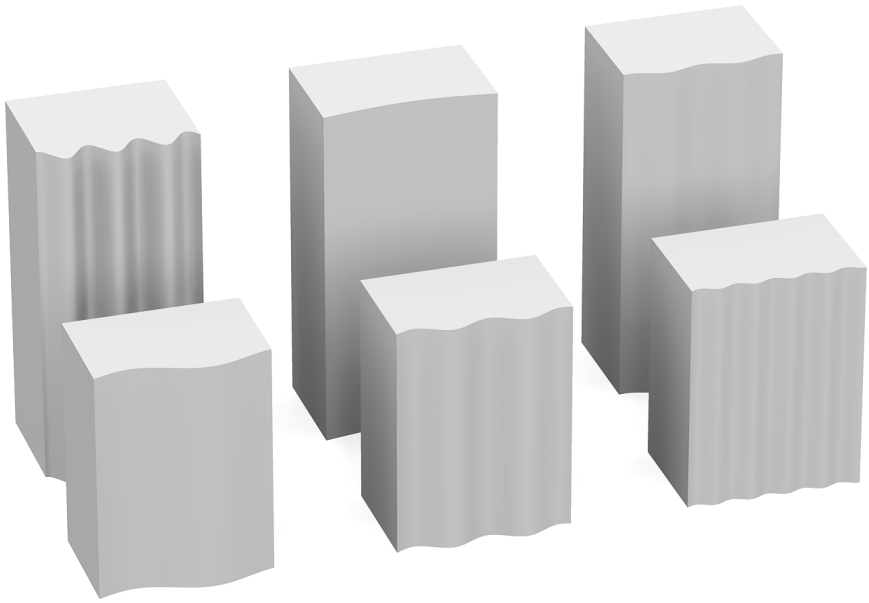


Figura 8.1: Urbanização de edifícios cuja fachada corresponde a paredes sinusoidais com parâmetros diferentes.

talmente inútil para modelar edifícios cuja fachada segue uma parábola, ou um logarítmo, ou uma exponencial ou, na verdade, qualquer outra curva que não seja reduzível a uma sinusóide. De facto, embora o ajuste dos parâmetros nos permita a infinita variabilidade do edifício modelado, ainda assim, ele será sempre um caso particular de uma fachada sinusoidal.

É claro que nada nos impede de definir outras funções para a modelação de fachadas diferentes. Imaginemos, por exemplo, que pretendemos uma fachada com a curvatura de uma parábola. A parábola com vértice em (x_v, y_v) e foco em $(x_v, y_v + d)$, sendo d a distância do vértice ao foco, define-se pela seguinte equação:

$$(x - x_v)^2 = 4d(y - y_v)$$

ou seja

$$y = \frac{(x - x_v)^2}{4d} + y_v$$

Em Julia, esta função define-se como:

```
parabola(xv, yv, d, x) =
  (x-xv)^2/(4*d)+yv
```

Para gerarmos os pontos da parábola temos, como de costume, de iterar ao longo de um intervalo $[x_0, x_1]$ com um incremento Δ_x:

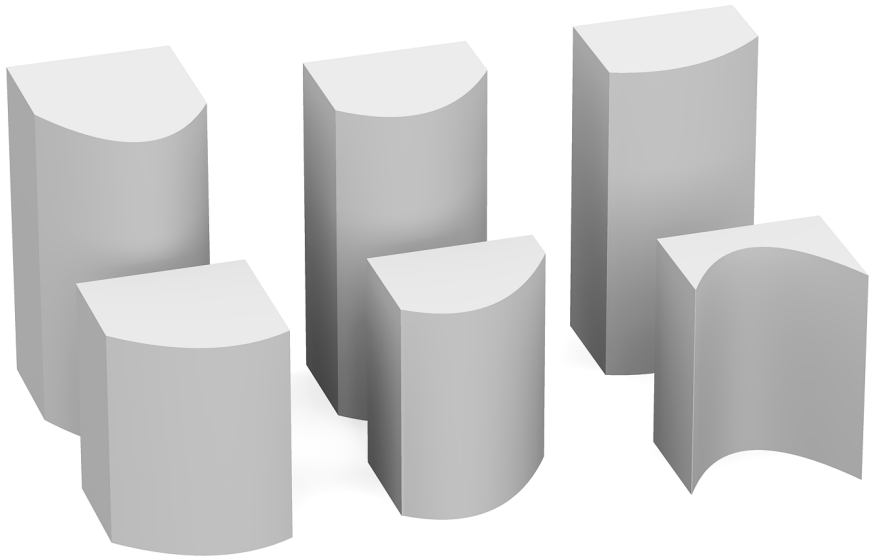


Figura 8.2: Urbanização de edifícios cuja fachada corresponde a paredes paraboloidais com parâmetros diferentes.

```
pontos_parabola(p, xv, yv, d, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p+vy(parabola(xv, yv, d, x0)),
     pontos_parabola(p+vx(dx), xv, yv, d, x0+dx, x1, dx)...]
  end
```

As seguintes expressões testam esta função com valores diferentes dos vários parâmetros:

```
edificio(pontos_parabola(xy( 0,  0), 10, 0, 5, 0, 15, 0.5), 10, 20)
edificio(pontos_parabola(xy(25,  0), 5, 0, 3, 0, 15, 0.2), 10, 20)
edificio(pontos_parabola(xy(50,  0), 7, 1, -2, 0, 15, 0.1), 10, 20)
edificio(pontos_parabola(xy( 0, 20), 8, 0, 2, 0, 15, 0.4), 10, 30)
edificio(pontos_parabola(xy(25, 20), 6, -2, 3, 0, 15, 0.2), 10, 30)
edificio(pontos_parabola(xy(50, 20), 5, 0, 6, 0, 15, 0.1), 10, 30)
```

gerando a "urbanização" representada na Figura 8.2.

Mais uma vez, a parametrização da função `pontos_parabola` permite-nos gerar uma infinidade de edifícios cuja fachada tem a curvatura de uma parábola, mas serão sempre edifícios diferentes daqueles que podemos gerar com a função `pontos_sinusoide`. Embora o processo de modelação empregue seja absolutamente idêntico nos dois casos, as funções de base empregues—`sinusoide` *vs* `parabola`—produzirão sempre curvas diferentes, independentemente dos parâmetros particulares utilizados para cada uma.

É claro que se agora pretendermos criar edifícios com fachada em curva determinada por uma função $f(x)$ que não seja nem uma sinusóide nem uma parábola, não poderemos empregar as funções anteriores. Primeiro, teremos de definir a função `f` que implementa essa curva e, segundo, teremos de definir a função `pontos_f` que gera os pontos de `f` num intervalo. Esta segunda parte parece excessivamente repetitiva e, de facto, é. Basta observarmos as funções `pontos_sinusoide` e `pontos_parabola` para constatarmos que são muito idênticas entre si. Logicamente, o mesmo irá acontecer com a nova função `pontos_f`.

Esta repetição leva-nos a pensar na possibilidade de abstrairmos os processos em questão de modo a que não sejamos obrigados a repetir definições que apenas variam em pequenos detalhes. Para isso, podemos começar por comparar as definições das funções e perceber onde estão as diferenças. Por exemplo, no caso das funções `pontos_sinusoide` e `pontos_parabola`, temos:

```
pontos_sinusoide(p, a, omega, fi, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p + vy(sinusoide(a, omega, fi, x0)),
     pontos_sinusoide(p + vx(dx),
                      a, omega, fi,
                      x0 + dx, x1, dx)...]
  end
```

```
pontos_parabola(p, xv, yv, d, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p + vy(parabola(xv, yv, d, x0)),
     pontos_parabola(p + vx(dx),
                      xv, yv, d,
                      x0 + dx, x1, dx)...]
  end
```

Assinalámos a itálico as diferenças entre as duas funções e, como podemos ver, elas resumem-se aos nomes dos parâmetros e ao nome da função que é invocada: `sinusoide` no primeiro caso, `parabola` no segundo. Sendo os nomes dos parâmetros de uma função visíveis apenas pelo corpo dessa função (i.e., são *locais* à função), podemos renomeá-los desde que o façamos de forma consistente. Isto quer dizer que as seguintes definições teriam exactamente o mesmo comportamento:

```
pontos_sinusoide(p, α, β, γ, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p + vy(sinusoide(α, β, γ, x0)),
     pontos_sinusoide(p + vx(dx),
                      α, β, γ,
                      x0 + dx, x1, dx)...]
  end

pontos_parabola(p, α, β, γ, x0, x1, dx) =
  if x0 > x1
    []
  else:
    [p + vy(parabola(α, β, γ, x0)),
     pontos_parabola(p + vx(dx),
                     α, β, γ,
                     x0 + dx, x1, dx)...]
  end
```

Torna-se agora absolutamente óbvio que a única diferença entre as duas funções está numa invocação que elas fazem, que é `sinusoide` num caso e `parabola` no outro.

Ora quando duas funções diferem apenas num nome que usam internamente, é sempre possível definir uma terceira função que as generaliza, simplesmente transformando esse nome num parâmetro adicional.

Imaginemos então que fazemos a seguinte definição, onde `f` é esse parâmetro adicional:

```
pontos_funcao(p, f, alfa, beta, gama, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p+vy(f(alfa, beta, gama, x0)),
     pontos_funcao(p+vx(dx), f, alfa, beta, gama, x0+dx, x1, dx)...]
  end
```

O aspecto inovador da função `pontos_funcao` está no facto de receber uma função como argumento, função essa que ficará associada ao parâmetro `f`. Quando, no corpo da função `pontos_funcao`, é feita uma invocação à função `f`, estamos, na verdade, a fazer uma invocação à função que tiver sido passada como argumento para o parâmetro `f`.

Desta forma, a expressão

```
pontos_sinusoide(p, a, omega, fi, 0, lx, dx)
```

é absolutamente idêntica a

```
pontos_funcao(p, sinusoide, a, omega, fi, 0, lx, dx)
```

Do mesmo modo, a expressão

```
pontos_parabola(p, xv, yv, d, 0, lx, dx)
```

é absolutamente idêntica a

```
pontos_funcao(p, parabola, xv, yv, d, 0, lx, dx)
```

Mais importante que o facto de podermos dispensar as funções `pontos_sinusoide` e `pontos_parabola` é o facto de, agora, podermos modelar edifícios cuja fachada segue qualquer outra curva que pretendamos. Por exemplo, a função que descreve o movimento oscilatório amortecido tem como definição:

$$y = ae^{-bx}\sin(cx)$$

ou, em Julia:

```
oscilatorio_amortecido(a, b, c, x) =
  a*exp(-(b*x))*sin(c*x)
```

Usando a função `pontos_funcao` é agora possível definir um edifício cuja fachada segue a curva do movimento oscilatório amortecido. Por exemplo, a seguinte expressão

```
edificio(pontos_funcao(xy(0, 0),
                       oscilatorio_amortecido,
                       -5, 0.1, 1,
                       0, 40, 0.4),
         10, 20)
```

produz, como resultado da sua avaliação, o edifício representado na Figura 8.3.

8.3 Funções de Ordem Superior

A função `pontos_funcao` é um exemplo de uma categoria de funções que designamos de *ordem superior*. Uma função de ordem superior é uma função que recebe outras funções como argumentos ou que produz outras funções como resultados. No caso da função `pontos_funcao` ela recebe uma função de um parâmetro que irá ser invocada para sucessivos pontos de um intervalo, produzindo a lista de coordenadas encontradas.

As funções de ordem superior são importantes ferramentas de abstracção. Elas permitem abstrair computações em que há uma parte da computação que é comum e uma (ou mais partes) que variam de caso para caso. Usando uma função de ordem superior, apenas implementamos a parte da computação que é comum, deixando as partes variáveis da computação para serem implementadas como funções que serão passadas nos parâmetros da função de ordem superior.

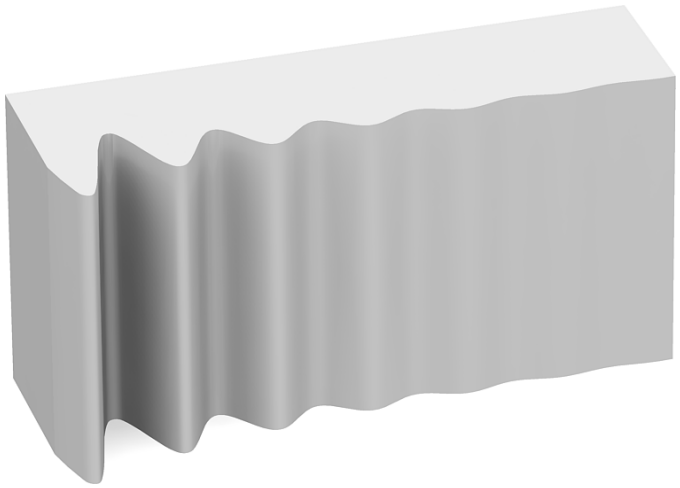


Figura 8.3: Um edifício com uma fachada que acompanha a curva do movimento oscilatório amortecido.

O conceito de função de ordem superior existe há já bastante tempo em matemática, embora raramente seja explicitamente mencionado. Consideremos, a título de exemplo, uma função que soma os logaritmos de todos os inteiros entre a e b, $\sum_{i=a}^{b} \log i$:

```
soma_logaritmos(a, b) =
  if a > b
    0
  else
    log(a) + soma_logaritmos(a + 1, b)
  end


julia> soma_logaritmos(1, 4)
3.17805383035
```

Consideremos agora uma outra função que soma as raizes quadradas de todos os inteiros entre a e b, $\sum_{i=a}^{b} \sqrt{i}$:

```
soma_raizes_quadradas(a, b) =
  if a > b
    0
  else
    sqrt(a) + soma_raizes_quadradas(a + 1, b)
  end

julia> soma_raizes_quadradas(1, 4)
6.14626436994
```

A mera observação da definição das funções mostra que elas possuem uma estrutura comum que se caracteriza pela seguinte definição:

```
soma_???(a, b) =
  if a > b
    0
  else
    ???(a) + soma_???(a + 1, b)
  end
```

Ora esta definição não é mais que uma soma de uma expressão matemática entre dois limites, i.e., um somatório $\sum_{i=a}^{b} f(i)$. O somatório é uma abstracção matemática para uma soma de números descritos por uma expressão matemática relativa ao índice do somatório que varia desde o limite inferior até ao limite superior. A expressão matemática é, portanto, função do índice do somatório.

O símbolo `???` representa a expressão matemática a realizar dentro do somatório e que vamos simplesmente transformar num parâmetro, escrevendo:

```
somatorio(f, a, b) =
  if a > b
    0
  else
    f(a) + somatorio(f, a + 1, b)
  end
```

Podemos agora facilmente avaliar diferentes somatórios:

```
julia> somatorio(log, 1, 4)
3.17805383035
julia> somatorio(sqrt, 1, 4)
6.14626436994
```

Por receber uma função como argumento, o somatório é uma função de ordem superior. A derivada de uma função é outro exemplo. A derivada f' de uma função f é uma função que recebe outra função f como argumento e produz outra função—a função derivada de f—como resultado.

Em geral, as funções de ordem superior surgem porque existem duas ou mais funções com uma estrutura semelhante. De facto, a repetição de um padrão ao longo de duas ou mais funções é um forte indicador da necessidade de uma função de ordem superior.

8.4 Funções Anónimas

A possibilidade de usarmos funções de ordem superior abre um leque enorme de novas aplicações. Por exemplo, se quiseremos calcular os somatórios

$$\sum_{i=1}^{10} i^2 + 3i$$

$$\sum_{i=1}^{100} i^3 + 2i^2 + 5i$$

não temos de fazer mais do que definir as funções $f_1(x) = x^2 + 3x$ e $f_2(x) = x^3 + 2x^2 + 5x$ e usá-las como argumento da função `somatorio`:

```
f1(x) =
  x*x + 3*x

f2(x) =
  x*x*x + 2*x*x + 5*x

julia> somatorio(f1, 1, 10)
550
julia> somatorio(f2, 1, 100)
26204450
```

Como se vê, podemos agora facilmente calcular somatórios de diferentes funções. No entanto, há um aspecto negativo a salientar: se, para cada somatório que pretendermos calcular, tivermos de definir a função em questão, iremos "poluir" o nosso ambiente Julia com inúmeras definições de funções cuja utilidade será, no máximo, servirem de argumento para o cálculo do somatório correspondente. Uma vez que cada função tem de ser associada a um nome, teremos também de inventar nomes para todas elas, o que poderá ser difícil, em particular, quando sabemos que estas funções não servem para mais nada. Nos dois últimos exemplos este problema já era visível: os nomes `f1` e `f2` apenas revelam a dificuldade em arranjar nomes mais expressivos.

Para resolvermos este problema, convém analisar mais em pormenor o conceito de definição de função. Até agora temos referido que, para definirmos uma função, igualamos o nome e os parâmetros da função ao corpo da função. Por exemplo, para a função $x^2 = x \times x$, escrevemos:

```
sqr(x) =
  x*x
```

Vimos que após a definição anterior, o símbolo `sqr` fica associado a uma função:

```
julia> sqr
sqr (generic function with 1 method)
```

No entanto, vimos na secção 1.14 que para definirmos constantes igualamos o nome que queremos definir à expressão cujo valor pretendemos associar a esse nome. Por exemplo:

```
fi = (1 + sqrt(5))/2
```

Vimos também que após a definição anterior, o símbolo `fi` fica associado a um valor:

```
julia> fi
1.618033988749895
```

Da análise destes exemplos podemos concluir que a única diferença entre uma definição de função e uma definição de constante se resume à forma como se obtém o *valor* que é definido num caso e noutro. Na definição de constantes, o valor é o resultado da avaliação de uma expressão. Na definição de função, o valor é uma função que se constrói a partir de uma descrição de uma lista de parâmetros e de um corpo de função. Isto leva-nos a pensar que se fosse possível termos uma expressão cuja avaliação produzisse uma função, então seria possível definir funções como se estivéssemos a definir constantes. Por exemplo, no caso da função `sqr`, se for λ essa expressão, deveria ser possível escrever `sqr` = λ.

Se a expressão `sqr` = λ associa ao símbolo `sqr` a função criada pela avaliação da expressão λ, o que será então a avaliação da expressão λ? Obviamente, tem de ser uma função, mas, como não está ainda associada a nenhum nome, é o que se denomina por função sem nome ou *função anónima*. Note-se que uma função anónima é uma função como qualquer outra, mas tem a particularidade de não estar associada a nenhum nome.

Falta-nos ainda conhecer a forma das expressões λ. A sua sintaxe é a seguinte:[1]

(parâmetro$_1$, ..., parâmetro$_n$) -> expressão

No caso (frequente) de a função só ter um parâmetro, a sintaxe pode ser simplificada para:

parâmetro -> expressão

Qualquer expressão com uma das formas anteriores, quando avaliada, produz uma função anónima. Estas expressões designam-se, em Julia, por *expressões lambda*. Reparemos agora na seguinte interacção:

```
julia> x -> x*x
#9 (generic function with 1 method)
julia> sqr = x -> x*x
#11 (generic function with 1 method)
julia> sqr(3)
9
```

Como se vê, a avaliação da expressão lambda devolve algo cuja representação externa nos indica que foi criado um procedimento. Quando associamos a função anónima a um nome, esse nome passa a designar a função,

[1] A utilização que fizemos do símbolo λ não foi inocente. Ele deriva das origens matemáticas do conceito de função anónima: o *cálculo* λ, um modelo matemático para funções *computáveis*, i.e., funções cuja invocação se pode avaliar mecanicamente.

podendo ser usado como se tivesse sido definido originalmente como função. Na prática, a definição de funções via expressões lambda é equivalente à definição usual de funções. Mais especificamente,

nome = (*parâmetro*$_1$, ..., *parâmetro*$_n$) -> *expressão*

é equivalente a:

nome(*parâmetro*$_1$, ..., *parâmetro*$_n$) = *expressão*

Esta equivalência é facilmente testável através da redefinição de funções que, agora, podem ser definidas como a criação de uma associação entre um nome e uma função anónima. O exemplo anterior demonstrou essa possibilidade para a função `sqr` mas ela existe para muitas outras funções. Por exemplo, a função que calcula a área de um círculo tanto pode ser definida por:

```
area_circulo(raio) =
  pi*raio^2
```

como pode ser definida por:

```
area_circulo = raio -> pi*raio^2
```

Embora do ponto de vista sintático, a forma `f = (...) -> ...` seja equivalente a `f(...) = ...`, existe uma diferença semântica com implicações importantes: a primeira forma avalia a expressão que determina o valor a atribuir, o que faz com que a função criada seja o resultado da avaliação de uma expressão. Iremos ver que isto permite formas sofisticadas de definição de funções.

A utilização de funções anónimas tem ainda outra vantagem que se torna evidente quando analisamos a função `pontos_funcao`:

```
pontos_funcao(p, f, alfa, beta, gama, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p+vy(f(alfa, beta, gama, x0)),
     pontos_funcao(p+vx(dx), f, alfa, beta, gama, x0+dx, x1, dx)...]
  end
```

Como vimos, para gerarmos uma lista de pontos de, por exemplo, uma sinusóide, podemos invocar esta função da seguinte forma:

```
pontos_funcao(xy(0, 0),
              sinusoide,
              0.75, 0.5, 0,
              0, 15, 0.4)
```

Notemos que, por ter sido definida como uma generalização das funções `pontos_sinusoide` e `pontos_parabola`, a função `pontos_funcao`,

para além de ter introduzido a função `f` como parâmetro, generalizou também os parâmetros `a`, `omega` e `fi` da função `pontos_sinusoide` e `xv`, `yv` e `d` da função `pontos_parabola`, chamando-lhes, respectivamente, `alfa`, `beta` e `gama`. Acontece que estes parâmetros nunca são alterados durantes as invocações recursivas realizadas pela função `pontos_funcao`. Na verdade, a função passa esses parâmetros de invocação recursiva em invocação recursiva apenas porque eles são necessários para a invocação da função `f`. Imaginemos agora que, ao invés de passarmos esses parâmetros, eles estivessem associados à própria função `f` que tínhamos passado. Neste caso, poderíamos reescrever a função `pontos_funcao` na seguinte forma:

```
pontos_funcao(p, f, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p + vy(f(x0)),
     pontos_funcao(p + vx(dx), f, x0 + dx, x1, dx)...]
  end
```

Com esta nova definição, a invocação anterior teria de ser reescrita na forma:

```
pontos_funcao(xy(0, 0),
              x -> sinusoide(0.75, 0.5, 0, x),
              0, 15, 0.4)
```

Embora não pareça ser um ganho substancial, a reescrita da função `pontos_funcao` tem a vantagem considerável de permitir a utilização de quaisquer funções, independentemente do número de parâmetros destas: se tiverem apenas um parâmetro, podem ser usadas directamente, caso contrário, podemos "envolvê-las" com uma função anónima de um só parâmetro que invoca a função pretendida com todos os argumentos necessários. Isto faz com que a função `pontos_funcao` fique ainda mais genérica, tal como é visível nos seguintes exemplos:

```
spline(
  pontos_funcao(xy(0, 6),
                sin,
                0, 4*pi, 0.2))

spline(
  pontos_funcao(xy(0, 3),
                x -> -(x/10)^3,
                0, 4*pi, 0.5))

spline(
  pontos_funcao(xy(0, 0),
                x -> oscilatorio_amortecido(1.5, 0.4, 4, x),
                0, 4*pi, 0.05))
```

que geram as curvas representadas na Figura 8.4.

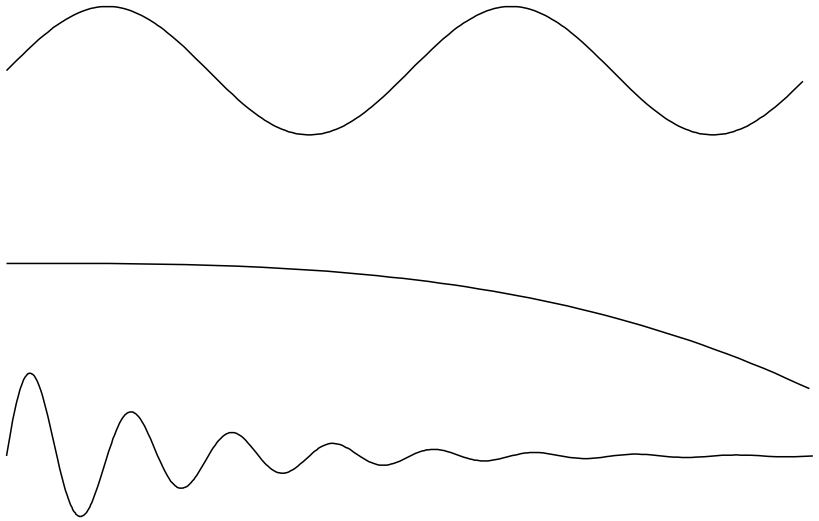


Figura 8.4: Curvas geradas pela função `pontos_funcao`. De cima para baixo, temos uma sinusóide, uma exponencial invertida e uma sinusóide amortecida.

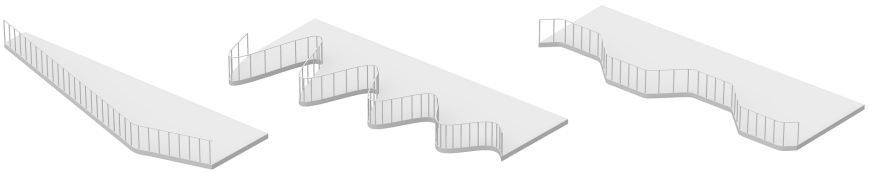
Embora a generalidade das funções de ordem superior permita que estas sejam usadas numa enorme variedade de situações, por vezes, é preferível encapsular um uso particular numa função mais facilmente reconhecível. Por exemplo, se um programa estiver sistematicamente a computar pontos de sinusóides, então é provável que esse programa fique mais legível se existir de facto a função `pontos_sinusoide`. Mas mesmo nesse caso, as funções de ordem superior vêem permitir que essa função seja definida de forma mais simples. Assim, ao invés de termos de escrever a habitual definição recursiva:

```
pontos_sinusoide(p, a, omega, fi, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p + vy(sinusoide(a, omega, fi, x0)),
     pontos_sinusoide(p + vx(dx), a, omega, fi,
                      x0 + dx, x1, dx)...]
  end
```

podemos passar a escrever:

```
pontos_sinusoide(p, a, omega, fi, x0, x1, dx) =
  pontos_funcao(p,
                x -> sinusoide(a, omega, fi, x),
                x0, x1, dx)
```

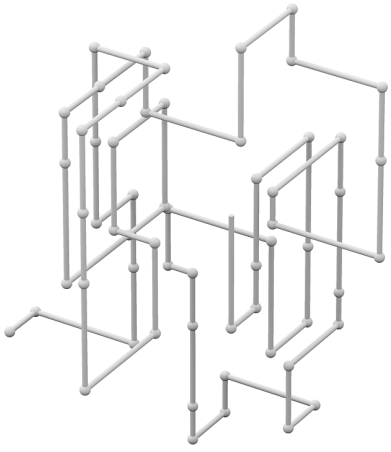
Exercício 8.4.1 Considere as varandas apresentadas na imagem seguinte:



As varandas são compostas por uma lage e uma guarda, sendo a guarda composta pelos prumos e pelo corrimão. Defina uma função `varanda` que, convenientemente parametrizada, é capaz de gerar não só as varandas apresentadas na imagem anterior mas também muitas outras. Para isso, a função `varanda` deverá receber não só os parâmetros geométricos da varanda (como seja a espessura da laje ou a altura do corrimão, etc.), mas também a função que determina a curva exterior da varanda.

Exercício 8.4.2 Defina as funções necessárias e escreva expressões Julia que reproduzem as varandas apresentadas na imagem anterior.

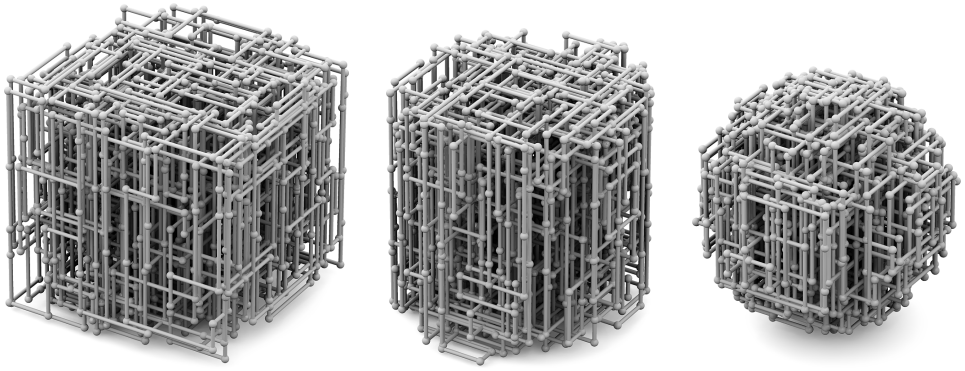
Exercício 8.4.3 Considere a imagem seguinte onde apresentamos uma sequência de tubos cilíndricos unidos por esferas que perfazem um caminho aleatório em que os troços do caminho são paralelos aos eixos coordenados. Note que os tubos têm um comprimento aleatório compreendido entre um comprimento máximo e 10% desse comprimento máximo e que as mudanças de direcção são também aleatórias mas nunca invertem a direcção imediatamente anterior. Note ainda que os tubos possuem um raio que é 2% do comprimento máximo do tubo e que as esferas possuem um raio que é 4% do comprimento máximo do tubo.



Defina a função `caminho_de_tubos` que, dados o ponto e a direcção inicial, o comprimento máximo de tubo e o número de tubos, contrói a sequência de tubos unidos por esferas que percorre um caminho aleatório.

Exercício 8.4.4 Muito embora um caminho possa ser aleatório, é possível limitá-lo no espaço de modo a que nunca ultrapasse uma determinada região. Esta limitação é visivel na imagem seguinte onde, da esquerda para a

direita, vemos a sequência de tubos a percorrer um caminho aleatório limitado a um cubo, um caminho aleatório limitado a um cilindro e, finalmente, um caminho aleatório limitado a uma esfera.



Defina a função `forma_de_tubos` como uma generalização da função `caminho_de_tubos` de modo a receber, para além dos parâmetros desta última, um parâmetro adicional que deverá ser um predicado que, dado um hipotético ponto para extensão de um caminho, indica se esse ponto está contido na região em questão. Se não estiver, o programa deve rejeitar esse ponto e gerar um novo.

Defina ainda as funções `cubo_de_tubos`, `cilindro_de_tubos`, e `esfera_de_tubos`, que possuem os mesmos parâmetros da função `caminho_de_tubos` e que usam a função `forma_de_tubos` com o predicado apropriado para a forma pretendida.

8.5 A Função Identidade

Vimos que a função `somatorio` implementa o clássico somatório empregue em álgebra $\sum_{i=a}^{b} f(i)$:

```
somatorio(f, a, b) =
  if a > b
    0
  else
    f(a) + somatorio(f, a + 1, b)
  end
```

Todos os somatórios podem ser calculados simplesmente especificando, por um lado, a função f que dá cada um dos termos do somatório e, por outro, os limites a e b desse somatório. Por exemplo, a expressão matemática $\sum_{i=1}^{10} \sqrt{i}$ é calculado pela correspondente expressão Julia:

```
somatorio(sqrt, 1, 10)
```

No caso de os termos do somatório serem computados por uma expressão para a qual o Julia não possui uma função pré-definida, a solução mais

imediata será empregar uma expressão lambda. Por exemplo, para calcularmos

$$\sum_{i=1}^{10} \frac{\sin i}{\sqrt{i}}$$

podemos escrever:

```
somatorio(i -> sin(i)/sqrt(i), 1, 10)
```

Em geral, a especificação da função f, que computa cada termo do somatório, não levanta quaisquer dificuldades mas existe um caso particular que pode causar alguma perplexidade e que, por isso, merece ser discutido. Consideremos o somatório $\sum_{i=1}^{10} i$. Qual será a expressão Julia que o calcula?

Neste último exemplo, não é inteiramente claro qual é a função f que está em jogo no cálculo do somatório, pois a observação da expressão matemática correspondente ao termo do somatório não permite descortinar qualquer função. E, no entanto, ela está lá. Para a tornarmos mais clara temos de nos recordar que a função `somatorio` calcula $\sum_{i=a}^{b} f(i)$, pelo que, neste último exemplo, os parâmetros em questão terão de ser $a = 1$, $b = 10$ e, finalmente, $f(i) = i$. É esta última função que é relativamente estranha: dado um argumento, ela limita-se a devolver esse argumento, sem realizar qualquer operação sobre ele. Matematicamente falando, esta função denomina-se *função identidade* e corresponde ao elemento neutro da composição de funções. A função identidade pode definir-se trivialmente em Julia a partir da sua própria definição matemática:

```
identidade(x) =
  x
```

Embora pareça ser uma função inútil, a função `identidade` é, na realidade, muito útil. Antes de mais, porque nos permite calcular o somatório $\sum_{i=1}^{10} i$ escrevendo apenas:

```
julia> somatorio(identidade, 1, 10)
55
```

Muitos outros problemas beneficiam igualmente da função identidade. Por exemplo, se definirmos a função que calcula o produtório $\prod$:

$$\prod_{i=a}^{b} f(i) = f(a) \cdot f(a+1) \cdot \cdots \cdot f(b-1) \cdot f(b)$$

```
produtorio(f, a, b) =
  if a > b
    1
  else
    f(a)*produtorio(f, a + 1, b)
  end
```

passa a ser possível definir a função factorial através do produtório:

$$n! = \prod_{i=1}^{n} i$$

```
factorial(n) =
  produtorio(identidade, 1, n)
```

Mais à frente iremos encontrar novos usos para a função identidade.

Exercício 8.5.1 Quer o somatório, quer o produtório podem ser vistos como casos especiais de uma outra abstracção ainda mais genérica, designada acumulatório. Nesta abstracção, são parâmetros: a operação de combinação dos elementos, a função a aplicar a cada um, o valor inicial, o limite inferior, a passagem para o elemento seguinte (designado o sucessor), e o limite superior. Defina esta função. Defina ainda o somatório e o produtório em termos de acumulatório.

Exercício 8.5.2 Sabe-se que a soma $\frac{8}{1\cdot3} + \frac{8}{5\cdot7} + \frac{8}{9\cdot11} + \cdots$ converge (muito lentamente) para π. Usando a função `acumulatorio` definida no exercício anterior, defina a função que calcula uma aproximação de π até ao n-ésimo termo da soma. Determine uma aproximação de π até ao termo 2000.

Exercício 8.5.3 A função `range` é capaz de gerar sucessões em progressão aritmética, i.e., sucessões em que existe uma diferença constante entre cada dois elementos. Pode ser necessário, contudo, produzir sucessões em que os elementos evoluem de forma diferente, por exemplo, em progressão geométrica.

Defina a função de ordem superior `sucessao` que recebe os limites a e b de um intervalo $[a, b[$ e ainda uma função f e que gera uma lista com todos os elementos x_i inferiores a b tais que $x_0 = a$ e $x_{i+i} = f(x_i)$. Por exemplo:

```
julia> sucessao(2, 600, x -> x*2)
[2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Exercício 8.5.4 Redefina a função `enumera` em termos da função `sucessao`.

8.6 A Função Restrição

Vimos nas secções anteriores vários exemplos de funções de ordem superior que recebiam funções como argumentos. Nesta secção iremos ver funções de ordem superior que produzem funções como resultado.

Comecemos por considerar o seguinte somatório: $\sum_{i=0}^{10} 2^i$. O valor desta expressão pode ser calculado em Julia através de:

```
somatorio(i -> 2^i, 0, 10)
```

É imediato constatarmos que a função 2^i representa um caso particular da função potência a^b. Mais especificamente, dizemos que a função 2^i é uma *restrição* da função a^b, em que o primeiro operando é sempre `2`. Do mesmo modo, podemos dizer que a função anónima `i -> 2^i` é uma restrição da função `^`, que passa a ter o primeiro operando fixo e igual a `2`.

Este tipo de restrição, em que usamos uma operação de dois operandos sempre com o mesmo primeiro operando, é um padrão que pode ser implementado usando uma função de ordem superior:

```
restricao(f, a) =
  x -> f(a, x)
```

O aspecto mais interessante da função `restricao` é que ela produz uma função como resultado. Usando esta função, podemos calcular o somatório $\sum_{i=0}^{10} 2^i$ sem termos de criar manualmente a função anónima:

```
somatorio(restricao(^, 2), 0, 10)
```

Naturalmente, podemos usar a função `restricao` para definir outras funções. Por exemplo, a função exponencial e^x, sendo $e = 2.718281828459045$ a base dos logaritmos Neperianos, é também um caso particular da função potência a^b. Assim, poderia ser definida à custa dela e da função restrição:

```
exponencial = restricao(^, 2.718281828459045)
```

podendo ser usada como qualquer outra função:

```
julia> exponencial(2)
7.3890560989306495
```

8.7 A Função Composição

Uma das mais úteis funções de ordem superior é a que permite realizar a *composição* de outras funções. Dadas duas funções f e g, a composição de f com g escreve-se $f \circ g$ e define-se como sendo a função

$$(f \circ g)(t) = f(g(t))$$

Esta forma de definir a composição de funções mostra que se pretende aplicar a função f ao resultado da aplicação da função g, mas não mostra que o que se está a definir é, na verdade, a função $\circ$. Para tornar esta definição mais evidente, é útil empregarmos a notação mais formal $\circ(f, g)$.

Nesta forma, já é mais óbvio que $\circ$ é uma função que recebe outras funções como argumentos. Só por isto, já podemos afirmar que $\circ$ é uma função de ordem superior mas, na verdade, ela faz mais do que receber duas funções como argumentos, ela também produz uma nova função como resultado que, para um dado argumento t, calcula $f(g(t))$. Uma vez que esta nova função não tem nome, a melhor forma de ser produzida é através de uma expressão `lambda`. A definição correcta da composição de funções é, então:

$$\circ(f, g) = \lambda t f(g(t))$$

Estamos agora em condições de definir esta função em Julia:

```
composicao(f, g) =
  t -> f(g(t))
```

Usando a função `composicao` podemos agora criar novas funções mais facilmente, sem termos de as definir explicitamente ou criar funções anónimas manualmente. Por exemplo, o somatório $\sum_{i=0}^{10} \sin \sqrt{i}$ pode ser obtido através de:

```
julia> somatorio(composicao(sin, sqrt), 0, 10)
6.0547478263584305
```

8.8 Funções de Ordem Superior sobre Listas

Vimos, aquando da introdução das listas como estrutura de dados, que o processamento das listas era facilmente realizado através da definição de funções recursivas. Na verdade, o comportamento destas funções era bastante estereotipado: as funções começavam por testar se a lista estava vazia e, caso não estivesse, processavam o primeiro elemento da lista, processando os restantes através de uma invocação recursiva.

Como acabámos de ver na secção anterior, quando algumas funções possuem um comportamento semelhante, é vantajoso abstraí-las numa função de ordem superior. É precisamente isso que vamos agora fazer através da definição de funções de ordem superior para três casos frequentes de processamento de listas: o mapeamento, a filtragem e a redução.

8.8.1 Mapeamento

Uma das operações mais úteis é aquela que transforma uma lista noutra lista através da aplicação de uma função a cada elemento da primeira lista. Por exemplo, dada uma lista de números, podemos estar interessados em produzir uma outra lista com os quadrados desses números. Neste caso, dizemos que estamos a *mapear* a função quadrado numa lista para produzir a lista dos quadrados. A definição da função apresenta o já típico padrão de recursão em listas:

```
sqr(x) =
  x*x

mapeia_quadrado(lista) =
  if lista == []
    []
  else
    [sqr(lista[1]), mapeia_quadrado(lista[2:end])...]
  end
```

Obviamente, estar a definir uma função apenas para mapear o quadrado é particularizar em excesso. Seria muito mais proveitoso definir uma função de ordem superior que mapeie qualquer função sobre uma lista. Felizmente, é fácil modificar a função anterior:

```
mapeia(f, lista) =
  if lista == []
    []
  else
    [f(lista[1]), mapeia(f, lista[2:end])...]
  end
```

Com esta função é trivial produzir, por exemplo, o quadrado de todos os números de 10 a 20:

```
julia> mapeia(sqr, enumera(10, 21, 1))
[100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]
```

A função `mapeia` já existe pré-definida em Julia com o nome `map`. A implementação providenciada em Julia permite ainda mapear sobre várias listas em simultâneo. Por exemplo, se quisermos somar os elementos de duas listas dois a dois, podemos escrever:

```
julia> map((a, b) -> a + b, 1:5, 2:6)
[3, 5, 7, 9, 11]
```

Para além desta função, o Julia disponibiliza ainda uma variante sintática que pode ser útil para evitar escrever funções anónimas: as listas *em compreensão*, tal como descrevemos na secção 5.4.1. Usando esta forma, os cálculos anteriores podem também ser realizados através de:

```
julia> [sqr(i) for i in 10:20]
[100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]
julia> [x + y for (x, y) in zip(1:5, 2:6)]
[3, 5, 7, 9, 11]
```

8.8.2 Filtragem

Uma outra função muito útil é a que *filtra* uma lista. A filtragem é feita fornecendo um predicado que é aplicado a cada elemento da lista. Os elementos que satisfazem o predicado (i.e., em relação aos quais o predicado é verdade) são colecionados numa nova lista.

A definição é a seguinte:

```
filtra(predicado, lista) =
  if lista == []
    []
  elseif predicado(lista[1])
    [lista[1], filtra(predicado, lista[2:end])...]
  else
    filtra(predicado, lista[2:end])
  end
```

Usando esta função podemos, por exemplo, obter apenas os quadrados *divisíveis por* 3 dos números entre 10 e 20:

```
julia> filtra(n -> n%3 == 0, mapeia(sqr, enumera(10, 20, 1)))
[144, 225, 324]
```

A função `filtra` já existe pré-definida em Julia com o nome `filter`.

Tal como acontecia no caso do mapeamento, também no caso de uma filtragem, o Julia oferece uma sintaxe alternativa baseada no uso de listas em compreensão. Por exemplo, para produzir o quadrado dos números entre 1 e 20 que são divisíveis por três, podemos escrever:

```
julia> [sqr(n) for n in 10:20 if n%3 == 0]
[144, 225, 324]
```

8.8.3 Redução

Uma terceira função de grande utilidade é a que realiza uma *redução* numa lista. Esta função de ordem superior recebe uma operação, um elemento inicial, e uma lista e reduz a lista a um valor que resulta de intercalar a operação entre todos os elementos da lista. Por exemplo, para somar todos os elementos de uma lista l=`[`e_0`, `e_1`, ..., `e_n`]` podemos fazer `reduz(+, 0, `l`)` e obter $e_0+e_1+\ldots+e_n+0$. Assim, temos:

```
julia> reduz(+, 0, enumera(1, 100, 1))
5050
```

A definição da função é bastante simples:

```
reduz(f, v, lista) =
  if lista == []
    v
  else
    f(lista[1], reduz(f, v, lista[2:end]))
  end
```

Para vermos um exemplo mais interessante do uso desta função, consideremos o cálculo do factorial de um número. De acordo com a definição tradicional, não recursiva, temos:

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

Dito de outra forma, trata-se de multiplicar todos os números de uma enumeração desde 1 a n, i.e.:

```
factorial(n) =
  reduz((a, b) -> a*b, 1, enumera(1, n, 1))
```

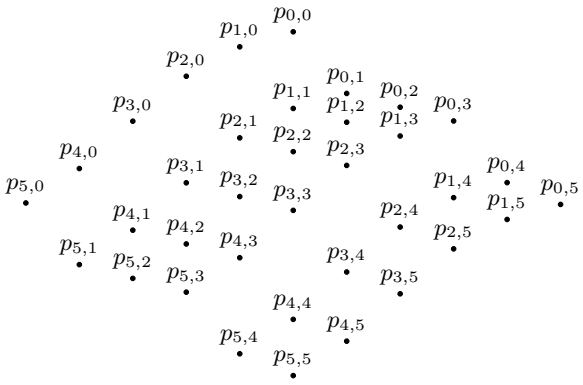
Como podemos confirmar nos exemplos anteriores, o valor inicial empregue é o elemento neutro da operação de combinação dos elementos da lista, mas não é forçoso que assim seja. Para vemos um exemplo onde isso não acontece, consideremos a determinação do maior valor existente numa lista de números. Neste caso, a função que empregamos para combinar sucessivamente os valores da lista é a função `max` que devolve o maior entre dois números. Se `l` for a lista de números, `[`e_0`, `e_1`, `e_2`, ...]`, o maior desses números pode-se obter através de `max(`e_0`, max(`e_1`, max(`e_2`, ...)))` que, obviamente, corresponde a uma redução da lista empregando a função `max`. Falta apenas determinar qual o elemento inicial a empregar. Uma hipótese será empregarmos a infinidade negativa $-\infty$ pois qualquer número será maior que ela. Outra, mais simples, será usarmos qualquer um dos elementos da lista, particularmente o primeiro por ser o de mais fácil acesso. Assim, podemos definir:

```
max_lista(lista) =
  reduz(max, lista[1], lista[2:end])
```

Tal como acontecia com as funções `map` e `filter`, também a função `reduz` existe pré-definida em Julia. Neste caso, o seu nome é `reduce` e recebe, por ordem, a função a combinar, a lista e, opcionalmente, o elemento neutro. Se este não for fornecido, o primeiro elemento da lista é usado como elemento neutro, o que permite simplificar a função anterior:

```
max_lista(lista) =
  reduce(max, lista)
```

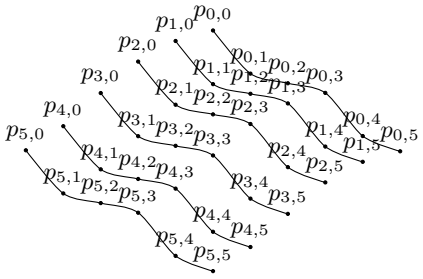
Exercício 8.8.1 Considere uma superfície representada por um conjunto de coordenadas tri-dimensionais, tal como se apresenta na seguinte imagem:

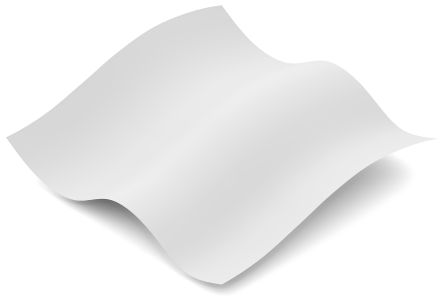


Admita que essas coordenadas estão armazenadas numa lista de listas da forma:

```
[[p0,0, p0,1, ..., p0,5],
 [p1,0, p1,1, ..., p1,5],
 ...,
 [p5,0, p5,1, ..., p5,5]]
```

Defina a função `superficie_interpolacao` que recebe uma lista com a forma anterior, começa por criar uma lista de *splines* em que cada *spline* S_i passa pelos pontos $p_{i,0}, p_{i,1}, \ldots, p_{i,5}$ tal como se apresenta na imagem seguinte à esquerda e acaba por usar essa lista de splines $S_0, S_1, \ldots, S_5$ para fazer uma interpolação suave de secções, tal como se apresenta na imagem seguinte à direita:





8.9 Geração de Modelos Tridimensionais

Até este momento, temos usado a ferramenta de CAD apenas como um visualizador das formas que os nossos programas geram. Vamos agora ver que uma ferramenta de CAD não é só um programa de desenho. É também uma base de dados sobre figuras geométricas. De facto, sempre que desenhamos algo a ferramenta de CAD regista na sua base de dados a entidade gráfica criada, bem como algumas informações adicionais relacionadas com essa entidade como, por exemplo, a sua cor.

Existem várias maneiras de se aceder às entidades criadas. Uma das mais simples para um utilizador "normal" da ferramenta de CAD será através do uso do rato, simplesmente "clicando" na entidade gráfica a que pretende aceder. Outra, mais útil para quem pretende programar, será através da invocação de funções do Julia que devolvem, como resultados, as entidades geométricas existentes. Há várias destas funções à nossa disposição mas, por agora, vamos limitar-nos a uma das mais simples: a função `all_shapes`.

A função `all_shapes` não recebe quaisquer argumentos e devolve uma lista com todas as entidade geométricas existentes na ferramenta de CAD. Naturalmente, se não existirem quaisquer entidades, a função devolve uma lista vazia.

A seguinte interacção demonstra o comportamento desta função:

```
julia> all_shapes()
[]
julia> circle(xy(1, 2), 3)
<circle 0>
julia> sphere(xyz(1, 2, 3), 4)
<sphere 1>
julia> all_shapes()
[<solid 2>, <circle 3>]
```

Na interacção anterior podemos reparar que a função `all_shapes` cria representações das entidades geométricas existentes na ferramenta de CAD sem ter qualquer ideia de como essas entidades lá foram parar, o que a impede de relacionar as formas existentes, como `<circle 3>`, com as formas que foram criadas a partir do Khepri, como `<circle 0>`. Para além disso, nem sempre a ferramenta de CAD disponibiliza informação sobre o tipo de forma geométrica em questão, o que explica o facto de a esfera `<sphere 1>` criada pelo Khepri ser posteriormente reconhecida apenas como o sólido `<solid 2>`.

Quando necessário (e possível), estas formas podem ser identificadas pelos reconhecedores `is_point`, `is_circle`, `is_line`, `is_polygon`, `is_spline`, `is_closed_spline`, `is_surface`, e `is_solid`.

Dada uma entidade geométrica, podemos estar interessados em conhecer as suas *propriedades*. O acesso a essas propriedades depende muito do que a ferramenta de CAD é capaz de disponibilizar. Por exemplo, para um círculo, podemos saber o seu centro através da função `circle_center` e o seu raio através de `circle_radius`, enquanto que para um ponto a função `point_position` devolve a sua posição e para uma linha poligonal a função `line_vertices` produz uma lista com as posições dos vértices dessa linha. Já para um sólido genérico, não temos acesso a qualquer propriedade.

Resumidamente, temos:

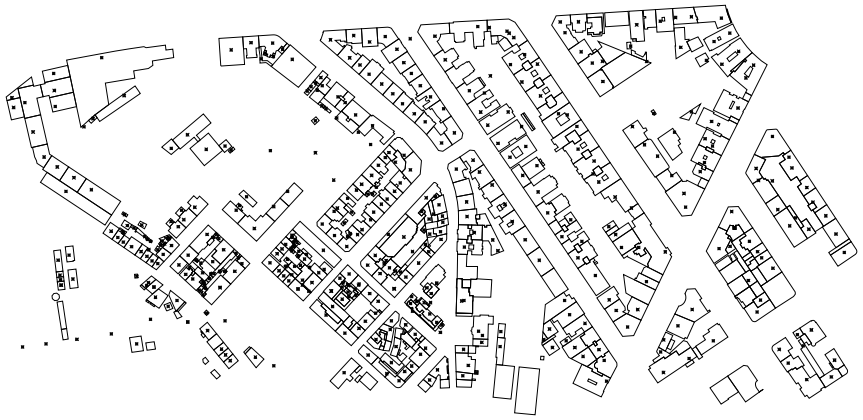


Figura 8.5: Planta fornecida pelos serviços camarários. Cada edifício é representado pelo seu polígono envolvente e por um ponto à cota do topo de edifício. Por erros nas plantas, alguns edifícios podem não possuir a cota respectiva enquanto que outros podem possuir mais do que uma cota. Pode também haver cotas que não estão associadas a qualquer edifício.

$$\texttt{point\_position(point(}p\texttt{))} = p$$

$$\texttt{circle\_center(circle(}p\texttt{, }r\texttt{))} = p$$

$$\texttt{circle\_radius(circle(}p\texttt{, }r\texttt{))} = r$$

$$\texttt{line\_vertices(line(}p_0\texttt{, }p_1\texttt{, ..., }p_n\texttt{))} = [p_0\texttt{, }p_1\texttt{, ..., }p_n]$$

Embora as equivalências anteriores mostrem a relação entre os construtores de entidades geométricas (`point`, `circle`, etc.) e os selectores dessas entidades (`point_position`, `circle_center`, etc.) é importante termos em conta que é possível usar os selectores com entidades que não foram construídas a partir do Khepri mas sim directamente na ferramenta de CAD. Isto pode ser particularmente útil quando pretendemos escrever programas que, ao invés de gerarem geometria, se limitam a processar geometria já existente.

Vamos agora exemplificar o uso destas operações na resolução de um problema real: a criação de modelos tridimensionais de edifícios a partir das plantas bidimensionais fornecidas pelos serviços camarários. Estas plantas caracterizam-se por representarem cada edifício através de um polígono que o delimita contendo, no seu interior, um ponto à cota do topo do edifício. A Figura 8.5 mostra um fragmento de uma destas plantas, em que se usou um ícone suficientemente grande para os pontos de modo a torná-los mais visíveis.

Infelizmente, não é invulgar encontrar plantas com erros e o exemplo apresentado na Figura 8.5 ilustra precisamente isso: uma observação atenta

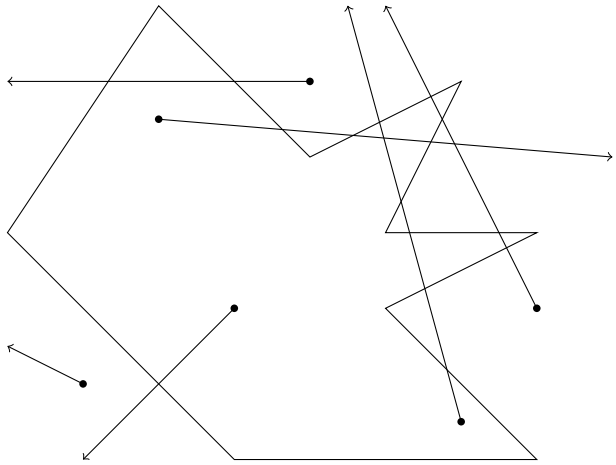


Figura 8.6: O uso de um raio para determinar se um ponto está contido num polígono.

irá encontrar edifícios que não possuem qualquer cota, assim como cotas que não estão associadas a qualquer edifício. Para além disso, é possível encontrar edifícios com mais do que uma cota. Estes são problemas que teremos de ter em conta quando pensarmos na criação de programas que manipulem estas plantas.

Para criarmos um modelo tridimensional a partir de uma destas plantas podemos, para cada polígono, formar uma região que vamos extrudir até à cota indicada pelo ponto correspondente. Para isso, temos de conseguir identificar, para cada polígono, qual é esse ponto (ou quais são os pontos, no caso de haver mais do que um).

Detectar se um ponto está contido no interior de um polígono é um problema clássico da Geometria Computacional para o qual existem várias soluções. Uma das mais simples consiste em traçar um raio a partir desse ponto e contar o número de intersecções que esse raio faz com as arestas do polígono, tal como se ilustra na Figura 8.6 para vários pontos. Se o número de intersecções for zero, então o ponto está obviamente fora do polígono. Se o número for um, então o ponto está necessariamente dentro do polígono. Se esse número for dois, então é porque o raio entrou e saiu do polígono e, portanto, o ponto está fora do polígono; se o número for três, então é porque o raio saiu, entrou e voltou a sair do polígono e, portanto, o ponto está no interior do polígono. Uma vez que cada entrada do raio no polígono implica a sua posterior saída, torna-se evidente que se o número de intersecções do raio com o polígono for par, então é porque o ponto estava fora do polígono e se o número de intersecções for ímpar então é porque o ponto estava no interior do polígono.

A implementação deste algoritmo é relativamente simples: dado um

ponto $P = (P_x, P_y)$ e uma lista V_s com os vértices do polígono $V_0, V_1, \ldots, V_n$, "criamos" um raio cuja origem é P e cujo destino é um ponto $Q = (Q_x, Q_y)$ qualquer suficientemente afastado do polígono. Para simplificar, vamos arbitrar que este raio é horizontal, o que permite que o ponto de destino Q tenha a mesma ordenada de P, i.e., $Q_y = P_y$. Para garantir que Q está suficientemente afastado do polígono, podemos calcular a maior abcissa dos vértices do polígono $\max(V_{0x}, V_{1x}, \ldots, V_{nx})$ e adicionamos-lhe uma pequena distância, por exemplo, uma unidade.

O cálculo da maior abcissa dos vértices torna-se simples quando se empregam funções de ordem superior: podemos começar por mapear a lista de vértices na lista das suas abcissas e, de seguida, operamos uma redução desta lista através da função `max`. Este raciocínio pode ser traduzido para Julia através da seguinte função:

```
ponto_no_poligono(p, vs) =
  let q = xy(reduce(max, map(cx, vs)) + 1, p.y)
    ...
  end
```

Em seguida, determinamos as intersecções entre o segmento definido pelos pontos $P - Q$ e os segmentos que constituem o polígono. Esses segmentos são definidos pelos vértices $V_0 - V_1$, $V_1 - V_2$, $\ldots$, $V_{n-1} - V_n$ e $V_n - V_0$, que podemos obter através de um mapeamento ao longo de duas listas, uma com os pontos $V_0, V_1, \ldots, V_n$ e outra com os pontos $V_1, \ldots, V_n, V_0$. A nossa função fica então com a forma:

```
ponto_no_poligono(p, vs) =
  let q = xy(reduce(max, map(cx, vs)) + 1, p.y)
    ... map((vi, vj) -> ...,
            vs,
            [vs[2:end]..., vs[1]])
```

A função que mapeamos ao longo das duas listas de vértices deverá produzir, para cada dois vértices consecutivos V_i e V_j, a intersecção do segmento $P - Q$ com o segmento $V_i - V_j$.

Para determinar a intersecção de dois segmentos de recta, podemos raciocinar da seguinte forma: dados os pontos P_0 e P_1 que delimitam uma recta, todos os pontos P_u entre P_0 e P_1 são determinados pela equação

$$P_u = P_0 + u(P_1 - P_0), 0 \leq u \leq 1$$

Se esse segmento de recta intersectar outro segmento de recta delimitado pelos pontos P_2 e P_3 e definido por

$$P_v = P_2 + v(P_3 - P_2), 0 \leq v \leq 1$$

então é forçoso que exista um u e v tais que

$$P_0 + u(P_1 - P_0) = P_2 + v(P_3 - P_2)$$

No caso bidimensional, temos $P_i = (x_i, y_i)$ e a equação anterior desdobra-se nas duas equações

$$x_0 + u(x_1 - x_0) = x_2 + v(x_3 - x_2)$$

$$y_0 + u(y_1 - y_0) = y_2 + v(y_3 - y_2)$$

Resolvendo o sistema, obtemos

$$u = \frac{(x_3 - x_2)(y_0 - y_2) - (y_3 - y_2)(x_0 - x_2)}{(y_3 - y_2)(x_1 - x_0) - (x_3 - x_2)(y_1 - y_0)}$$

$$v = \frac{(x_1 - x_0)(y_0 - y_2) - (y_1 - y_0)(x_0 - x_2)}{(y_3 - y_2)(x_1 - x_0) - (x_3 - x_2)(y_1 - y_0)}$$

É claro que para que as divisões possam ser realizadas é necessário que o denominador $(y_3 - y_2)(x_1 - x_0) - (x_3 - x_2)(y_1 - y_0)$ seja diferente de zero. Se tal não acontecer, é porque as rectas são paralelas. Se não forem paralelas, pode dar-se o caso de a intersecção ocorrer fora dos segmentos de recta em questão, pelo que temos de verificar se os valores de u e v obtidos obedecem às condições $0 \leq u \leq 1$ e $0 \leq v \leq 1$. Quando tal acontece, o ponto exacto da intersecção pode ser calculado substituindo u na equação $P_u = P_0 + u(P_1 - P_0)$.

Isto leva-nos à seguinte definição para uma função que calcula a intersecção:

```
interseccao_segmentos(p0, p1, p2, p3) =
  let denom = (p3.y - p2.y)*(p1.x - p0.x) - (p3.x - p2.x)*(p1.y - p0.y)
    if denom == 0
      nothing
    else
      let u = ((p3.x - p2.x)*(p0.y - p2.y) - (p3.y - p2.y)*(p0.x - p2.x))/denom,
          v = ((p1.x - p0.x)*(p0.y - p2.y) - (p1.y - p0.y)*(p0.x - p2.x))/denom
        if 0 <= u <= 1 && 0 <= v <= 1
          xy(p0.x + u*(p1.x - p0.x), p0.y + u*(p1.y - p0.y))
        else
          nothing
        end
      end
    end
  end
```

Usando esta função, podemos determinar todas as intersecções com as arestas de um polígono fazendo

```
ponto_no_poligono(p, vs) =
  let q = xy(reduce(max, map(cx, vs)) + 1, p.y)
    ... map((vi, vj) -> interseccao_segmentos(p, q, vi, vj),
            vs,
            [vs[2:end]..., vs[1]])
```

O resultado do mapeamento será, para cada par de vértices consecutivos $V_i - V_j$, um ponto de intersecção com a recta $P - Q$ ou `nothing` no

caso de não existir intersecção. Uma vez que apenas pretendemos conhecer as intersecções, podemos agora filtrar esta lista, ficando apenas com os que são pontos, i.e., os que não forem falsos:

```
ponto_no_poligono(p, vs) =
  let q = xy(reduce(max, map(cx, vs)) + 1, p.y)
      rs = filter(e -> e != nothing,
                  map((vi, vj) -> interseccao_segmentos(p, q, vi, vj),
                      vs,
                      [vs[2:end]..., vs[1]]))
    ...
  end
```

Agora, para sabermos quantas intersecções ocorrem, basta-nos medir o comprimento da lista resultante e verificar se o resultado é impar:

```
ponto_no_poligono(p, vs) =
  let q = xy(reduce(max, map(cx, vs)) + 1, p.y)
      rs = filter(e -> e != nothing,
                  map((vi, vj) -> interseccao_segmentos(p, q, vi, vj),
                      vs,
                      [vs[2:end]..., vs[1]]))
    length(rs)%2 == 1
  end
```

Exercício 8.9.1 A função `ponto_no_poligono` não é tão eficiente como poderia ser pois realiza um mapeamento seguido de uma filtragem apenas para contar quantos elementos resultam na lista final. Na prática, esta combinação de operações não é mais do que uma contagem do número de vezes que um predicado binário é satisfeito ao longos dos sucessivos elementos de duas listas. Defina a operação `quantos` que implementa este processo. A título de exemplo, considere:

```
julia> quantos((a, b) -> a > b, [1, 5, 3, 4, 6, 2], [1, 6, 2, 5, 4, 3])
2
```

Exercício 8.9.2 Reimplemente a função `ponto_no_poligono` de modo a utilizar a função `quantos`.

A partir da função `ponto_no_poligono` já nos é possível estabelecer a associação entre cada ponto e cada polígono tal como estes ocorrem nas plantas fornecidas pelos serviços camarários. Contudo, tal como referimos inicialmente, temos de ter cuidado com o facto de ser também possível, para um dado polígono, não existir qualquer ponto associado ou existir mais do que um ponto associado. Uma maneira de tratarmos de forma idêntica todas estas situações será computar, para cada polígono, a lista de pontos que ele contém. Idealmente, essa lista deverá ter apenas um elemento, mas, no caso de haver erros numa planta, poderá não ter nenhum ou ter mais do que um. Em qualquer caso, a função retorna sempre uma lista, pelo que nunca dará erro.

Para produzir essa lista de pontos para um dado polígono teremos de testar cada um dos pontos da planta, para ver se ele pertence ao polígono. Ora isto não é mais do que uma filtragem da lista de pontos, ficando apenas com aqueles que estão contidos no polígono. Assim, admitindo que `pts` é a lista de pontos da planta e `vs` são os vértices de um polígono em particular, podemos definir:

```
pontos_no_poligono(pts, vs) =
  filter(pt -> ponto_no_poligono(xyz(pt.x, pt.y, vs[1].z), vs),
         pts)
```

Finalmente, dada uma lista de pontos representando as coordenadas do topo dos edifícios e uma listas de polígonos (cada um implementado pela lista dos seus vértices) representando o perímetro da base dos edifícios, vamos criar uma representação tridimensional desses edifícios determinando, para cada poligono, quais os pontos nele contidos e, de seguida, actuamos em conformidade com o número de pontos encontrados:

- Se esse número é zero, não sabemos qual a cota do edifício. Isso representa um erro na planta que podemos não querer tratar ou que podemos tratar simplesmente arbitrando uma altura. Por agora, vamos escolher nada fazer.

- Se esse número é um, então esse ponto está localizado no topo do edifício e a sua coordenada z dá-nos a altura correspondente.

- Se esse número é maior do que um, então há múltiplos pontos aplicáveis, o que poderá corresponder a várias situações possíveis:

  – Se um dos pontos em questão pertencer também a outro polígono, então é porque há polígonos contidos noutros polígonos ou polígonos que se intersectam. O primeiro caso pode corresponder a um edifício cuja forma varia em altura (por exemplo, contendo uma casa de máquinas no topo). O segundo caso, tipicamente, é um erro na planta.

  – Se os pontos pertencem todos exclusivamente ao mesmo polígono, então isso poderá representar edifícios com topos que não são planos horizontais.

O tratamento correcto destes últimos casos é relativamente complexo. Como apenas pretendemos criar uma aproximação prismática à forma do edifício, vamos empregar uma abordagem muito simples: empregamos como altura do edifício a maior das cotas encontradas. A seguinte função implementa este comportamento:

```
cria_edificios(pontos, poligonos) =
  for poligono in poligonos
    let pts = pontos_no_poligono(pontos, poligono),
        n_pts = length(pts)
      if n_pts == 0
        nothing
      elseif n_pts == 1
        cria_edificio(poligono, cz(pts[1]))
      else
        cria_edificio(poligono, reduce(max, map(cz, pts)))
      end
    end
  end
```

Exercício 8.9.3 Uma outra abordagem possível para calcular a altura de um edifício cujo polígono correspondente contém vários pontos consiste em usar a *média* das coordenadas z desses pontos. Implemente esta abordagem.

Para criarmos um edifício a partir da lista de vértices do seu perímetro e da sua altura vamos simplesmente criar uma região poligonal na base do edifício que extrudimos até ao seu topo:

```
cria_edificio(vertices, cota) =
  if cota > 0
    extrusion(surface_polygon(vertices), vz(cota))
  end
```

Até agora temos resolvido o problema apenas do ponto de vista geométrico, representando os pontos da planta pelas suas coordenadas e os polígonos pelas coordenadas dos seus vértices. No entanto, como sabemos, aquilo que a ferramenta de CAD disponibiliza são entidades geométricas e são estas que contêm a informação de que necessitamos. Ora, dada uma lista de entidades, seleccionar as que correspondem aos pontos não é mais do que uma filtragem que apenas fica com as entidades que satisfazem o predicado `is_point`. Dadas estas entidades, obter as suas coordenadas não é mais do que um mapeamento com a função `point_position`. Isto quer dizer que podemos definir a função que obtém as coordenadas de todos os pontos existentes numa lista de entidades:

```
pontos(entidades) =
  map(point_position, filter(is_point, entidades))
```

Do mesmo modo, dada uma lista de entidades, podemos seleccionar a lista dos vértices dos polígonos através de uma filtragem seguida de um mapeamento, tal como se segue:

```
poligonos(entidades) =
  map(line_vertices, filter(is_polygon, entidades))
```

Finalmente, estamos em condições de definir uma função que, a partir

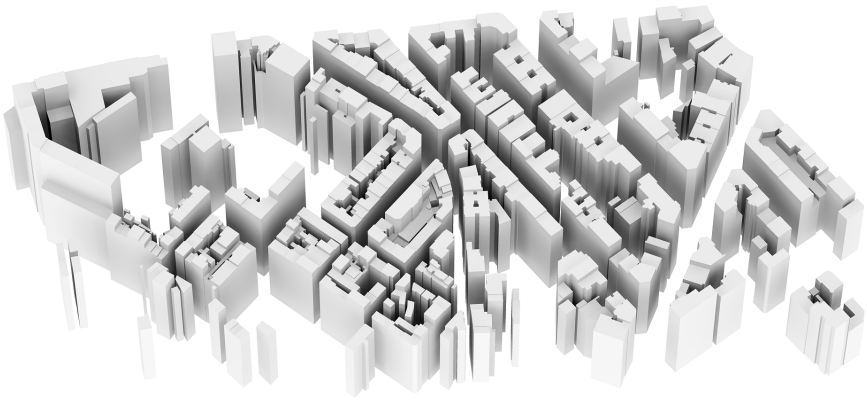


Figura 8.7: Modelação tridimensional dos edifícios presentes na planta apresentada na Figura 8.5.

do conjunto de entidades de uma planta, cria as representações tridimensionais correspondentes:

```
cidade(entidades) =
  cria_edificios(pontos(entidades), poligonos(entidades))
```

Naturalmente, o conjunto de entidades a processar pode ser produzido de forma selectiva ou, alternativamente, podem ser todas as entidades existentes numa dada planta. Neste último caso, basta-nos fazer:

```
cidade(all_shapes())
```

A título de exemplo, apresentamos na Figura 8.7 o resultado da avaliação da expressão anterior para a planta que apresentámos na Figura 8.5.

Capítulo 9

Representação Paramétrica

9.1 Introdução

Até agora, apenas temos produzido curvas descritas por funções na forma $y = f(x)$. Estas funções dizem-se na forma *Cartesiana*.

Uma outra forma de representar matematicamente uma curva é através de equações $F(x, y) = 0$. Esta forma, dita *implícita*, é mais geral que a anterior que, na verdade, não é mais que a resolução desta em relação à ordenada y. Como exemplo de uma curva descrita na forma implícita consideremos a equação $x^2 + y^2 - r^2 = 0$ que descreve uma circunferência de raio r com centro em $(0, 0)$. Infelizmente, esta segunda forma de representar curvas não é tão útil como a anterior pois nem sempre é trivial (ou possível) resolver a equação em relação à ordenada.[1] Por exemplo, no caso da circunferência, o melhor que conseguimos fazer é produzir *duas* funções:

$$y(x) = \pm\sqrt{r^2 - x^2}$$

Existe, no entanto, uma terceira forma de representação de curvas que se torna particularmente útil: a forma *paramétrica*. A forma paramétrica baseia-se na ideia de que a curva pode ser percorrida por um ponto cuja posição evolui ao longo do tempo. O tempo, aqui, é meramente um *parâmetro* que determina a posição do ponto sobre a curva. Retomando o caso da circunferência com centro em $(0, 0)$, a sua descrição paramétrica será

$$x(t) = r\cos t$$

$$y(t) = r\sin t$$

Como é óbvio, para que o ponto de coordenadas (x, y) possa percorrer toda a circunferência, basta que o parâmetro t varie no intervalo $[0, 2\pi[$.

[1] Excepto no caso de rectas, cuja equação geral é $ax + by + c = 0, b \neq 0$ e onde é óbvio que a resolução em relação à ordenada é $y = -\frac{ax+c}{b}$.

As equações anteriores denominam-se as *equações paramétricas* da curva. Se, numa representação paramétrica, eliminarmos o parâmetro, naturalmente encontraremos a equação original da curva. Por exemplo, no caso da circunferência, se somarmos os quadrados das equações paramétricas obtemos

$$x^2 + y^2 = r^2(\cos^2 t + \sin^2 t) = r^2$$

Um aspecto interessante da forma paramétrica é que ela permite usar qualquer sistema de coordenadas. As equações anteriores, por exemplo, podem ficar ainda mais simples se usarmos o sistema de coordenadas polares:

$$\begin{cases} \rho(t) = r \\ \phi(t) = t \end{cases}$$

Embora tenhamos explicado a representação paramétrica em termos de coordenadas bidimensionais, a extensão ao espaço tridimensional é trivial. Basta, para isso, considerar cada ponto tridimensional função de um parâmetro, i.e., $(x, y, z)(t) = (x(t), y(t), z(t))$.

9.2 Computação de Funções Paramétricas

Uma vez que a representação paramétrica simplifica substancialmente a construção de curvas, pretendemos agora definir funções que, a partir da descrição paramétrica de uma curva, produzem uma lista de pontos pertencentes a essa curva. Para isso, teremos de fazer variar o parâmetro t ao longo do seu domínio e, para cada valor, colecionamos numa lista cada um dos pontos computados pelas equações paramétricas. Por exemplo, no caso de querermos gerar n pontos uniformemente distribuídos ao longo de uma circunferência de raio r, teremos de fazer t variar no intervalo $[0, 2\pi[$, mas separando os valores de t por $\Delta_t = \frac{2\pi}{n}$. Note-se que o intervalo é aberto em 2π pois o ponto da circunferência correspondente a $t = 0$ coincide com o ponto correspondente a $t = 2\pi$.

Para gerarmos estes valores de t uniformemente espaçados de Δ_t podemos usar a função `enumera`, discutida na secção 5.3.1. De seguida, podemos mapear a equação paramétrica sobre essa lista de valores de t:

```
map(t -> pol(r, t),
    enumera(0, 2*pi, dt))
```

Como habitualmente, podemos encapsular aquela expressão numa função parametrizada e podemos generalizá-la para permitir especificar o centro p da circunferência:

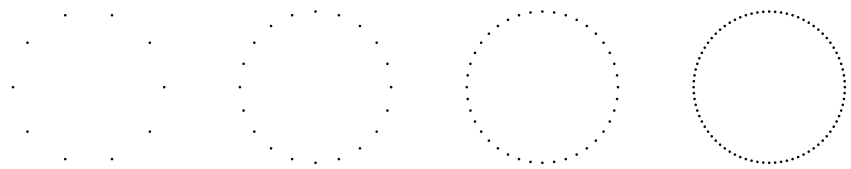


Figura 9.1: Pontos de uma circunferência. Da esquerda para a direita, temos $\Delta_t = \frac{2\pi}{10}$, $\Delta_t = \frac{2\pi}{20}$, $\Delta_t = \frac{2\pi}{30}$, e $\Delta_t = \frac{2\pi}{40}$

```
pontos_circulo(p, r, dt) =
  map(t -> p + vpol(r, t), enumera(0, 2*pi, dt))
```

A Figura 9.1 mostra as posições geradas pela função `pontos_circulo` para diferentes valores de Δ_t.

Para vermos um outro exemplo prático, consideremos a *espiral de Arquimedes*: a curva descrita por um ponto que se move com velocidade constante v ao longo de um raio que gira em torno de um pólo com velocidade angular constante ω. Em coordenadas polares, a equação da espiral de Arquimedes tem a forma

$$\rho = \frac{v}{\omega}\phi$$

ou, com $\alpha = \frac{v}{\omega}$,

$$\rho = \alpha\phi$$

Quando convertemos a equação anterior para coordenadas rectangulares, usando as fórmulas

$$\begin{cases} x = \rho\cos\phi \\ y = \rho\sin\phi \end{cases}$$

obtemos

$$\begin{cases} x(\phi) = \alpha\phi\cos\phi \\ y(\phi) = \alpha\phi\sin\phi \end{cases}$$

que, como se pode ver, é uma descrição paramétrica em função de ϕ.

Mais simples ainda é a conversão directa para a forma paramétrica da representação polar da espiral de Arquimedes. Basta-nos substituir ϕ por t e acrescentar mais uma equação que expressa esta substituição, ou seja

$$\begin{cases} \rho(t) = \alpha t \\ \phi(t) = t \end{cases}$$

Tal como fizemos para a circunferência, a computação das coordenadas da espiral de arquimedes pode ser feita mapeando a função que nos dá as posições sobre uma lista de valores de t. No entanto, agora t não está limitado ao intervalo $[0, 2\pi[$, podendo variar livremente até onde pretendermos, num intervalo $[t_0, t_1[$. Assim, podemos escrever:

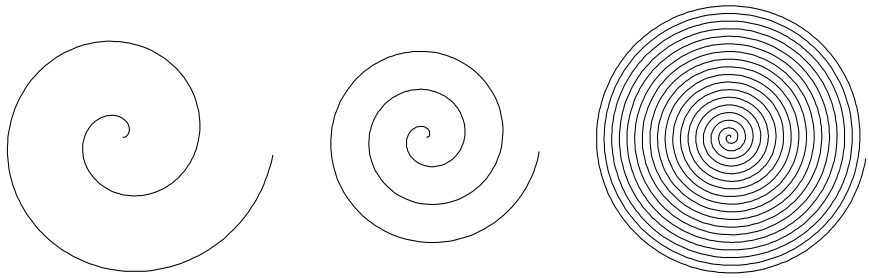


Figura 9.2: Espirais de Arquimedes. Da esquerda para a direita, os parâmetros são $\alpha = 2, t \in [0, 4\pi[$, $\alpha = 1, t \in [0, 6\pi[$, $\alpha = 0.2, t \in [0, 36\pi[$.

```
espiral_arquimedes(p, alfa, t0, t1, dt) =
  map(t -> p + vpol(alfa*t, t), enumera(t0, t1, dt))
```

A Figura 9.2 mostra três espirais de Arquimedes desenhadas a partir das seguintes invocações:

```
spline(espiral_arquimedes(xy(0, 0), 2.0, 0, 4*pi, 0.1))
spline(espiral_arquimedes(xy(50, 0), 1.0, 0, 6*pi, 0.1))
spline(espiral_arquimedes(xy(100, 0), 0.2, 0, 36*pi, 0.1))
```

9.3 Erros de Arredondamento

Embora muito útil, a função `enumera` tem um problema: quando o incremento `dt` não é um número inteiro, a sua adição sucessiva vai acumulando erros de arredondamento, o que pode provocar um comportamento bizarro. Para exemplificarmos a situação, consideremos o número de elementos de duas enumerações do intervalo $[0, 1[$, a primeira com um incremento de $\frac{1}{10}$ e a segunda com um incremento de $\frac{1}{100}$:

```
julia> length(enumera(0, 1, 1/10))
11
julia> length(enumera(0, 1, 1/100))
100
```

Obviamente, há ali um problema: aparentemente, a primeira enumeração não terminou quando devia, produzindo um elemento a mais. O problema resulta do facto de os incrementos usados não serem representáveis com exactidão em notação binária. Na verdade, 0.1 é representado por um número que é ligeiramente inferior a $\frac{1}{10}$, enquanto que 0.01 é representado por um número ligeiramente superior à fracção $\frac{1}{100}$. A consequência é que uma soma suficientemente grande destes números acaba por acumular

um erro que faz com que o último valor calculado seja, afinal, superior ao limite, pelo que é descartado.[2]

Para evitar estes problemas seria natural pensarmos que nos devemos limitar a usar incrementos sob a forma de fracções mas, infelizmente, isso nem sempre é possível. Por exemplo, quando usamos funções trigonométricas (como senos e cossenos), é usual termos de gerar valores num intervalo que corresponde a um múltiplo do período das funções que, como sabemos, envolve o número irracional π, não representável como fracção. Assim, para lidarmos correctamente com números reais, devemos usar outra abordagem que se baseie em produzir o número de valores realmente pretendido, evitando adições sucessivas. Para isso, ao invés de usarmos o incremento como parâmetro, iremos antes usar o número de valores pretendidos.

Na sua forma actual, a partir dos limites t_0 e t_1 e do incremento Δ_t, a função `enumera` gera cada um dos pontos t_i calculando:

$$t_i = t_0 + \underbrace{\Delta_t + \Delta_t + \cdots + \Delta_t}_{i \text{ vezes}} \equiv$$

$$t_i = t_0 + \sum_{j=0}^{i} \Delta_t$$

O problema da fórmula anterior é, como vimos, a potencialmente grande acumulação de erros que ocorre ao somarmos um elevado número de vezes o ligeiro erro existente no valor de Δ_t. Para minimizarmos essa acumulação de erros temos de encontrar uma formula alternativa que evite o termo Δ_t. Uma possibilidade atraente é considerar, não um incremento Δ_t, mas sim um número n de incrementos Δ_t que interessa computar. Obviamente, esse número n relaciona-se com Δ_t pela equação

$$n = \frac{t_1 - t_0}{\Delta_t}$$

Consequentemente, temos também

$$\Delta_t = \frac{t_1 - t_0}{n}$$

Substituindo na equação anterior, obtemos

$$t_i = t_0 + \sum_{j=0}^{i} \Delta_t \equiv$$

[2] Este fenómeno tem sido causa de inúmeros problemas no mundo da informática. Desde erros nos sistemas de defesa anti-míssil ao cálculo errado de índices bolsistas, a história da informática está repleta de exemplos catastróficos causados por erros de arredondamento.

$$t_i = t_0 + \sum_{j=0}^{i} \frac{t_1 - t_0}{n} \equiv$$

$$t_i = t_0 + \frac{t_1 - t_0}{n} \sum_{j=0}^{i} 1 \equiv$$

$$t_i = t_0 + \frac{t_1 - t_0}{n} i$$

O aspecto fundamental da última equação é que ela já não corresponde a uma soma de i termos onde pode ocorrer uma acumulação de erros de arredondamento, mas sim a uma "função" $f(i) = \frac{t_1 - t_0}{n} i$ de i, em que i não acarreta qualquer erro pois é simplesmente um número inteiro que evolui desde 0 até n (exclusive). A partir deste número é agora possível calcular o valor de t_i que, embora possa ter os inevitáveis erros de arredondamento causados pela não utilização de números fraccionários, não terá uma acumulação desses erros.

Com base nesta última fórmula é agora fácil escrever uma nova função `enumera_n` que computa directamente o valor de t_i a partir do número n de valores pretendidos no intervalo $[t_0, t_1[$:

```
enumera_n(t0, t1, n) =
  map(i -> t0 + i*(t1 - t0)/n, enumera(0, n, 1))
```

Como é natural, esta nova definição não elimina erros de arredondamento, mas evita a sua acumulação.

Na verdade, existe uma variante da função `enumera_n` pré-definida em Khepri, com o nome `division`. À semelhança da função `enumera_n`, a função `division` tem como parâmetros os limites do intervalo e o número de incrementos:

```
julia> division(0, 1, 4)
[0.0, 0.25, 0.5, 0.75, 1.0]
julia> division(0, pi, 4)
[0.0,
 0.7853981633974483,
 1.5707963267948966,
 2.356194490192345,
 3.141592653589793]
```

Diferentemente da função `enumera_n`, a função `division` tem ainda um parâmetro opcional (por omissão, o valor `true`) destinado a indicar se queremos incluir o último elemento do intervalo. Esta opção destina-se a facilitar a utilização de funções periódicas. Para melhor percebermos este parâmetro, consideremos que queremos colocar quatro objectos nos quatro pontos cardeais. Para isso, podemos usar coordenadas polares, dividindo o círculo em quatro partes, com os ângulos 0, $\frac{\pi}{2}$, π, e $\frac{3\pi}{2}$. No entanto, se

usarmos a expressão `division(0, 2*pi, 4)` iremos não só obter aqueles valores mas ainda o limite superior do intervalo 2π, o que iria implicar colocar dois objectos sobrepostos, um para o ângulo 0 e outro para 2π. Naturalmente, podemos resolver o problema usando `division(0, 3*pi/2, 3)` mas isso parece bastante menos natural do que escrever `division(0, 2*pi, 4, false)`, ou seja, que queremos dividir 2π em quatro bocados mas não queremos o último valor pois será igual ao primeiro (a menos de um período).

9.4 Mapeamentos e enumerações

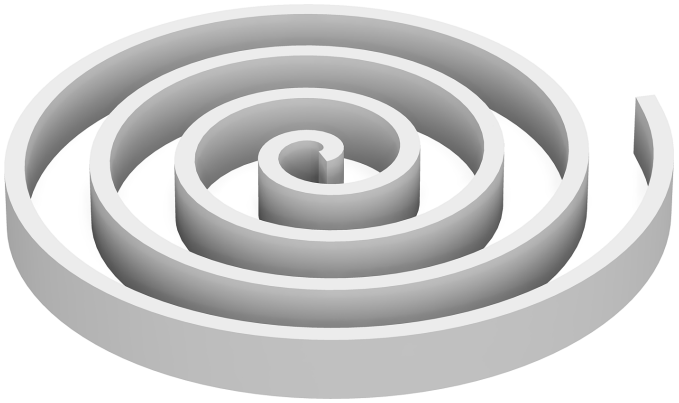
Como vimos na função `espiral_arquimedes`, a geração de curvas paramétricas implica mapear uma função sobre uma divisão de um intervalo em partes iguais. Uma vez que esta combinação é tão frequente, o Khepri disponibiliza uma função denominada `map_division` que a realiza de forma mais eficiente. Formalmente, temos:

$$\texttt{map\_division}(f, t_0, t_1, n) \equiv \texttt{map}(f, \texttt{division}(t_0, t_1, n))$$

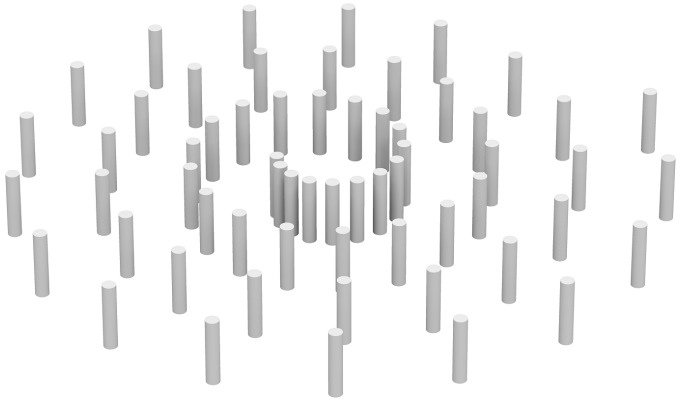
As funções `map`, `division`, e `map_division` permitem-nos gerar curvas com grande simplicidade. São, consequentemente, um excelente ponto de partida para a experimentação. Nas secções seguintes vamos visualizar algumas das curvas que, por uma razão ou outra, ficaram para a história da Matemática.

Exercício 9.4.1 Redefina a função `espiral_arquimedes` que calcula uma lista de pontos por onde passa uma espiral de Arquimedes, mas empregando a função `division`. Ao invés do incremento Δ_t, a sua função deverá antes receber o número de pontos n.

Exercício 9.4.2 Defina a função `parede_espiral_arquimedes` que, a partir de uma espessura e altura e ainda dos parâmetros de uma espiral de Arquimedes, cria uma parede em espiral, tal como se apresenta na seguinte imagem:



Exercício 9.4.3 Defina a função `cilindros_espiral_arquimedes` que constrói n cilindros de raio r e altura h e os coloca ao longo de uma espiral de arquimedes, de acordo com os parâmetros desta espiral, tal como se apresenta na seguinte figura:



9.4.1 Espiral de Fermat

A espiral de Fermat, também conhecida pelo nome de espiral parabólica, é uma curva semelhante a uma espiral de Arquimedes mas em que a equação que a define é

$$\rho^2 = \alpha^2 \phi$$

Resolvendo a equação em ordem a ρ, obtemos

$$\rho = \pm\alpha\sqrt{\phi}$$

Dividindo a curva em duas metades para lidar com os dois sinais, temos:

```
meia_espiral_fermat(p, a, t, n) =
  [p + vpol(a*sqrt(t), t) for t in division(0, t, n)]


espiral_fermat(p, a, t, n) =
  [reverse(meia_espiral_fermat(p, +a, t, n))...,
   meia_espiral_fermat(p, -a, t, n)[2:end]...]
```

Para vermos um exemplo da espiral de Fermat, podemos avaliar a seguinte expressão:

```
spline(espiral_fermat(xy(0, 0), 1.0, 16*pi, 400))
```

O resultado da avaliação anterior está representado na Figura 9.3.

Um aspecto interessante da espiral de Fermat é o facto de modelar alguns fenómenos naturais, em particular, o arranjo das sementes numa flor

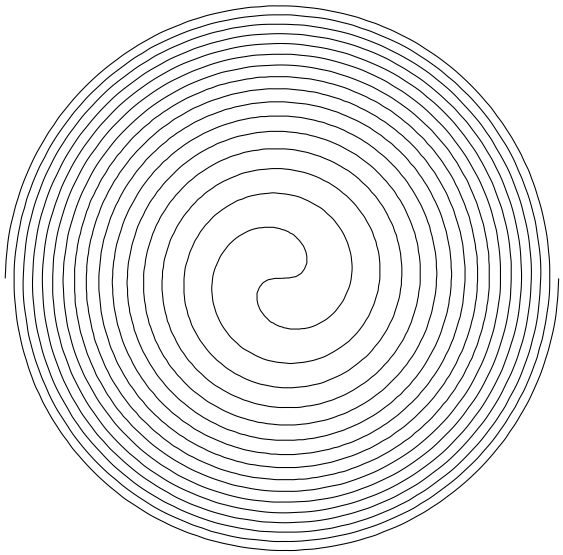


Figura 9.3: A espiral de Fermat para $\phi \in [0, 16\pi]$.

de girassol. Num girassol, as sementes são dispostas numa superfície circular e, de modo a conseguir fazer crescer o maior número de sementes que for possível, elas encontram-se encostadas umas às outras, da forma mais compacta que for possível.

Nestas flores, cada semente está posicionada de acordo com as equações[3]

$$\begin{cases} \rho = \alpha\sqrt{n} \\ \phi = \delta \times n \end{cases}$$

em que n é o índice da semente contado a partir do centro da flor[4], α é o factor de escala e δ é o *ângulo dourado* (também chamado ângulo de Fibonacci) e definido por $\delta = \frac{\pi}{\Phi^2}$ onde $\Phi = \frac{\sqrt{5}+1}{2}$ é a razão dourada. Da definição resulta que $\delta = 2.39996 \approx 2.4$.

Como podemos ver, esta equação é idêntica à da espiral de Fermat mas com a diferença de o ângulo ϕ não crescer em função de n, mas sim em função de $\delta \times n$, afastando cada duas sementes de um ângulo δ.

Para visualizarmos esta curva, é preferível desenhar uma "semente" em cada coordenada. Assim, podemos definir a função `girassol` que, dado um ponto central `p`, um factor de crescimento `a`, um ângulo `d`, um raio `r` e um número de sementes `n`, desenha as sementes do Girassol com um círculo de raio `r` para cada semente.

[3]Este modelo foi proposto por H. Vogel em 1979.

[4]Este índice é inversamente proporcional à ordem de crescimento da semente.

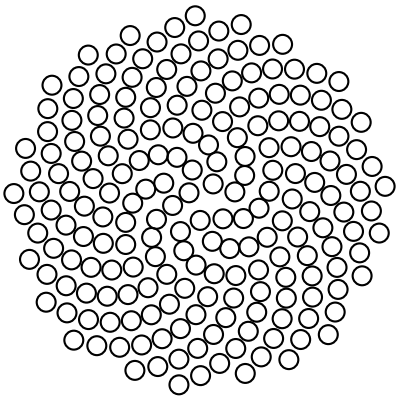


Figura 9.4: A disposição das sementes de um Girassol.

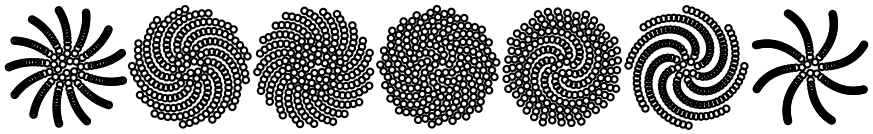


Figura 9.5: A disposição das sementes de Girassóis imaginários em que, da esquerda para a direita, o ângulo $delta$ corresponde a um incremento, relativamente ao ângulo dourado $\frac{\pi}{\Phi^2}$ de $+0.03$, $+0.02$, $+0.01$, $+0.00$, -0.01, -0.02 e -0.03.

```
girassol(p, a, d, r, n) =
  for t in division(1, n, n - 1)
    circle(p + vpol(a*sqrt(t), d*t), r)
  end
```

As seguintes avaliações permitem-nos desenhar uma aproximação à disposição das sementes do Girassol:

```
angulo_dourado = 2*pi/((sqrt(5) + 1)/2)^2

girassol(xy(0, 0), 1.0, angulo_dourado, 0.75, 200)
```

O resultado está visível na Figura 9.4.

É interessante verificar que a disposição das sementes é extremamente sensível ao ângulo δ. Na Figura 9.5 estão desenhados girassóis em tudo idênticos ao da Figura 9.4 excepto no ângulo δ que difere do ângulo dourado $\frac{\pi}{\Phi^2}$ em apenas umas centésimas de radiano.

9.4.2 Cissóide de Diocles

Diz a lenda que, no século V antes de Cristo, a população Grega foi atingida pela peste. Na esperança de uma resposta, os Gregos recorreram ao

Oráculo de Delos que profetizou que o problema estava na incorrecta veneração que estava a ser prestada ao Deus Apolo. De acordo com o Oráculo, Apolo só ficaria apaziguado se se duplicasse o volume do altar em forma de cubo que os Gregos lhe tinham dedicado.

Os Gregos, na tentativa de rapidamente se livrarem dos terríficos efeitos da peste, encetaram de imediato a tarefa de construção de um novo altar cúbico, mas, infelizmente, ao invés de duplicarem o seu *volume*, duplicaram a sua *aresta*, implicando que o volume final era oito vezes maior que o volume original e não apenas duas vezes maior como o oráculo tinha ordenado. Aparentemente, o oráculo sabia o que dizia, pois Apolo não se satisfez com o novo altar e a peste continuou a devastar a Grécia.

Após terem percebido o erro, os Gregos tentaram então descobrir qual a alteração correcta a fazer ao comprimento da aresta do cubo para duplicar o seu volume mas nunca conseguiram encontrar uma solução. A duplicação do volume de um cubo passou então a designar-se o problema de Delos e, durante séculos, atormentou os geómetras.[5] Foi apenas dois mil anos mais tarde, pela mão de Descartes, que se demonstrou que as técnicas Gregas—que limitavam a construção geométrica ao uso de régua e compasso—nunca poderiam resolver o problema.

No entanto, muito antes de Descartes, no século II antes de Cristo, o matemático grego Diócles tinha encontrado uma "solução," mas à custa da utilização de uma curva especial, actualmente denominada *cissóide* de Diócles. Essa curva extraordinária, infelizmente, não é desenhável apenas empregando régua e compasso, o que implica que a solução encontrada seria, quando muito, uma solução aproximada e não a verdadeira solução do problema.

A cissóide de Diócles define-se pela equação

$$y^2(2a - x) = x^3, x \in [0, a[$$

Colocando y em termos de x, obtemos:

$$y = \pm\sqrt{\frac{x^3}{2a - x}}$$

Infelizmente, esta formulação da curva é pouco apropriada pois obriga-nos a tratar separadamente os sinais $\pm$. Uma conversão para coordenadas polares permite-nos obter

$$\rho^2 \sin^2 \phi(2a - \rho\cos\phi) = \rho^3 \cos^3 \phi$$

Simplificando e usando a identidade Pitagórica $\sin^2 \phi + \cos^2 \phi = 1$, obtemos

$$(1 - \cos^2 \phi)(2a - \rho\cos\phi) = \rho\cos^3 \phi$$

[5] Felizmente, a peste não durou tanto como as dificuldades dos geómetras.

ou seja,

$$2a(1-\cos^2\phi)-\rho\cos\phi+\rho\cos^3\phi=\rho\cos^3\phi$$

Simplificando, obtemos finalmente

$$\rho=2a(\frac{1}{\cos\phi}-\cos\phi)$$

Uma vez que $\cos\pm\frac{\pi}{2}=0$, a curva tende para infinito para aqueles valores. Isto delimita o intervalo de variação a $\phi\in]-\frac{\pi}{2},+\frac{\pi}{2}[$.

Obviamente, para transformarmos a representação polar numa representação paramétrica fazemos simplesmente

$$\begin{cases}\rho(t)=2a(\frac{1}{\cos t}-\cos t)\\ \phi(t)=t\end{cases}$$

Finalmente, para permitir "centrar" a curva num ponto arbitrário, vamos proceder a uma translação a partir desse ponto. Assim, para definirmos esta curva em Julia basta-nos então fazer:

```
cissoide_diocles(p, a, t0, t1, n) =
  map_division(t -> p + vpol(2*a*(1.0/cos(t) - cos(t)), t), t0, t1, n)
```

A Figura 9.6 mostra uma sequência de Cissóides de Diócles desenhadas pela expressões:

```
spline(cissoide_diocles(xy(0, 0), 10.0, -0.65, 0.65, 100))
spline(cissoide_diocles(xy(5, 0), 5.0, -0.8, 0.8, 100))
spline(cissoide_diocles(xy(10, 0), 2.5, -1.0, 1.0, 100))
spline(cissoide_diocles(xy(15, 0), 1.0, -1.25, 1.25, 100))
spline(cissoide_diocles(xy(20, 0), 0.5, -1.4, 1.4, 100))
```

9.4.3 Lemniscata de Bernoulli

Em 1694, Bernoulli publicou uma curva a que deu o nome de *lemniscata*. Essa curva acabou por se tornar imensamente famosa por ter sido adoptada como símbolo para a representação do infinito: ∞. Actualmente, a curva é conhecida por lemniscata de Bernoulli, para a distinguir de outras curvas que possuem uma forma semelhante.

A lemniscata de Bernoulli é descrita pela equação

$$(x^2+y^2)^2=a^2(x^2-y^2)$$

Infelizmente, como já discutimos anteriormente, esta forma de representação analítica da curva é pouco apropriada para o seu desenho, pois é difícil colocar uma das variáveis em termos da outra. No entanto, se lhe

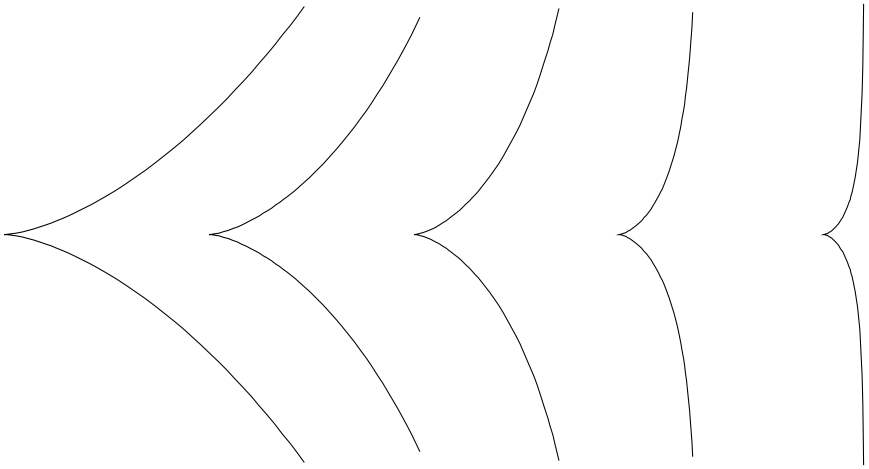


Figura 9.6: Cissóides de Diocles. Da esquerda para a direita, os parâmetros são $a = 10, t \in [-0.65, 0.65]$, $a = 5, t \in [-0.8, 0.8]$, $a = 2.5, t \in [-1, 1]$, $a = 1, t \in [-1.251.25]$ e $a = 0.5, t \in [-1.4, 1.4]$.

aplicarmos a conversão de coordenadas polares para rectangulares obtemos

$$(\rho^2 \cos^2 \phi + \rho^2 \sin^2 \phi)^2 = a^2(\rho^2 \cos^2 \phi - \rho^2 \sin^2 \phi)$$

Dividindo ambos os termos por ρ^2 e aplicando as identidades trigonométricas $\sin^2 x + \cos^2 x = 1$ e $\cos^2 x - \sin^2 x = \cos 2x$, obtemos

$$\rho^2 = a^2 \cos 2\phi$$

Esta equação é agora fácil de se converter para a forma paramétrica:

$$\begin{cases} \rho(t) = \pm a\sqrt{\cos 2t} \\ \phi(t) = t \end{cases}$$

Note-se que a presença do símbolo $\pm$ indica que, na verdade, se estão a traçar duas curvas em simultâneo. Para simplificar, vamos traçar cada uma destas curvas independentemente. Assim, vamos começar por definir uma função que calcula *meia* lemniscata com origem no ponto `p`:

```
meia_lemniscata_bernoulli(p, a, t0, t1, n) =
  map_division(t -> p + vpol(a*sqrt(cos(2*t)), t), t0, t1, n)
```

Em seguida, definimos uma função que desenha a lemniscata completa a partir do desenho de duas meias lemniscatas:

```
grafico_lemniscata_bernoulli(p, a, t0, t1, n) =
  spline([meia_lemniscata_bernoulli(p, +a, t0, t1, n)...,
          meia_lemniscata_bernoulli(p, -a, t1, t0, n)[2:end]...])
```

Podemos agora visualizar a "infinidade" produzida por esta curva através da seguinte expressão:

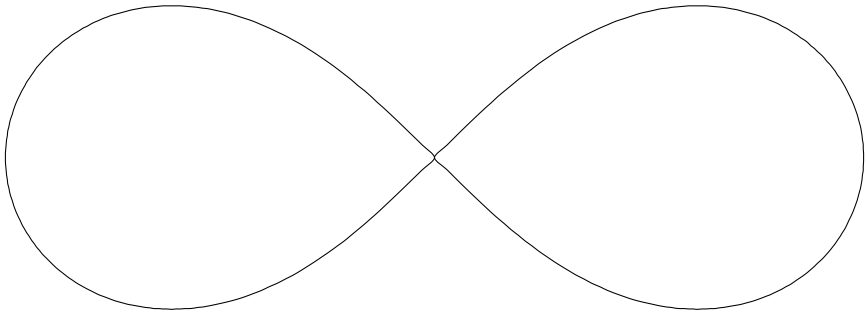


Figura 9.7: Lemniscata de Bernoulli.

```
grafico_lemniscata_bernoulli(xy(0, 0), 1.0, -pi/4, pi/4, 100)
```

O resultado está representado na Figura 9.7.

Exercício 9.4.4 Embora nos exemplos anteriores tenhamos empregue a representação paramétrica por esta ser a mais simples, nem sempre isso acontece. Por exemplo, a lemniscata de Gerono, estudada pelo matemático Camille-Christophe Gerono, define-se matematicamente pela equação

$$x^4 - x^2 + y^2 = 0$$

Resolvendo em ordem a y, obtemos

$$y = \pm\sqrt{x^2 - x^4}$$

Para permanecer no plano real, é necessário que $x^2 - x^4 \geq 0$, o que implica $x \in [-1, 1]$.

Para evitar ter de desenhar duas curvas (provocadas pela presença do símbolo $\pm$), podemos ser tentados a usar a representação paramétrica desta curva, definida por

$$\begin{cases} x(t) = \dfrac{t^2 - 1}{t^2 + 1} \\ y(t) = \dfrac{2t(t^2 - 1)}{(t^2 + 1)^2} \end{cases}$$

Infelizmente, enquanto a representação Cartesiana apenas precisa de fazer x variar entre -1 e $+1$, a representação paramétrica precisa de fazer t variar entre $-\infty$ e $+\infty$, o que nos levanta uma impossibilidade computacional. Por este motivo, neste caso a representação Cartesiana é preferível.

Tendo estas considerações em conta, defina a função `grafico_lemniscata_gerono` que desenha a lemniscata de Gerono.

9.4.4 Curva de Lamé

A curva de Lamé é uma curva cujos pontos satisfazem a equação

$$\left|\frac{x}{a}\right|^n + \left|\frac{y}{b}\right|^n = 1$$

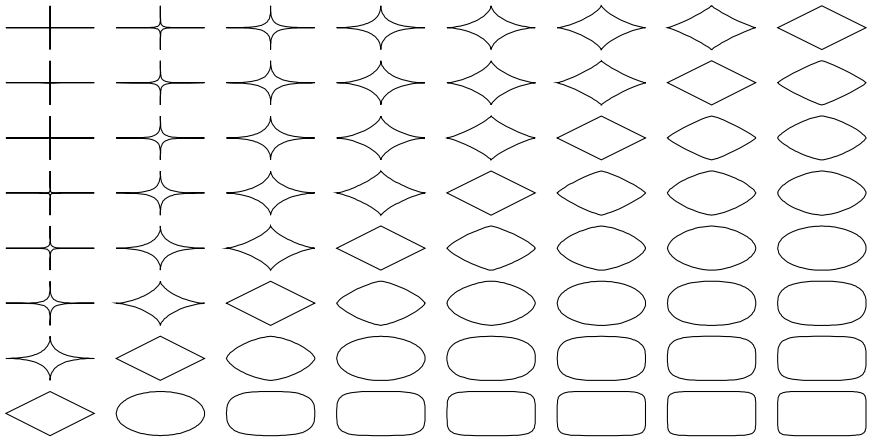


Figura 9.8: A curva de Lamé para $a = 2$, $b = 1$, $n = p/q$, $p, q \in 1..8$. A variável p varia ao longo do eixo das abcissas, enquanto q varia ao longo do eixo das ordenadas.

em que $n > 0$ e a e b são os raios da curva.

A curva foi estudada no século dezanove pelo matemático Francês Gabriel Lamé como uma generalização óbvia da elipse. Quando n é um número racional, a curva de Lamé diz-se uma *superelipse*. Quando $n > 2$, obtemos uma hiperelipse, tanto mais próxima de um rectângulo quanto maior for n. Quando $n < 2$ obtemos uma hipoelipse, tanto mais próxima de uma cruz quanto menor for n. Para $n = 2$, obtemos uma elipse e, para $n = 1$, obtemos um losango. Estas variações são visiveis na Figura 9.8 que apresenta variações desta curva para $a = 2$, $b = 1$ e diferentes valores de n.

A curva de Lamé tornou-se famosa quando foi proposta pelo cientista e poeta Dinamarquês Piet Hein como um compromisso estético e funcional entre formas baseadas em padrões rectangulares e formas baseadas em padrões curvilíneos. No seu estudo para uma intersecção de ruas no bairro Sergels Torg, em Estocolmo, em que se estava a hesitar entre uma rotunda tradicional ou um arranjo rectangular de vias, Piet Hein sugeriu a superelipse como forma intermédia entre aquelas duas, produzindo o resultado que está visível na Figura 9.9. De todas as superelipses, as mais esteticamente perfeitas eram, na opinião de Piet Hein, as parametrizadas por $n = \frac{5}{2}$ e $\frac{a}{b} = \frac{6}{5}$.

Para desenharmos a curva de Lamé é preferível utilizar a seguinte formulação paramétrica.

$$\begin{cases} x(t) = a\left(\cos^2 t\right)^{\frac{1}{n}} \cdot \operatorname{sgn}\cos t \\ y(t) = b\left(\sin^2 t\right)^{\frac{1}{n}} \cdot \operatorname{sgn}\sin t \end{cases} \qquad -\pi \leq t < \pi$$

A tradução desta fórmula para Julia é directa:

Figura 9.9: A superelipse proposta por Piet Hein para a praça Sergels, em Estocolmo. Fotografia de Nozzman.

```
sgn(x) =
  -1 ? x < 0 : 0 ? x == 0 : 1

superelipse(p, a, b, n, t) =
  p + vxy(a*(cos(t)^2)^(1/n)*sgn(cos(t)),
          b*(sin(t)^2)^(1/n)*sgn(sin(t)))
```

Para gerarmos uma sequência de pontos da superelipse podemos, como anteriormente, empregar a função `map_division`:

```
pontos_superelipse(p, a, b, n, pts) =
  [superelipse(p, a, b, n, t) for t in division(-pi, pi, pts, false)]
```

Finalmente, para traçar a curva da superelipse, podemos definir:

```
curva_superelipse(p, a, b, n, pts) =
  closed_spline(pontos_superelipse(p, a, b, n, pts))
```

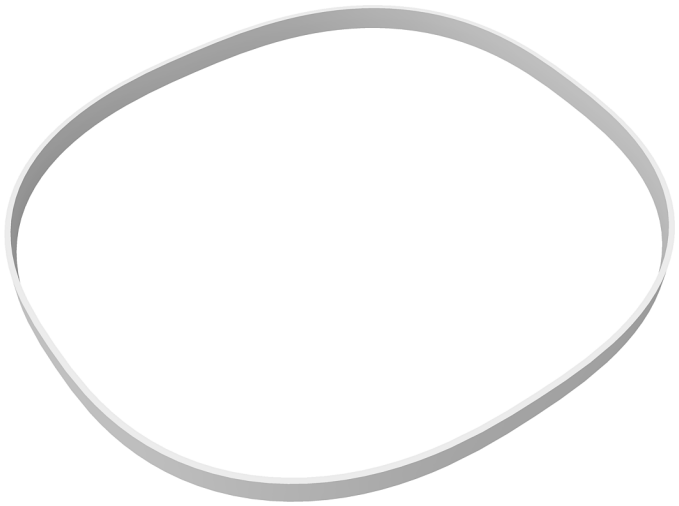
As superelipses da Figura 9.8 foram geradas usando a função anterior. Para isso, definimos uma função `testa_superelipses` que, para valores de a e b e para uma sequência de números, testa todas as combinações de expoente $n = p/q$, com p e q a tomarem sucessivos valores dessa sequência. Para se poder visualizar as várias superelipses cada uma tem uma origem proporcional à combinação p e q usada:

```
testa_superelipses(a, b, ns) =
  for num in ns
    for den in ns
      curva_superelipse(xy(num*2.5*a, den*2.5*b), a, b, num/den, 100)
    end
  end
```

A Figura 9.8 foi gerada directamente a partir da avaliação da seguinte expressão:

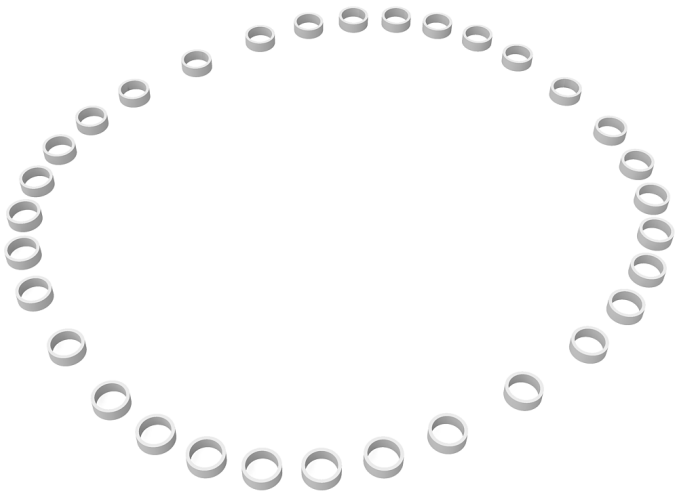
```
testa_superelipses(2.0, 1.0, [1, 2, 3, 4, 5, 6, 7, 8])
```

Exercício 9.4.5 Defina a função `tanque_superelipse` que constrói um tanque de forma superelíptica idêntico ao da praça de Sergels Torg, tal como se apresenta em seguida:



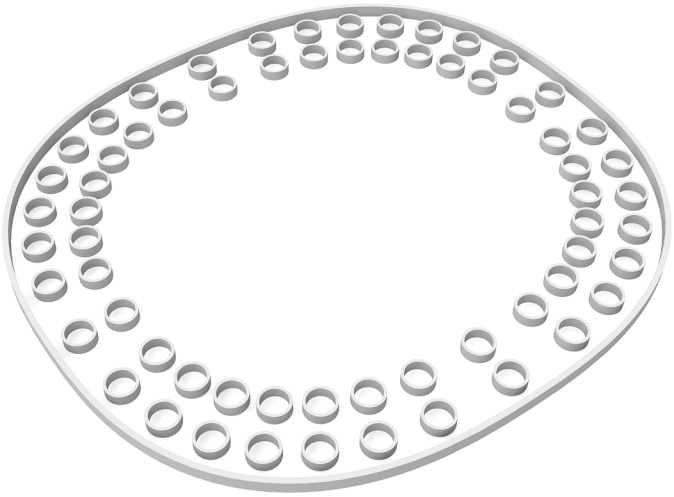
Sugestão: defina previamente uma função denominada `parede_curva` que constrói paredes ao longo de uma determinada curva. A função deverá receber a espessura e altura da parede a construir e uma entidade representando a curva (por exemplo, um círculo, uma *spline*, etc). Considere também a utilização da função `sweep` definida na secção 6.6.2.

Exercício 9.4.6 Defina a função `tanques_circulares` que constrói uma sucessão de tanques circulares cujos centros estão localizados ao longo de uma superelipse, tal como se apresenta na imagem seguinte:

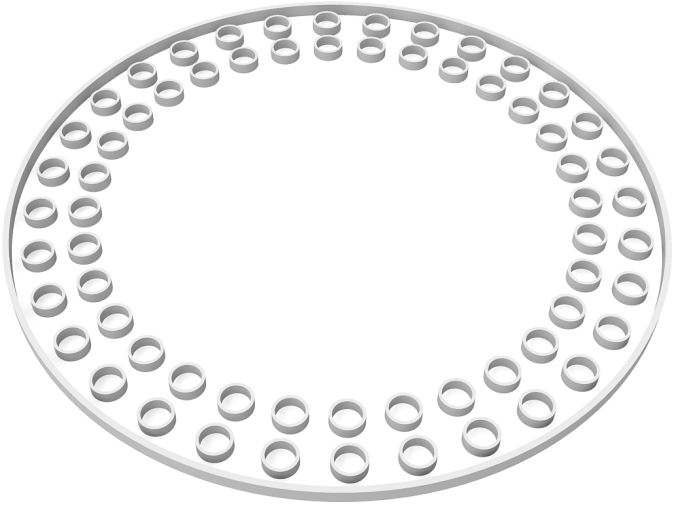


A função deverá receber os parâmetros da superelipse, os parâmetros de um tanque circular e o número de tanques a criar.

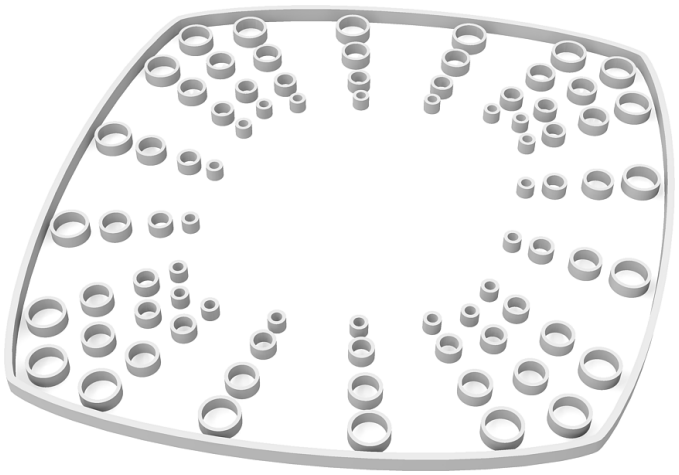
Exercício 9.4.7 Defina uma função que, invocando as funções `tanque_superelipse` e `tanques_circulares`, cria uma praça tão semelhante quanto possível à praça de Sergels Torg, tal como se apresenta na imagem seguinte:



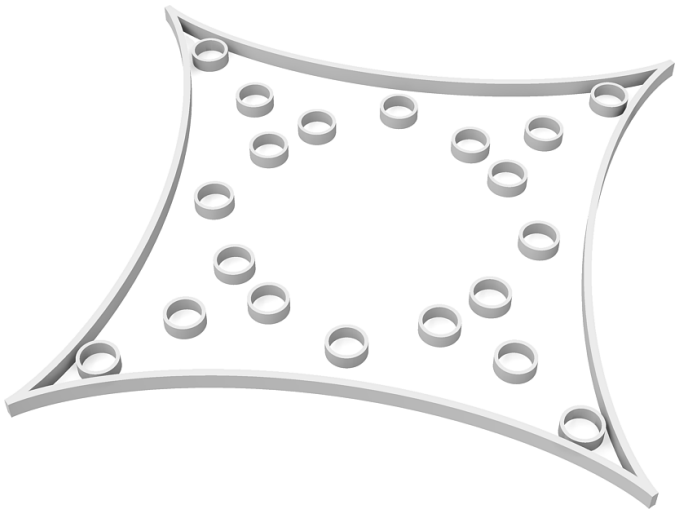
Exercício 9.4.8 Modifique a função anterior para produzir uma variante perfeitamente circular:



Exercício 9.4.9 Modifique a função anterior para produzir a seguinte variante:



Exercício 9.4.10 Modifique as funções anteriores para produzir a seguinte variante:



9.5 Curvas Espaciais

Todas as curvas que vimos nas secções anteriores são *planares*, pois encontram-se inteiramente contidas num plano. Quando uma curva não está contida num plano diz-se *espacial*.

Do ponto de vista da modelação que temos feito, uma curva espacial não tem nada de particularmente diferente de uma curva planar, excepto o facto de empregar três equações paramétricas, em vez de duas como acontecia com as planares.

A título de exemplo, consideremos uma *hélice*, uma curva muito usada em Arquitectura, por exemplo, em escadas. Esta curva tem uma descrição

paramétrica definida por

$$(\rho, \phi, z) = (1, t, t)$$

Esta definição mostra que à medida que o ângulo ϕ aumenta, também a cota z aumenta.

Para flexibilizar a construção de hélices, podemos parameterizá-la um pouco mais, acrescentando alguns coeficientes a cada uma das equações paramétricas:

$$\begin{cases} \rho(t) = r \\ \phi(t) = \alpha t \\ z(t) = \beta t \end{cases}$$

É fácil vermos que o parâmetro α representa o número de "voltas" (i.e., múltiplos de 2π) que a hélice vai dar e β representa a "velocidade" com que a hélice "avança" ao longo do eixo da hélice (neste caso, o eixo Z).[6] A função Julia correspondente será então:

```
pontos_helice(p, r, a, b, n) =
  map_division(ti -> p + vcyl(r, a*ti, b*ti), 0, 2*pi, n)
```

As seguintes expressões produzem diferentes hélices:

```
spline(pontos_helice(x(0), 2, 4, 1, 200))
spline(pontos_helice(x(5), 1, 8, 1, 400))
spline(pontos_helice(x(9), 2, 2, 1, 100))
```

O resultado da avaliação está representado da Figura 9.10.

Uma das vantagens da utilização da representação paramétrica é a facilidade com que podemos fazer alterações. Por exemplo, se resolvermos tornar as coordenadas ρ, ϕ e z em funções lineares de t,

$$\begin{cases} \rho(t) = r_0 + r_1 t \\ \phi(t) = \alpha_0 + \alpha_1 t \\ z(t) = \beta_0 + \beta_1 t \end{cases}$$

passamos a ter uma *hélice cónica*. O parâmetro r_0 representa o raio da abertura inicial da hélice cónica enquanto r_1 representa a "velocidade" de abertura da hélice cónica. Os parâmetros α_0 e β_0 representam valores iniciais para o ângulo ϕ e para a altura z. A definição Julia correspondente é:

```
pontos_helice_conica(p, r0, r1, a0, a1, b0, b1, n) =
  map_division(ti -> p + vcyl(r0 + r1*ti, a0 + a1*ti, b0 + b1*ti),
               0, 2*pi, n)
```

Logicamente, a hélice simples é um caso particular da hélice cónica, em que temos $r_0 = r$, $r_1 = 0$, $\alpha_0 = 0$, $\alpha_1 = \alpha$, $\beta_0 = 0$, e $\beta_1 = \beta$.

[6] Ao "avanço" da hélice no período 2π denomina-se o *passo* da hélice.

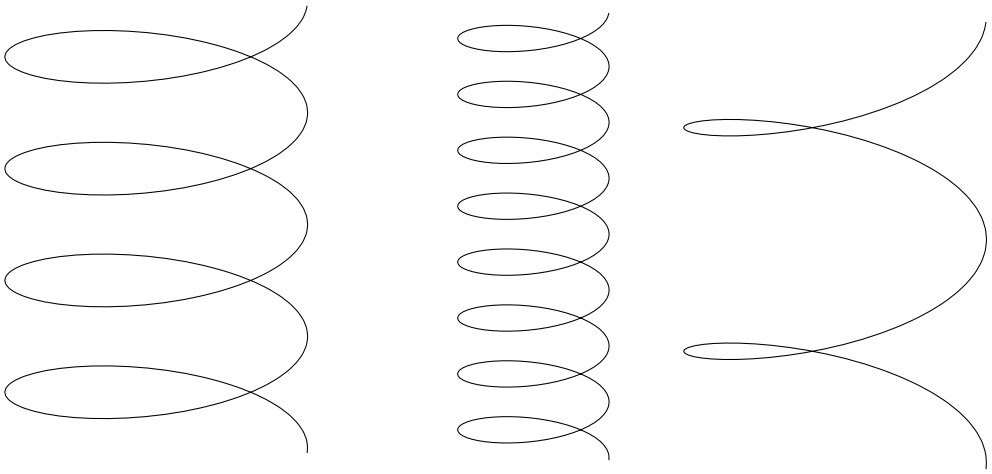


Figura 9.10: Três hélices. Da esquerda para a direita, os parâmetros são $r = 2, 1, 2$, $\alpha = \{4, 8, 2\}$, e $\beta = \{1, 1, 1\}$.

Usando a hélice cónica torna-se fácil construir uma das curvas mais famosas dos tempos modernos: a *dupla hélice* que representa a estrutura molecular do ADN e que foi descoberta por James D. Watson e Francis Crick na década de cinquenta do século passado. Para isso, basta avaliar duas expressões:

```
spline(pontos_helice_conica(u0(), 1, 0, 0, 3, 0, 1, 100))
spline(pontos_helice_conica(u0(), 1, 0, pi, 3, 0, 1, 100))
```

9.6 Precisão

Consideremos a curva Borboleta, proposta pelo matemático Temple H. Fay. Esta curva periódica define-se pela equação em coordenadas polares

$$r = e^{\sin\phi} - 2\cos(4\phi) + \sin^5\left(\frac{2\phi - \pi}{24}\right)$$

À semelhança do que fizemos anteriormente, vamos considerar a curva relativamente a um ponto de partida P, dentro de um intervalo de variação do parâmetro independente $t \in [t_0, t_1]$.

```
curva_borboleta(p, t0, t1, n) =
  map(t -> p+vpol(exp(sin(t))-2*cos(4*t)+sin((2*t-pi)/24)^5, t),
      division(t0, t1, n, false))
```

Para o desenho desta curva falta-nos agora determinar o valor adequado do parâmetro n. Para isso, é importante percebermos que o período da curva Borboleta é de 24π, sendo a curva completa gerada para

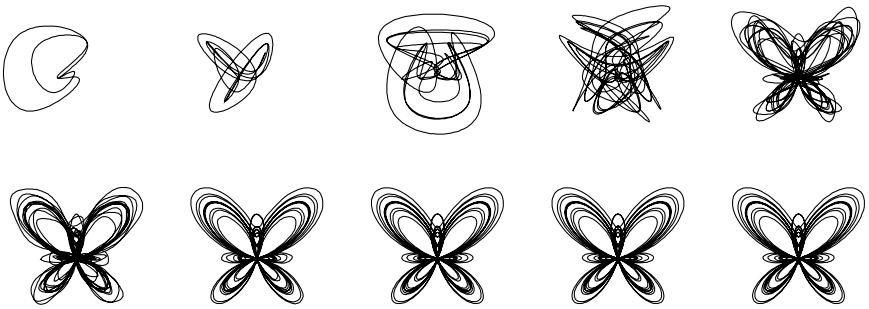


Figura 9.11: A curva Borboleta produzida para valores crescentes do número n de pontos.

$t \in [0, 24\pi]$. Uma vez que este período é relativamente grande, o número n de pontos a empregar deverá ser suficiente para que se consiga reproduzir fielmente a curva e é precisamente aqui que surge a maior dificuldade: sem conhecer a forma da curva, é difícil estimar qual o melhor valor para n. Se o valor de n for demasiado baixo, a curva não será representada com fidelidade, em particular, nos locais em que existem curvaturas muito acentuadas; se o valor for demasiado alto, maior será a necessidade de recursos computacionais. Na prática, é necessária alguma experimentação para perceber qual o melhor valor.

Esta necessidade de experimentação relativamente ao valor do número de pontos a empregar está bem patente na Figura 9.11 que mostra as curvas Borboleta desenhada para os seguintes crescentes valores de n:

```
closed_spline(curva_borboleta(xy(0, 10), 0, 24*pi, 10))
closed_spline(curva_borboleta(xy(10, 10), 0, 24*pi, 20))
closed_spline(curva_borboleta(xy(20, 10), 0, 24*pi, 40))
closed_spline(curva_borboleta(xy(30, 10), 0, 24*pi, 80))
closed_spline(curva_borboleta(xy(40, 10), 0, 24*pi, 160))
closed_spline(curva_borboleta(xy(0, 0), 0, 24*pi, 320))
closed_spline(curva_borboleta(xy(10, 0), 0, 24*pi, 640))
closed_spline(curva_borboleta(xy(20, 0), 0, 24*pi, 1280))
closed_spline(curva_borboleta(xy(30, 0), 0, 24*pi, 2560))
closed_spline(curva_borboleta(xy(40, 0), 0, 24*pi, 5120))
```

Como podemos verificar, quando o número de pontos é insuficiente, as curvas produzidas podem ser grotescas distorções da realidade. Por este motivo, devemos ter um particular cuidado na escolha do valor adequado para este parâmetro.

Uma alternativa interessante será deixar ao computador a responsabilidade de adaptar o número de pontos às necessidades de cada curva.

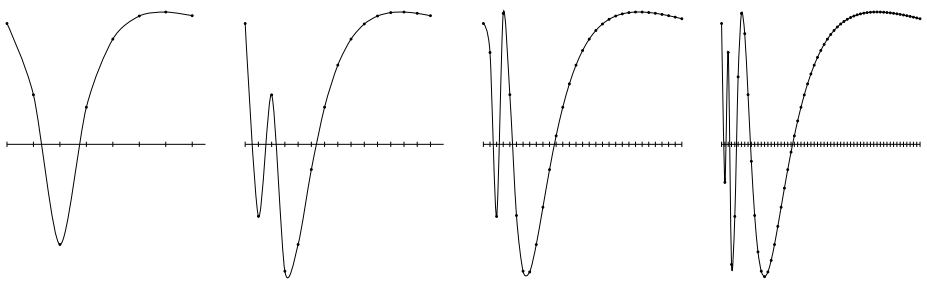


Figura 9.12: Gráfico da função $f(x) = \frac{1}{2}sin(\frac{1}{x})$ no intervalo $[0.05, 0.8]$. Da esquerda para a direita empregaram-se, respectivamente, $8, 15, 30$ e 60 pontos de amostragem.

9.6.1 Amostragem Adaptativa

A função `map_division` permite-nos gerar a curva de qualquer função $f(t)$ no intervalo $[t_0, t_1]$ através da aplicação da função f a uma sequência de n incrementos de t igualmente espaçados $\{t_0, \ldots, t_1\}$. A esta sequência dá-se o nome de sequência de amostragem.

É fácil vermos que quanto maior for o número de elementos da sequência de amostragem, maior será a precisão da curva produzida. Infelizmente, quanto maior for essa precisão, maior será o esforço dispendido na computação do valor da função em todos os elementos da sequência. Na prática, é necessário encontrarmos um equilíbrio entre a precisão e o esforço computacional.

Infelizmente, existem variadíssimas funções para as quais é difícil ou mesmo impossível encontrar um número de pontos de amostragem apropriado. A título de exemplo, experimentemos produzir o gráfico da função $f(t) = (t, \frac{1}{2}sin(\frac{1}{t}))$ no intervalo $[0.05, 0.8]$. Esta função tem como gráfico uma curva que oscila com uma frequência que é tanto maior quanto mais próximos estamos da origem. A Figura 9.12 mostra uma sequência de gráficos desta função em que, da esquerda para a direita, o número de pontos de amostragem foi sendo progressivamente aumentado.

Como podemos ver na figura, com poucos pontos de amostragem a curva resultante é uma grosseira aproximação da curva real e é apenas quando o número de pontos de amostragem se torna muito elevado que se consegue obter alguma precisão. Acontece que essa precisão apenas é relevante nas zonas da curva onde há grandes variações. Nas outras zonas, onde a curva evolui suavemente, mais pontos de amostragem apenas implicam maior tempo de processamento, sem qualquer vantagem significativa para a qualidade do resultado. Isto sugere que devemos adaptar o número de pontos de amostragem ao longo da curva de acordo com a sua suavidade: nas zonas onde a curva varia mais bruscamente devemos

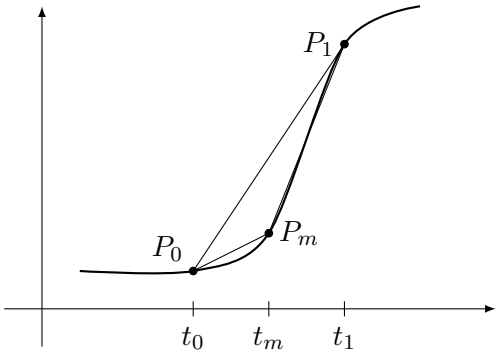


Figura 9.13: Geração adaptativa de pontos

empregar mais pontos de amostragem, mas nas zonas onde a variação for mais linear, podemos diminuir o número de pontos de amostragem. Desta forma, o número de pontos de amostragem adapta-se à forma da curva.

Para que possamos empregar uma amostragem adaptativa precisamos de definir um critério para classificar a "suavidade" de uma curva. Tal como ilustramos na Figura 9.13, podemos começar por admitir que temos duas abcissas x_0 e x_1 que usamos para calcular os pontos P_0 e P_1. Para sabermos se a curva se comporta de forma linear entre esses dois pontos (podendo assim ser aproximada por um segmento de recta), vamos calcular o valor médio x_m desse intervalo e o correspondente ponto P_m da curva. Se a função for realmente linear nesse intervalo, então os pontos P_0, P_m e P_1 serão colineares. Na prática, isso raramente acontecerá, pelo que teremos de empregar um conceito de colinearidade *aproximada*, o que podemos fazer empregando qualquer um de vários critérios possíveis como, por exemplo:

- A área do triângulo P_0, P_m, P_1 ser inferior a um valor ϵ suficientemente pequeno.

- O ângulo P_0, P_m, P_1 ser aproximadamente igual a π.

- A soma da distância entre P_0 e P_m com a distância entre P_m e P_1 ser aproximadamente igual à distância entre P_0 e P_1.

Outros critérios serão igualmente aplicáveis, mas, por agora, vamos escolher o primeiro. A sua tradução para Julia é a seguinte:

```
aproximadamente_colineares(p0, pm, p1, epsilon) =
  let a = distance(p0, pm),
      b = distance(pm, p1),
      c = distance(p1, p0)
    area_triangulo(a, b, c) < epsilon
  end

area_triangulo(a, b, c) =
  let s = (a + b + c)/2.0
    sqrt(s*(s - a)*(s - b)*(s - c))
  end
```

A função `aproximadamente_colineares` implementa o critério que nos permite dizer que os segmentos de recta que ligam os pontos `p0`, `pm`, e `p1` constituem uma boa aproximação ao comportamento da curva entre esses pontos. Quando este critério não se verifica, podemos partir o intervalo em dois sub-intervalos e analisar cada um deles em separado, concatenando os resultados.

```
map_division_adapt(f, t0, t1, epsilon) =
  let p0 = f(t0),
      p1 = f(t1),
      tm = (t0 + t1)/2.0,
      pm = f(tm)
    if aproximadamente_colineares(p0, pm, p1, epsilon)
      [p0, pm, p1]
    else
      [map_division_adapt(f, t0, tm, epsilon)...,
       map_division_adapt(f, tm, t1, epsilon)[2:end]...]
    end
  end
```

Esta forma adaptativa de produzir uma amostragem de uma curva permite resultados com muito maior precisão mesmo empregando um número total de pontos de amostragem relativamente pequeno. A Figura 9.14 mostra uma sequência de gráficos idênticos aos apresentados na Figura 9.12, mas agora empregando um esquema de amostragem adaptativo com aproximadamente o mesmo número de pontos de amostragem em cada caso. Como podemos verificar, os gráficos são substancialmente mais precisos, em particular nas zonas perto da origem, sem que isso diminua visivelmente a qualidade nas zonas da curva cuja variação é mais suave.

Exercício 9.6.1 O algoritmo usado na função `map_division_adapt` não consegue lidar correctamente com funções periódicas em que temos $f(t_0) = f(t_1) = f(\frac{t_0+t_1}{2})$. Neste caso, o triangulo degenerado formado por estes três pontos tem área zero, o que leva o algoritmo a terminar de imediato. Modifique a função de modo que ela receba um parâmetro adicional Δ_t que determina o maior intervalo admissível entre dois valores consecutivos de t.

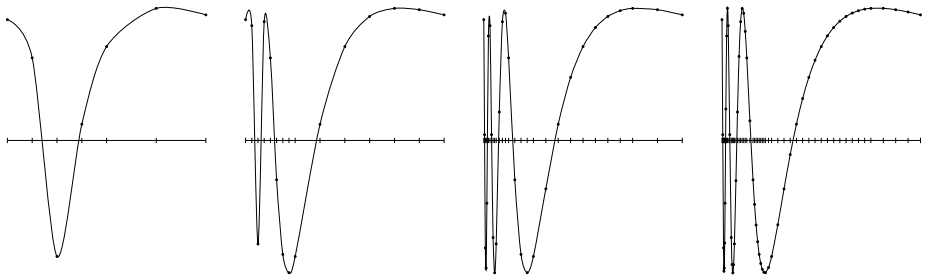


Figura 9.14: Gráfico da função $f(x) = \frac{1}{2}sin(\frac{1}{x})$ no intervalo $[0.05, 0.8]$. Da esquerda para a direita, a sequência de amostragem adaptativa possui, respectivamente, $7, 15, 29$ e 57 pontos.

Exercício 9.6.2 A função `map_division_adapt` não é tão eficiente quanto seria possível, em particular, porque repete sistematicamente a aplicação da função f no ponto médio: $f(t_m)$ é calculado para decidir se continua a recursão e, quando continua, terá de ser calculado novamente para permitir calcular $f(t_0)$ ou $f(t_1)$ na invocação seguinte. Redefina a função `map_division_adapt` de modo a evitar cálculos repetidos.

9.7 Superfícies Paramétricas

Ao longo da história, a arquitectura tem explorado uma variedade enorme de formas. Como vimos, algumas dessas formas correspondem a sólidos bem conhecidos desde a antiguidade, como cubos, esferas e cones. Mais recentemente, os arquitectos têm virado a sua atenção para formas bastante menos clássicas, que exigem um tipo de descrição diferente. O famoso arquitecto espanhol Félix Candela é um bom exemplo.

Candela explorou exaustivamente as propriedades do parabolóide hiperbólico, permitindo-lhe criar obras que são hoje referências no mundo da arquitectura. A Figura 9.15 ilustra a construção de uma das obras de Félix Candela, onde o elemento fundamental é precisamente uma fina casca de betão armado com a forma de um parabolóide hiperbólico. No seu tempo, Candela realizava manualmente todos os cálculos necessários para determinar as formas que pretendia realizar. Iremos agora ver como podemos obter os mesmos resultados de modo bastante mais eficiente.

Para isso, vamos generalizar as descrições paramétricas de curvas de modo a permitirem a especificação de superfícies. Embora uma curva seja descrita por um triplo de funções de um único parâmetro t, por exemplo, $(x(t), y(t), z(t))$, no caso de uma superfície esta terá de ser descrita por um triplo de funções de dois parâmetros que variam independentemente,[7] por

[7]Matematicamente falando, uma superfície paramétrica é simplesmente a imagem de

Figura 9.15: A capela de Lomas de Cuernavaca em construção. Fotografia de Doroty Candela.

exemplo, $(x(u,v), y(u,v), z(u,v))$. Naturalmente, a escolha de coordenadas rectangulares, cilíndricas, esféricas ou outras é apenas uma questão de conveniência: em qualquer caso, serão necessárias três funções.

Tal como acontece com as curvas, embora possamos descrever uma superfície na forma implícita, a representação paramétrica é mais adequada à geração das posições por onde passa a superfície. Por exemplo, uma esfera com centro no ponto (x_0, y_0, z_0) e raio r, pode ser descrita implicitamente pela equação

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 = 0$$

mas esta forma não permite a geração directa de coordenadas, sendo preferível adoptar uma descrição paramétrica, pelas funções

$$\begin{cases} x(\phi, \psi) = x_0 + r \sin\phi \cos\psi \\ y(\phi, \psi) = y_0 + r \sin\phi \sin\psi \\ z(\phi, \psi) = z_0 + r \cos\phi \end{cases}$$

Para ilustrarmos o conceito de superfície paramétrica, vamos considerar uma famosa superfície: a Faixa de Möbius.[8]

uma função injectiva de $\mathbb{R}^2$ em $\mathbb{R}^3$.

[8] A Faixa de Möbius não é uma verdadeira superfície, mas sim uma *superfície com fronteira*.

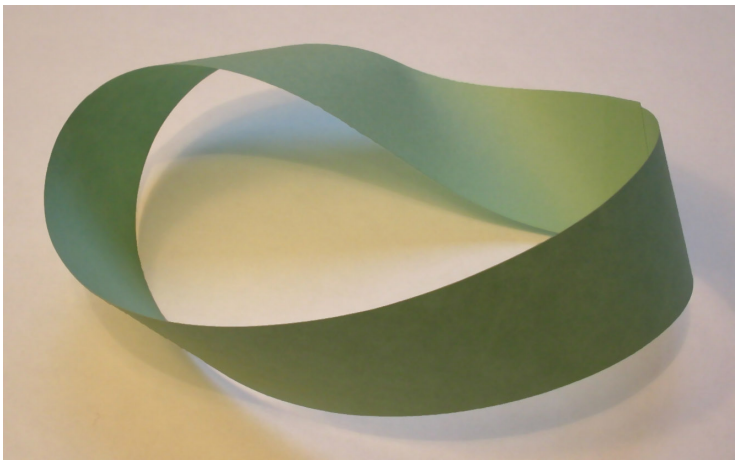


Figura 9.16: A faixa de Möbius.

9.7.1 A Faixa de Möbius

A faixa de Möbius (também conhecida por *banda* de Möbius) foi descrita de forma independente por dois matemáticos alemães em 1858, primeiro, por Johann Benedict Listing e, dois meses mais tarde, por August Ferdinand Möbius. Embora tenha sido apenas Listing a publicar a sua descoberta, esta acabou por ser baptizada com o nome de Möbius. A Figura 9.16 apresenta uma imagem de uma faixa de Möbius feita com uma tira de papel e onde se torna possível perceber uma das extraordinárias características desta superfície: é possível percorrê-la inteiramente sem nunca sair do mesmo "lado" pois, na verdade, a faixa de Möbius tem apenas um lado.[9] Do mesmo modo, a superfície está limitada por uma aresta apenas, que se estende a toda a superfície. Na verdade, a faixa de Möbius é a mais simples superfície de um só lado e com uma só aresta.

A faixa de Möbius tem sido sucessivamente usada nos mais variados contextos, desde os famosos trabalhos de Maurits Cornelis Escher que empregam a faixa de Möbius, como "Möbius Strip II," até ao símbolo representativo da *reciclagem*, que é constituído por três setas dispostas ao longo de uma faixa de Möbius, tal como se pode ver na Figura 9.17.

As equações paramétricas desta superfície, em coordenadas cilíndricas, são as seguintes:

$$\begin{cases} \rho(u,v) = 1 + v\cos\dfrac{u}{2} \\ \theta(u,v) = u \\ z(u,v) = v\sin\dfrac{u}{2} \end{cases}$$

[9] Esta propriedade tem sido explorada, por exemplo, em correias de transmissão que, quando adoptam a topologia de uma faixa de Möbius, desgastam ambos os "lados" da correia, em vez de desgastarem apenas um, permitindo assim durarem o dobro do tempo.

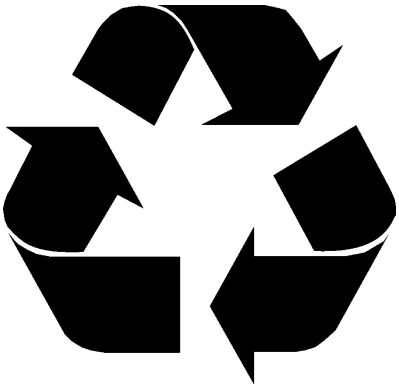


Figura 9.17: O símbolo representativo da *reciclagem*: três setas dispostas ao longo de uma faixa de Möbius.

Dada a periodicidade das funções trigonométricas, temos $\frac{u}{2} \in [0, 2\pi]$ ou $u \in [0, 4\pi]$. Nesse período, para um dado v, a coordenada z assumirá valores simétricos em relação ao plano xy, no limite, entre $-v$ e v. Isto quer dizer que a faixa de Möbius se desenvolve para cima e para baixo do plano xy.

Para desenharmos esta superfície podemos usar, como abordagem inicial, uma generalização do processo que empregámos para desenhar curvas paramétricas. Neste processo, limitavamo-nos a aplicar a função argumento $f(t)$ ao longo de uma sucessão de n valores desde t_0 até t_1. No caso de superfícies paramétricas, pretendemos aplicar a função $f(u, v)$ ao longo uma sucessão de m valores, desde u_0 até u_1 e, para cada um, ao longo de uma sucessão de n valores, desde v_0 até v_1. Assim, a função $f(u, v)$ será aplicada em $m \times n$ pares de valores. Este comportamento é facilmente obtido à custa de dois mapeamentos encadeados:

```
map(u -> map(v -> f(u, v),
             division(v0, v1, n)),
    division(u0, u1, m))
```

Por exemplo, no caso da faixa de Möbius, podemos definir:

```
faixa_moebius(u0, u1, m, v0, v1, n) =
  map(u -> map(v -> cyl(1 + v*cos(u/2), u, v*sin(u/2)),
               division(v0, v1, n)),
      division(u0, u1, m))
```

Na realidade, esta combinação de mapeamentos com divisões de intervalos é tão frequente que o Khepri a disponibiliza através de uma extensão da função `map_division`, permitindo escrever

```
faixa_moebius(u0, u1, m, v0, v1, n) =
  map_division((u, v) -> cyl(1 + v*cos(u/2), u, v*sin(u/2)),
               u0, u1, m,
               v0, v1, n)
```

Vimos que no caso de uma função de um só parâmetro $f(t)$, tínhamos que

```
map_division(f, t0, t1, n)
```

era equivalente a

```
map(f,
    division(t0, t1, n))
```

Essa equivalência é agora estendida ao caso de funções de dois parâmetros $f(s,t)$, ou seja,

```
map_division(f, u0, u1, n, v0, v1, m)
```

é equivalente a

```
map(u -> map(v -> f(u, v)
             division(v0, v1, m)),
    division(u0, u1, n))
```

Naturalmente, se o mapeamento interior produz uma lista com os resultados da aplicação da função f, o mapeamento exterior irá produzir uma lista com as listas de resultados da aplicação da função f. Admitindo que cada aplicação da função f produz um ponto em coordenadas tri-dimensionais, facilmente vemos que esta expressão produz uma lista de listas de pontos. Assim sendo, o resultado será da forma:

```
[[P0,0, P0,1, ..., P0,m],
 [P1,0, P1,1, ..., P1,m],
 ...,
 [Pn,0, Pn,1, ..., Pn,m]]
```

Para desenharmos esta lista de listas de pontos, podemos simplesmente aplicar a função `spline` a cada uma das listas de pontos:

```
splines(ptss) =
  map(spline, ptss)
```

Assim, podemos facilmente apresentar uma primeira visualização da faixa de Möbius:

```
splines(faixa_moebius(0, 4*pi, 80, 0, 0.3, 10))
```

O resultado da avaliação da expressão anterior está representado da Figura 9.18.

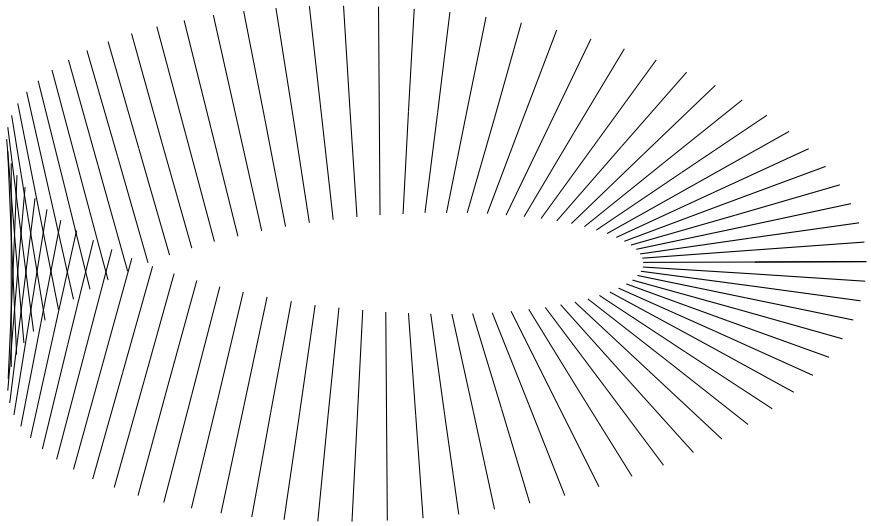


Figura 9.18: A faixa de Möbius.

Um aspecto interessante das superfícies paramétricas tri-dimensionais é que o conjunto de pontos que as definem pode ser organizado numa matriz. De facto, se a superfície paramétrica é obtida através da aplicação da função $f(s,t)$ ao longo uma sucessão de m valores, desde s_0 até s_1 e, para cada um, ao longo de uma sucessão de n valores, desde t_0 até t_1, podemos colocar os resultados dessas aplicações numa matriz de m linhas e n colunas, tal como se segue:

$$\begin{bmatrix} f(s_0,t_0) & \dots & f(s_0,t_1) \\ \dots & \dots & \dots \\ f(s_1,t_0) & \dots & f(s_1,t_1) \end{bmatrix}$$

O que a função `map_division` devolve não é mais do que uma representação desta matriz, implementada em termos de uma lista de linhas da matriz, cada linha implementada em termos de uma lista de valores correspondentes aos elementos dessa linha da matriz.

É agora fácil vermos que a aplicação da função `splines` à representação da matriz não faz mais do que desenhar m *splines*, cada uma definida pelos n pontos de cada uma das m linhas da matriz. Outra alternativa será desenhar n *splines* definidas pelos m pontos de cada uma das n colunas da matriz. Para isso, o mais simples é *transpor* a matriz, i.e., trocar as linhas pelas colunas, ficando esta com a forma:

$$\begin{bmatrix} f(s_0,t_0) & \dots & f(s_1,t_0) \\ \dots & \dots & \dots \\ f(s_0,t_1) & \dots & f(s_1,t_1) \end{bmatrix}$$

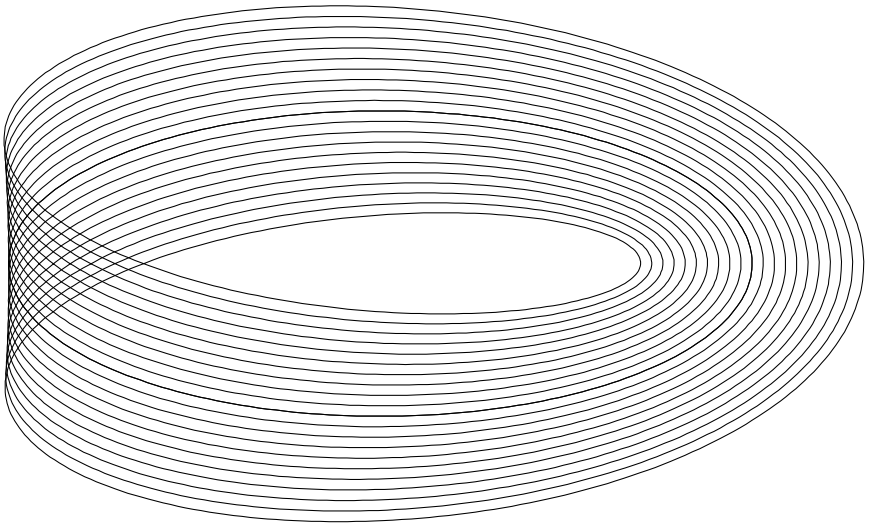


Figura 9.19: A faixa de Möbius.

Para isso, podemos definir uma função que, dada uma matriz implementada como uma lista de listas, devolve a transposta dessa matriz na mesma implementação:

```
matriz_transposta(matrix) =
  [[row[i] for row in matrix]
   for i in 1:length(matrix[1])]
```

Usando esta função, é agora possível traçar curvas ortogonais às representadas na Figura 9.18 simplesmente traçando as curvas obtidas a partir da matriz transposta, i.e.,

```
splines(matriz_transposta(faixa_moebius(0, 4*pi, 80, 0, 0.3, 10)))
```

O resultado encontra-se na Figura 9.19.

Finalmente, para obtermos uma *malha* com o efeito combinado de ambas as curvas, podemos definir uma função auxiliar `malha_splines`:

```
malha_splines(ptss) =
  begin
    splines(ptss)
    splines(matriz_transposta(ptss))
  end
```

Podemos agora usar esta função tal como se segue:

```
malha_splines(faixa_moebius(0, 4*pi, 80, 0, 0.3, 10))
```

O resultado encontra-se na Figura 9.20.

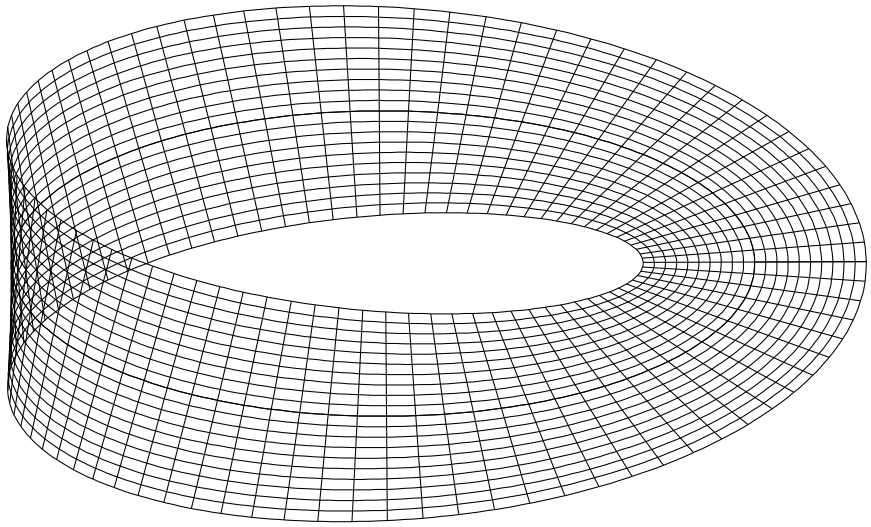


Figura 9.20: A faixa de Möbius.

Este género de representação, baseado na utilização de linhas para dar a ilusão de um objecto, denomina-se *modelo de arame* (*wire frame*, em Inglês). Formalmente, um modelo de arame não é mais que um conjunto de linhas que representa uma superfície. Os modelos de arame têm a vantagem, sobre outras formas de visualização, de permitirem um desenho muito mais rápido.

9.8 Superfícies

Os modelos de arame não constituem superfícies. De facto, por as linhas serem infinitamente finas e não existir qualquer "material" entre elas, não é possível associar um modelo de arame a uma verdadeira superfície.

Se o que pretendemos é, de facto, desenhar superfícies, então é preferível explorar as capacidades do Khepri para a criação de superfícies.

As malhas poligonais são aglomerados de polígonos, geralmente triângulos e quadriláteros, que definem a forma de um objecto. Relativamente aos modelos constituídos apenas por linhas, as malhas poligonais têm a vantagem de poderem ser mais realisticamente visualizadas, por exemplo, com remoção de superfícies invisíveis, com inclusão de sombreados, etc. Obviamente, cada face destas malhas poligonais possui espessura zero, pelo que não são verdadeiros sólidos mas apenas representações abstractas de superfícies. Ainda assim, podem ser muito úteis para se obter uma mais correcta visualização de uma superfície e podem ser subsequentemente transformadas para a criação de sólidos, por exemplo, através da função `thicken` que confere uma espessura uniforme à superfície.

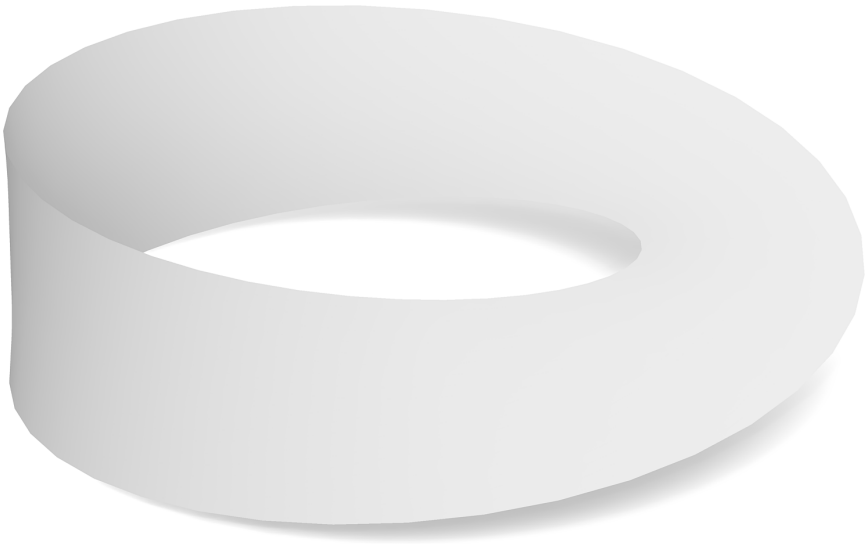


Figura 9.21: A faixa de Möbius.

O Khepri disponibiliza uma operação para a criação destas malhas poligonais denominada `surface_grid`. A Figura 9.21 apresenta uma representação realística da faixa de Möbius gerada a partir da seguinte expressão:

```
surface_grid(faixa_moebius(0, 4*pi, 80, 0, 0.3, 10))
```

É agora possível experimentar variações na faixa de Möbius, por exemplo, para fazer variar a "largura" da faixa, tal como se representa na Figura 9.22.

Para visualizarmos um exemplo mais arquitectónico, vamos considerar o parabolóide hiperbólico empregue por Félix Candela na Capela de Lomas de Cuernavaca. Esta superfície pode ser descrita na sua representação

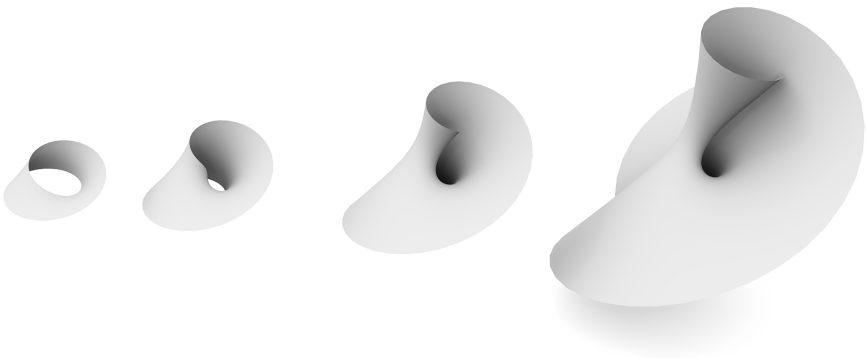


Figura 9.22: A faixa de Möbius para diferentes "larguras." Da esquerda para a direita o intervalo de variação em v é de $[0, \frac{1}{2}]$, $[0, 1]$, $[0, 2]$ e $[0, 4]$.

implícita através da equação

$$x^2 - y^2 - z = 0$$

ou, equivalentemente, através da sua representação paramétrica:

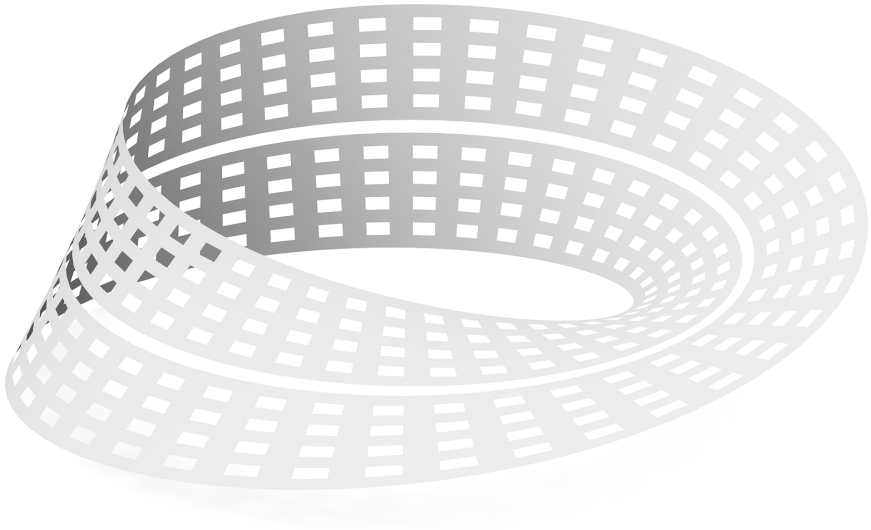
$$\begin{cases} x(s,t) = s \\ y(s,t) = t \\ z(s,t) = s^2 - t^2 \end{cases}$$

A tradução para Khepri é:

```
thicken(
  surface_grid(
    map_division((x, y) -> xyz(x, y, x*x - y*y),
                 -0.5, 1.3, 40, -2, 2, 80)),
  0.03)
```

Na expressão anterior, usámos os limites do domínio que aproximam a superfície da obra real e demos espessura à superfície através da função `thicken`. Na Figura 9.23 mostramos uma visualização da obra onde eliminámos a porção que fica debaixo de terra.

Exercício 9.8.1 Defina uma função que, com os argumentos adequados, constrói a superfície representada na seguinte Figura.



Exercício 9.8.2 Considere a seguinte figura:

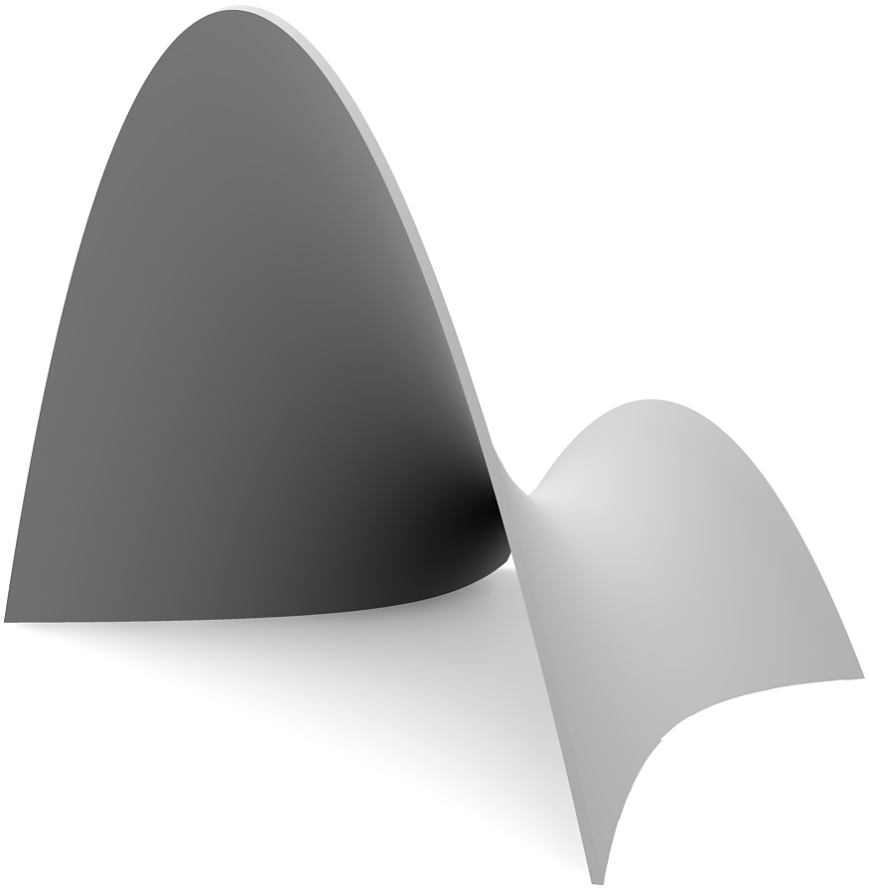
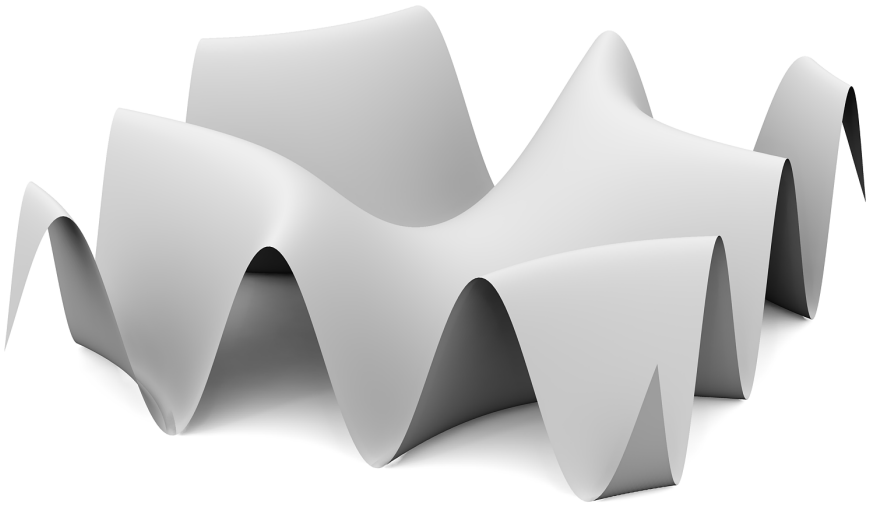


Figura 9.23: Uma aproximação da Capela de Lomas de Cuernavaca.



Assuma que a figura está centrada na origem e representada usando uma perspectiva isométrica. Repare que as intersecções da superfície com qualquer plano paralelo ao plano XZ ou ao plano YZ produz uma sinusóide perfeita (embora produza sinusóides de frequência diferente em função da distância do plano de intersecção à origem). A partir destas dicas, tente descobrir qual as equações paramétricas que lhe deram origem.

9.8.1 Helicoide

Discutimos, na secção 9.5, o desenho da hélice. Vamos agora discutir a superfície que generaliza aquela curva: a *helicóide*.

A helicóide foi descoberta em 1776 por Jean Baptiste Meusnier. As suas equações paramétricas (em coordenadas cilíndricas) são:

$$\begin{cases} \rho(u,v) = u \\ \theta(u,v) = \alpha v \\ z(u,v) = v \end{cases}$$

A Figura 9.24 mostra duas helicóides com parâmetros diferentes, gerados a partir do seguinte fragmento de programa:

```
helicoide(p, a, u0, u1, m, v0, v1, n) =
  map_division((u, v) -> p + vcyl(u, a*v, v),
               u0, u1, m, v0, v1, n)

surface_grid(helicoide(xyz(0, 0, 0), 1, 0, 1, 10, 0, 2*pi, 100))
surface_grid(helicoide(xyz(7, 0, 0), 3, 0, 5, 50, 0, 2*pi, 200))
```

Uma característica interessante da helicóide é que por cada ponto da helicóide passa uma hélice.

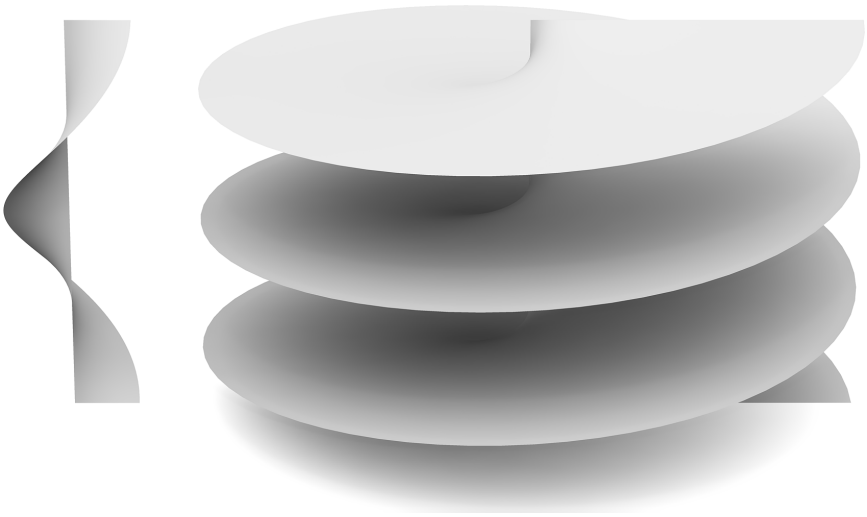


Figura 9.24: Duas helicóides com intervalo de variação em v de $[0, 2\pi]$. A helicóide da esquerda tem $\alpha = 1$ e intervalo de variação em u de $[0, 1]$ enquanto que a da direita tem $\alpha = 3$ e $u \in [0, 5]$.

9.8.2 Mola

Uma *mola* é outra figura geométrica que possui afinidades com uma hélice. Matematicamente, a mola é o volume gerado por um círculo que se desloca ao longo de uma hélice. Em termos mais simples, uma mola é uma hélice com "espessura," i.e., um tubo que se enrola ao longo de um eixo. Se for r_0 o raio do tubo, r_1 a distância do eixo do tubo ao eixo da hélice, α o ângulo de rotação inicial da mola e v_z a "velocidade" ao longo do eixo Z, as equações paramétricas da mola são:

$$\begin{cases} \rho(u, v) = r_1 + r_0 \cos u \\ \theta(u, v) = \alpha + v \\ z(u, v) = \dfrac{v_z v}{\pi} + r_0 \sin u \end{cases}$$

A tradução da definição anterior para Julia fica:

```
mola(p, a, r0, r1, vz, u0, u1, m, v0, v1, n) =
  map_division((u, v) -> p+vcyl(r1+r0*cos(u), a+v, vz*v/pi+r0*sin(u)),
               u0, u1, m, v0, v1, n)
```

Na Figura 9.25 encontra-se a visualização de uma mola no seu processo de extensão. Cada uma das imagens corresponde a uma mola com as mesmas dimensões, excepto no que diz respeito à "velocidade" em z.

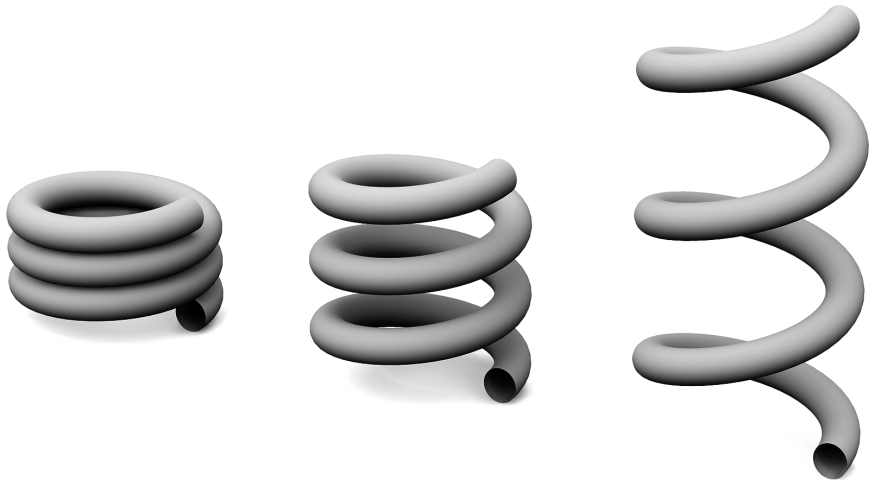
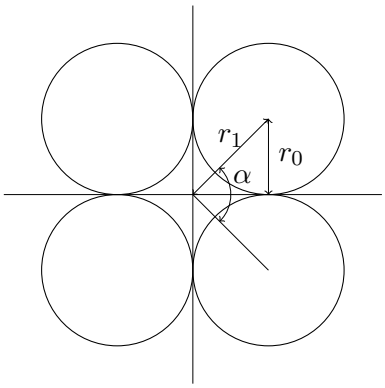


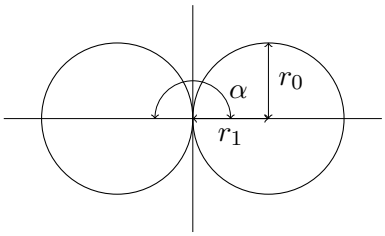
Figura 9.25: Três molas construídas com os parâmetros $r_0 = 1$, $r_1 = 5$, $u \in [0, 2\pi]$ e $v \in [0, 6\pi]$. Da esquerda para a direita, temos a velocidade $v_z \in \{1, 2, 4\}$.

```
surface_grid(
    mola(xyz(0, 0, 0), 0, 1, 5, 1, 0, 2*pi, 50, 0, 3*2*pi, 150))
surface_grid(
    mola(xyz(20, 0, 0), 0, 1, 5, 2, 0, 2*pi, 50, 0, 3*2*pi, 150))
surface_grid(
    mola(xyz(40, 0, 0), 0, 1, 5, 4, 0, 2*pi, 50, 0, 3*2*pi, 150))
```

Exercício 9.8.3 A mola tradicional é apenas a forma mais simples que é possível gerar a partir da função `mola`. Formas mais elaboradas podem ser geradas quer através da parametrização, quer através da composição. Por exemplo, através da composição de "molas" podemos gerar *cordas*. Neste caso, tratamos as "molas" como "fios" ou "filamentos" que, enrolados uns com os outros, constituem uma corda.

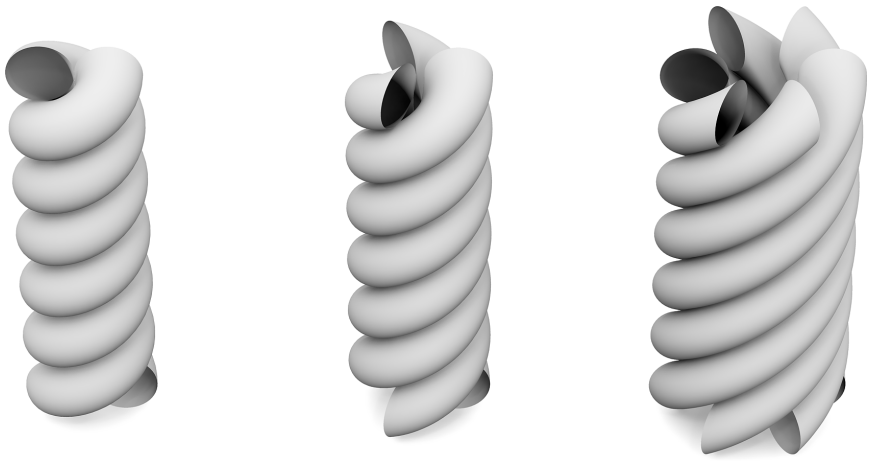
Assim, uma corda pode ser vista como um conjunto de molas entrelaçadas, com todas as molas a desenvolverem-se no mesmo eixo mas "rodadas" (i.e., com um ângulo inicial) e com um passo (i.e., uma velocidade de desenvolvimento) que lhes permite ficarem encostadas umas às outras. Na Figura seguinte estão representados dois esquemas que mostram, à esquerda, o posicionamento das molas no caso de uma corda de quatro "fios." Esta corda pode ser generalizada para qualquer número de fios, até o limite inferior de dois fios, representado à direita.





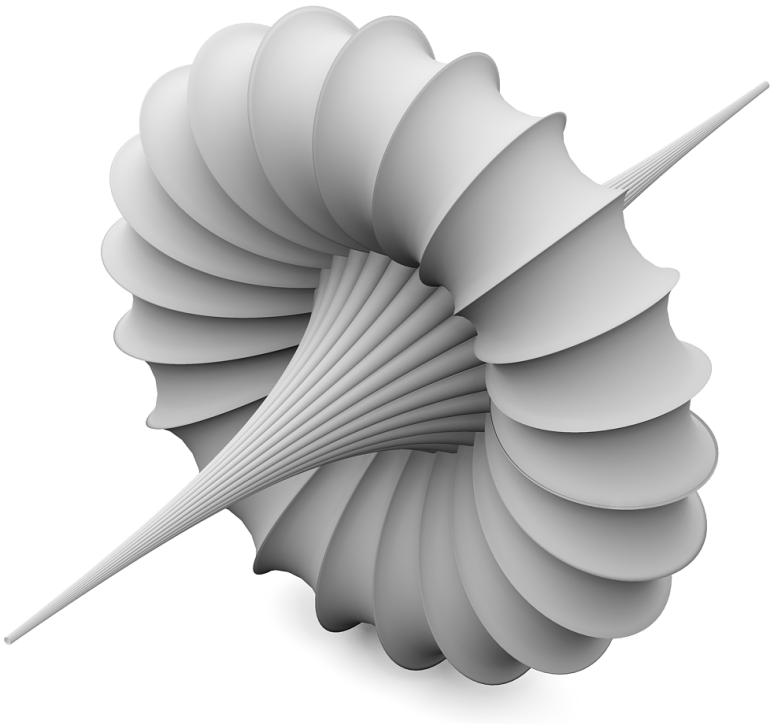
Defina a função `corda` que, recebendo os parâmetros adequados, cria cordas. A título de exemplo, considere as seguintes expressões cuja avaliação produz as três cordas construídas com dois, três e seis fios apresentadas em seguida:

```
corda(xyz(0, 0, 0), 2, 1, 3)
corda(xyz(10, 0, 0), 3, 1, 2)
corda(xyz(20, 0, 0), 6, 1, 1)
```



Exercício 9.8.4 Um *breather* é uma superfície matemática que caracteriza um tipo particular de onda e que se encontra exemplificada na figura seguinte. A sua equação paramétrica é:

$$\begin{cases} x(u,v) = -u + \dfrac{2\left(1-a^2\right)\cosh(au)\sinh(au)}{a\left(\left(1-a^2\right)\cosh^2(au) + a^2\,\sin^2\left(\sqrt{1-a^2}v\right)\right)} \\ y(u,v) = \dfrac{2\sqrt{1-a^2}\cosh(au)\left(-\sqrt{1-a^2}\cos(v)\cos\left(\sqrt{1-a^2}v\right) - \sin(v)\sin\left(\sqrt{1-a^2}v\right)\right)}{a\left(\left(1-a^2\right)\cosh^2(au) + a^2\,\sin^2\left(\sqrt{1-a^2}v\right)\right)} \\ z(u,v) = \dfrac{2\sqrt{1-a^2}\cosh(au)\left(-\sqrt{1-a^2}\sin(v)\cos\left(\sqrt{1-a^2}v\right) + \cos(v)\sin\left(\sqrt{1-a^2}v\right)\right)}{a\left(\left(1-a^2\right)\cosh^2(au) + a^2\,\sin^2\left(\sqrt{1-a^2}v\right)\right)} \end{cases}$$



Note que, dada a periodicidade das funções trigonométricas, temos $-2\pi \leq u \leq 2\pi$ e $-12\pi \leq v \leq 12\pi$. O parâmetro a caracteriza diferentes superfícies e pode variar entre 0 e 1.

Defina a função `breather` que desenha a superfície em questão a partir de um ponto P, do parâmetro b e dos extremos de variação e do número de pontos a considerar ao longo dessa variação, respectivamente, para os parâmetros u e v. Por exemplo, a figura anterior foi produzida pela avaliação da expressão:

```
surface_grid(
    breather(xyz(0, 0, 0), 0.4, -13.2, 13.2, 200, -37.4, 37.4, 200))
```

9.8.3 Conchas

Vários autores defendem que a Natureza é uma manifestação matemática e justificam essa crença pela surpreendente semelhança que algumas superfícies matemáticas apresentam com formas naturais.

A superfície de Dini, é um desses casos. As equações paramétricas que a definem são:

$$\begin{cases} x(u,v) = \cos(u)\sin(v) \\ y(u,v) = \sin(u)\sin(v) \\ z(u,v) = \cos(v) + \log(\tan(\frac{v}{2})) + a * u \end{cases}$$

em que $0 \leq u \leq 2\pi$ e $0 \leq v \leq \pi$.

A tradução destas equações para Julia é directa:

```
dini(p, a, u0, u1, m, v0, v1, n) =
  map_division((u, v) -> p + vxyz(cos(u)*sin(v),
                                  sin(u)*sin(v),
                                  cos(v) + log(tan(v/2.0)) + a*u),
               u0, u1, m,
               v0, v1, n)
```

As seguintes invocações desta função mostram como é possível, de facto, simular as formas naturais de certo tipo de conchas. O resultado encontra-se representado na Figura 9.26.

```
surface_grid(
    dini(xyz(0, 0, 0), 0.2, 0, 6*pi, 100, 0.1, pi*0.5, 100))
surface_grid(
    dini(xyz(3, 0, 0), 0.2, 0, 6*pi, 100, 0.1, pi*0.7, 100))
surface_grid(
    dini(xyz(6, 0, 0), 0.2, 0, 6*pi, 100, 0.1, pi*0.9, 100))
```

Um outro exemplo é o da concha em caracol, definida matematicamente (em Julia) por:

```
concha(p, a, b, u0, u1, m, v0, v1, n) =
  let f(u, v) =
        let e = exp(u/6.0/pi),
            c = cos(v/2.0)^2
          p + vxyz(a*(1 - e)*cos(u)*c,
                   b*(e - 1)*sin(u)*c,
                   1 - exp(u/3.0/pi) - sin(v) + e*sin(v))
        end
    map_division(f, u0, u1, m, v0, v1, n)
  end
```

A partir da função anterior, podemos experimentar variações como as que se apresentam em seguida e que geram as superfícies representadas na Figura 9.27.

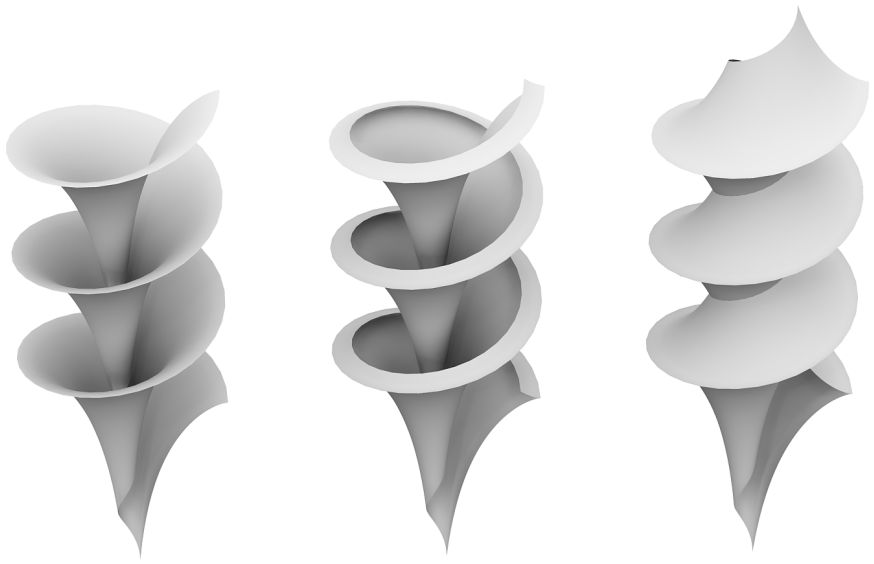


Figura 9.26: A superfície de Dini como aproximação matemática da forma de certo tipo de conchas.

```
surface_grid(
    concha(xyz(0, 0, 0), 1, 1, 0, 7*pi, 100, 0, 2*pi, 100))
surface_grid(
    concha(xyz(7, 0, 0), 2, 2, 0, 7*pi, 100, 0, 2*pi, 100))
surface_grid(
    concha(xyz(18, 0, 0), 3, 1, 0, 7*pi, 100, 0, 2*pi, 100))
```

9.8.4 Cilindros, Cones, e Esferas

Embora tenhamos ilustrado a secção anterior com superfícies relativamente complexas, nesta secção vamos começar por fazer superfícies comparativa-

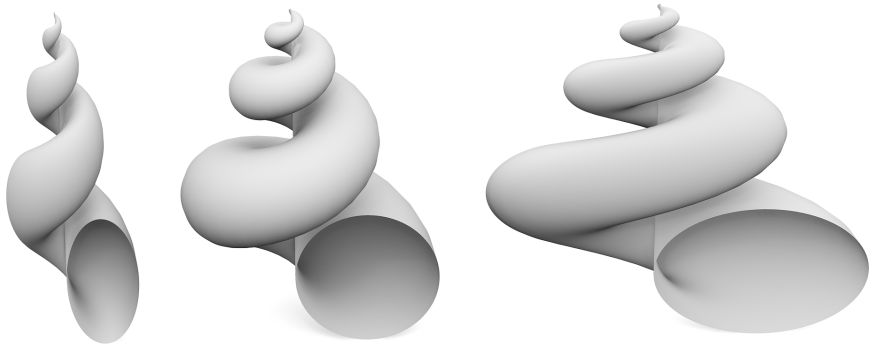


Figura 9.27: A aproximação matemática da forma das conchas em caracol.

mente muito mais simples, o que nos permitirá perceber que a construção paramétrica de superfícies é relativamente fácil desde que compreendamos o impacto da variação dos parâmetros independentes u e v.

Por exemplo, a superfície de um cilindro de raio r caracteriza-se por ser o conjunto de pontos que estão à distância fixa r de um eixo e entre uma "altura" mínima h_0 e máxima h_1 relativamente a esse eixo.

Admitindo, por simplicidade, que vamos fazer coincidir o eixo do cilindro com o eixo Z, isto quer dizer que podemos empregar coordenadas cilíndricas (ρ, ϕ, z), ficando a superfície do cilindro definida simplesmente fixando $\rho = r$ e deixando ϕ variar entre 0 e 2π e z variar entre h_0 e h_1. Estando ρ fixo e sendo as variações de ϕ e z independentes, a utilização de superfícies paramétricas é simples: basta usar um dos parâmetros u ou v para fazer a variação de ϕ e o outro para a variação de z.

Mais especificamente, vamos fazer $\rho(u, v) = r$ e vamos deixar $\phi(u, v) = u$ e $z(u, v) = v$ variar independentemente, respectivamente entre $u \in [0, 2\pi]$ e $v \in [h_0, h_1]$. Uma vez que ϕ depende directa e exclusivamente de u e z depende directa e exclusivamente de v, podemos usar estes parâmetros directamente na função. Desta forma, estamos em condições de escrever uma função que produza uma coordenada cilíndrica a partir do valor do raio r e a partir dos parâmetros independentes ϕ e z.

A imagem da esquerda da Figura 9.28 mostra o cilindro de raio 1 entre $h_0 = -1$ e $h_1 = 1$, produzido pela avaliação da seguinte expressão:

```
surface_grid(
    map_division(
        (fi, z) -> cyl(1, fi, z),
        0, 2*pi, 60,
        -1, 1, 20))
```

Se, ao invés de fixarmos o raio $\rho(fi, z) = r$, o fizermos variar em função da altura z, então é óbvio que o raio do "cilindro" será zero na origem e aumentará (em valor absoluto) à medida que nos afastamos da origem. A única diferença para o caso anterior será então na expressão que cria as coordenadas cilíndricas que, agora, será `cyl(z, fi, z)`, reflectindo o aumento linear do raio com z. Evidentemente, a figura resultante não será um cilindro mas antes o cone, que apresentamos na imagem central da Figura 9.28.

Finalmente, consideremos a descrição de uma esfera de raio r em coordenadas esféricas (ρ, ϕ, ψ), que se reduz a $\rho = r$, com ϕ a variar entre 0 e 2π e ψ a variar entre 0 e π. Mais uma vez, necessitamos de duas variações independentes, pelo que podemos atribuir arbitrariamente u a uma delas e v à outra. Assim, vamos fazer $\rho(u, v) = r$, $\phi(u, v) = u$ e $\psi(u, v) = v$ com $u \in [0, 2\pi]$ e $v \in [0, \pi]$. Tal como fizémos nos dois exemplos anteriores, esta associação entre ϕ e ψ e, respectivamente, u e v permite-nos escrever a função directamente em termos dos primeiros. Exemplificando com $r = 1$, temos:

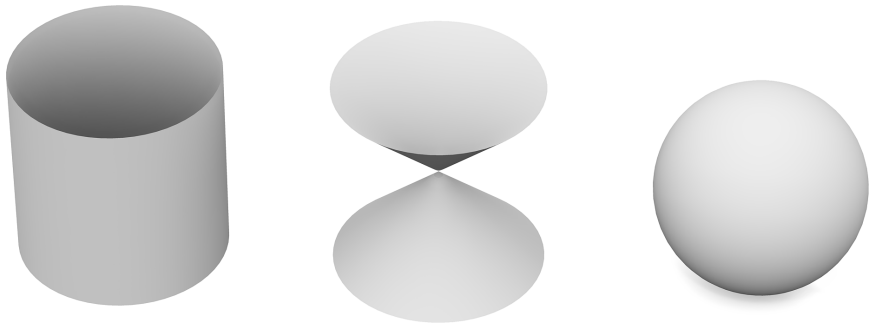


Figura 9.28: Um cilindro, um cone e uma esfera gerados como superfícies paramétricas.

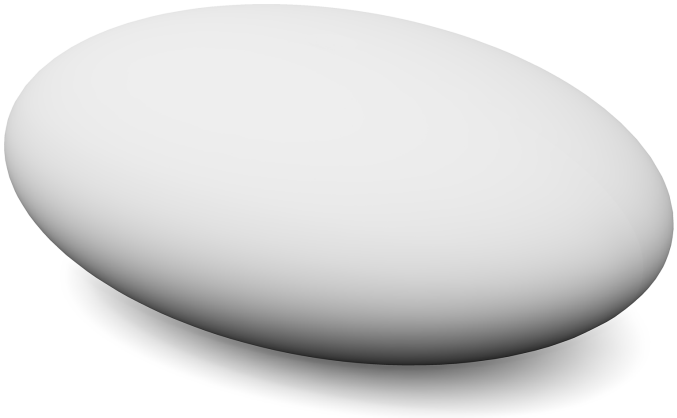
```
surface_grid(
    map_division(
        (fi, psi) -> sph(1, fi, psi),
        0, 2*pi, 60,
        0, pi, 30))
```

O resultado encontra-se na imagem à direita da Figura 9.28.

Para vermos o impacto que pequenas alterações nos parâmetros podem ter na forma das superfícies geradas, vamos experimentar três variações na forma da esfera. A primeira variação consiste em adicionar ao raio da esfera uma sinusóise de elevada frequência e reduzida amplitude, função da "latitude" ψ, por exemplo, fazendo $\rho = 1 + \frac{\sin(20\psi)}{20}$. O resultado será a formação de umas ondas de pólo a pólo, tal como é visível na imagem da esquerda da Figura 9.29. Se, por outro lado, resolvermos fazer a mesma variação no raio ficar dependente de uma sinusóide idêntica, mas função de ϕ, então já as ondas marcham ao longo da "longitude", por exemplo, de este para oeste. Este efeito está ilustrado na imagem central da Figura 9.29. Finalmente, podemos obter um efeito combinado fazendo o raio depender simultaneamente de uma sinusóide função da "latitude" e de outra sinusóide função da "longitude," i.e., $\rho = 1 + \frac{\sin(20\phi)}{20} + \frac{\sin(20\psi)}{20}$. O resultado encontra-se na imagem à direita, na Figura 9.29.

Exercício 9.8.5 Considere o seguinte elipsóide:

Figura 9.29: Esferas cujo raio é função da "latitude," da "longitude" ou de ambos.



Um elipsóide é caracterizado pelas dimensões dos seus três raios ortogonais a, b, e c. A sua equação paramétrica é:

$$\begin{aligned} x &= a \sin\psi \cos\phi \\ y &= b \sin\psi \sin\phi \\ z &= c \cos\psi \end{aligned}$$

$$-\tfrac{\pi}{2} \leq \phi \leq +\tfrac{\pi}{2}; \quad -\pi \leq \psi \leq +\pi;$$

Defina a função `elipsoide` que produz o elipsóide a partir dos três raios a, b, c e, ainda, do número n de valores a usar ao longo de ϕ e de ψ.

9.9 A Adega Ysios

A adega Ysios é um edifício desenhado por Santiago Calatrava e destinado à produção, armazenamento e distribuição de vinhos. A Figura 9.30 apresenta uma vista frontal do edifício.

O edifício tem duas paredes longitudinais de forma sinusoidal cujo remate superior é também de forma sinusoidal. Estas paredes têm 196 metros

Figura 9.30: Adega Ysios. Fotografia de Luis Antonio Ortuño.

de comprimento e estão afastadas uma da outra 26 metros ao longo da linha média. A cobertura tem a forma de uma onda feita de paralelipípedos que se apoiam nos topos sinusoidais das paredes. Note-se que os topos das paredes onde se apoiam os paralelipípedos estão geralmente a alturas diferentes, pois as duas sinusoides dos topos estão em oposição de fase uma em relação à outra.[10]

Como se pode observar na Figura 9.30, a parte central da fachada sul, destinada a albergar o centro de visitas, é mais saliente e mais elevada que o resto do edifício.

Tal como se pode ver na Figura 9.31, os paralelipípedos da cobertura são de alumínio, em contraste com a madeira de cedro que é usada na fachada sul. Na fachada norte emprega-se betão com algumas pequenas aberturas para iluminação. As fachadas este e oeste são constituídas por placas de alumínio ondulado.

Nesta secção vamos abordar a modelação da parede frontal da adega. Numa primeira análise, podemos modelar essa parede como a extrusão vertical de uma sinusóide. Infelizmente, embora esta operação consiga modelar correctamente as extremidades laterais da parede, já não consegue fazê-lo na parte central, pois, nessa região, a parede tem uma evolução mais complexa, afastando-se da vertical.

Uma segunda abordagem poderá ser a interpolação de regiões, mas esta só será realizável se conseguirmos modelar as curvas que delimitam a parede. Para além disso, poderá ser necessário um número elevado de curvas

[10] Oposição de fase significa que, quando uma das sinusóides atinge o seu valor máximo, a outra atinge o seu valor mínimo e vice-versa. Para que isto aconteça, a diferença entre as fases das duas sinusóides tem de ser π.

Figura 9.31: Adega Ysios. Fotografia de Koikile.

para conseguirmos representar fielmente a superfície da parede.

Uma terceira abordagem, mais rigorosa, consiste na descrição matemática da superfície em questão. Desta forma, as "curvas" necessárias poderão ser geradas automaticamente. Embora possa ser difícil perceber à primeira vista que a parede da adega Ysios pode ser modelada por uma função matemática, vamos ver que é relativamente simples proceder a essa modelação através da composição e modificação de funções mais simples.

Como primeira aproximação, podemos começar por considerar uma parede sinusóidal idêntica à que obteríamos se fizessemos a extrusão vertical de uma sinusóide. Para simplificar, vamos centrar a base da parede na origem, o que implica que a bossa central tenha a sua amplitude máxima na origem. Uma vez que o seno tem amplitude zero na origem, isto sugere que, ao invés de um seno, seja preferível empregar um cosseno, cuja amplitude é máxima na origem. A observação desta obra de Calatrava indica que este cosseno realiza três ciclos para cada lado, pelo que o período deverá variar entre -6π e $+6\pi$. Admitindo que esta parede evolui ao longo do plano XZ, podemos começar por experimentar a seguinte expressão:

```
surface_grid(
    map_division(
        (x, z) -> xyz(x,
                      cos(x),
                      z),
        -7*pi, 7*pi, 100,
        0, 5, 10))
```

A expressão mostra que fazemos a coordenada x variar livremente com

u, fazemos a coordenada y ser o cosseno de x e, finalmente, fazermos a coordenada z variar livremente com v. Como podemos observar pela Figura 9.32, a superfície é uma boa aproximação das extremidades laterais da parede frontal da adega Ysios.

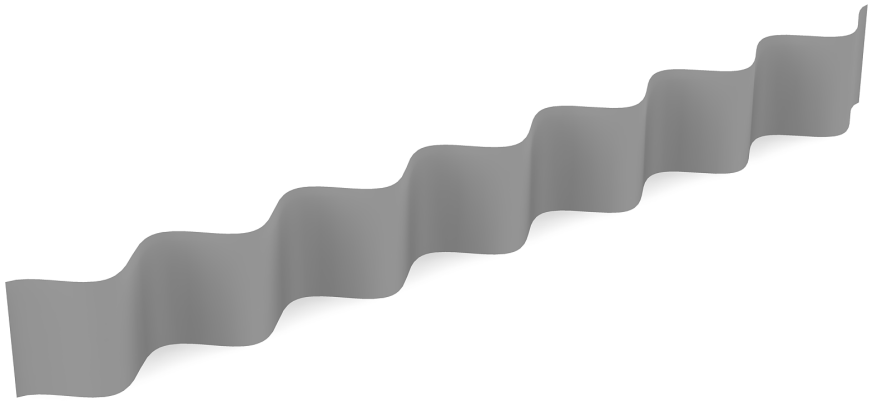


Figura 9.32: Primeira aproximação à modelação da Adega Ysios.

Vamos agora modelar as sinusóides do topo da parede, onde se apoiam os paralelipípedos da cobertura. Para isso, vamos simplesmente "deformar" a superfície, fazendo com que a coordenada z vá oscilando de forma cossinusóidal ao longo do eixo X, aumentando a amplitude da oscilação à medida que subimos em z. Para que a parede tenha a sua base livre da influência dessa cossinosoide, vamos elevar o cosseno de uma unidade e vamos empregar uma reduzida amplitude inicial:

```
surface_grid(
    map_division(
        (x, z) -> xyz(x,
                      cos(x),
                      z*(1 + 0.2*cos(x))),
        -7*pi, 7*pi, 100,
        0, 5, 10))
```

O resultado da modelação anterior está representado na Figura 9.33.

Para modelar o aumento de amplitude que a "bossa" central vai tendo com a altura podemos considerar aumentar a amplitude do cosseno à medida que aumenta a cota z, fazendo:

```
surface_grid(
    map_division(
        (x, z) -> xyz(x,
                      (z + 3)*cos(x),
                      z*(1 + 0.2*cos(x))),
        -7*pi, 7*pi, 100,
        0, 5, 10))
```

O resultado, visível na Figura 9.34, mostra que a evolução da parte cen-

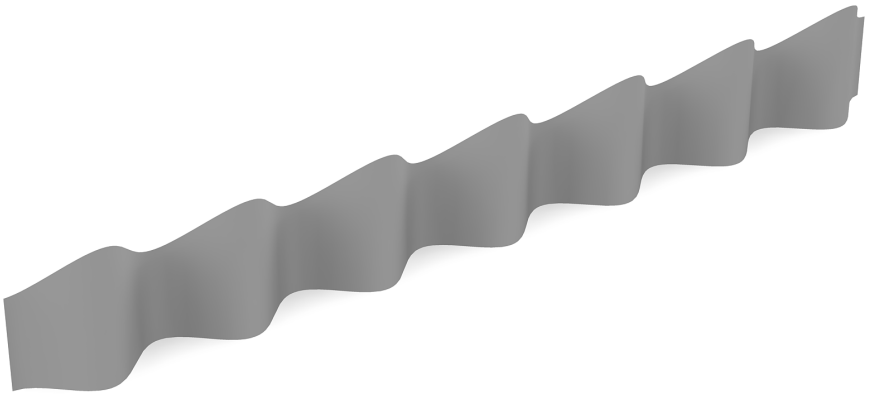


Figura 9.33: Segunda aproximação à modelação da Adega Ysios.

tral está já correctamente modelada, mas é preciso evitar que esta evolução afecte também as partes laterais. Para isso, basta-nos fazer uma combinação entre duas evoluções distintas, uma para a região central da parede, correspondente a meio ciclo do cosseno, i.e., a uma gama de variação em x entre $-\frac{\pi}{2}$ e $+\frac{\pi}{2}$, e outra para tudo o resto. Temos de ter em conta que para compensar a diferença de declive provocada pelo aumento da coordenada y na região central temos de aumentar igualmente a amplitude da variação em z. Assim, temos:

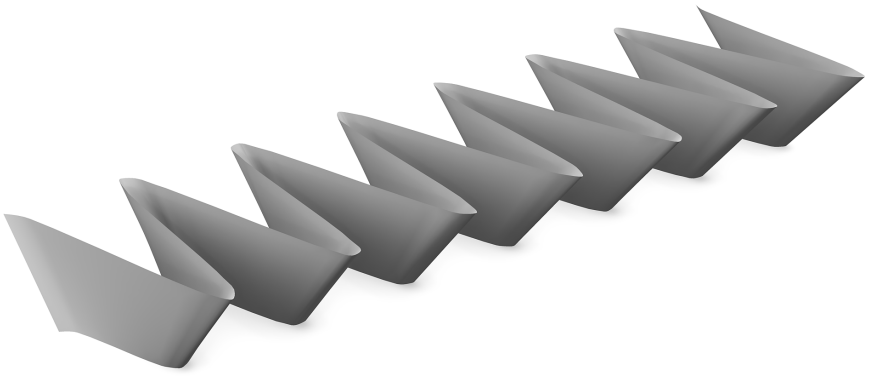


Figura 9.34: Terceira aproximação à modelação da Adega Ysios.

```
surface_grid(
  map_division(
    (x, z) -> xyz(x,
                  -pi/2 <= x <= pi/2 ?
                    (z + 3)*0.8*cos(x) :
                    cos(x),
                  -pi/2 <= x <= pi/2 ?
                    z*(1 + 0.4*cos(x)) :
                    z*(1 + 0.2*cos(x))),
                  -7*pi, 7*pi, 100,
                  0, 5, 10))
```

O resultado desta alteração está representado na Figura 9.35.

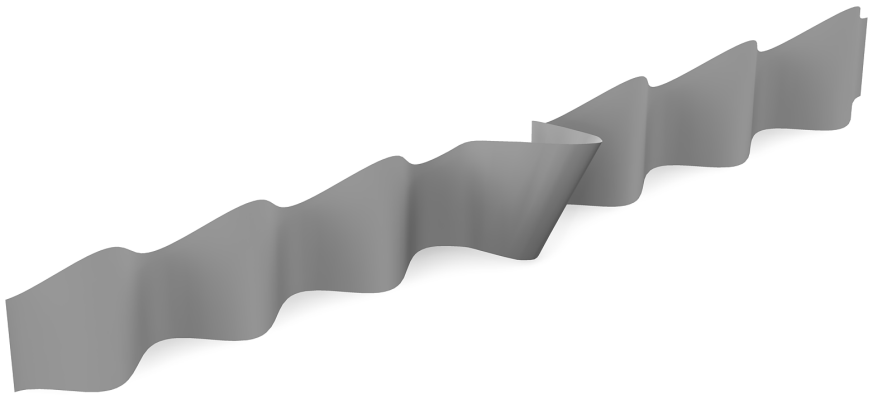
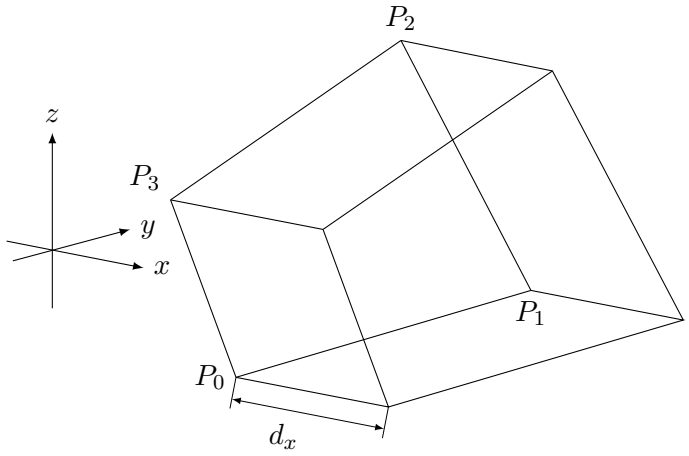


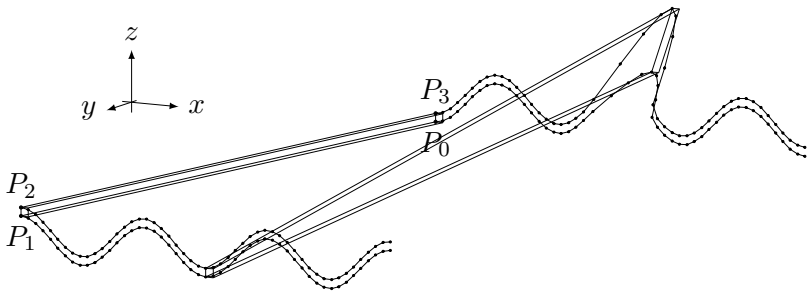
Figura 9.35: Quarta aproximação à modelação da Adega Ysios.

O formato final da função já dá uma boa aproximação à forma da adega. Esta função pode ser vista como a representação matemática das ideias de Calatrava para a parede frontal da Adega Ysios.

Exercício 9.9.1 Defina a função `prisma_poligonal` que, a partir de uma lista de pontos coplanares e de um deslocamento d_x paralelo ao eixo X, cria o prisma poligonal apresentado no esquema seguinte e que foi gerado por `prisma_poligonal([`P_0`,` P_1`,` P_2`,` P_3`],` d_x`)`

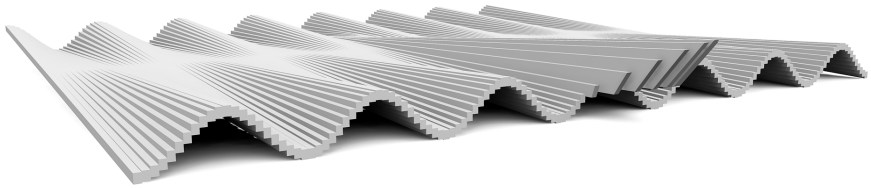


Exercício 9.9.2 Defina uma função `cobertura_ysios` capaz de criar coberturas com o "estilo" da Adega Ysios, mas em que a forma da cobertura é relativamente arbitrária. Para permitir essa arbitrariedade, assuma que a cobertura é definida por apenas quatro listas com igual número de pontos (representados pelas suas coordenadas) e em que os pontos que estão na mesma posição nas quatro listas estão num mesmo plano paralelo ao plano YZ, tal como se esquematiza na imagem seguinte:

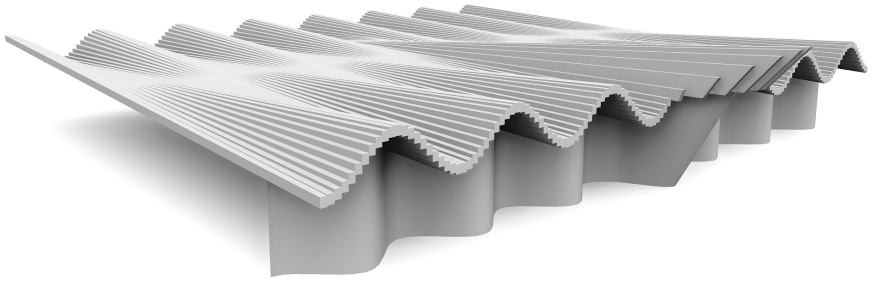


Usando um ponto de cada uma das quatro listas, empregue a função `prisma_poligonal` (explicada no exercício anterior) para criar um prisma cuja espessura d_x é igual à distância entre os planos definidos por esses pontos e pelos quatro pontos seguintes. A imagem anterior mostra dois dos prismas construídos por este processo.

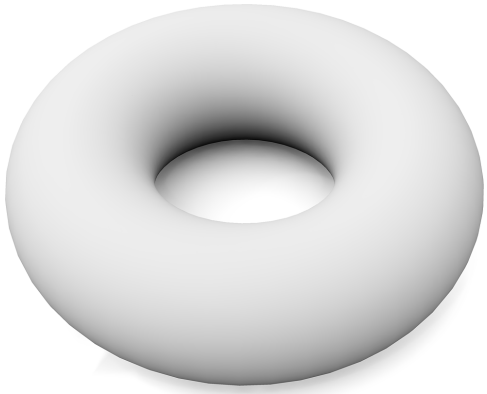
Exercício 9.9.3 Defina a função `curva_ysios` que, convenientemente parametrizada, é capaz de gerar uma lista com um dos vértices dos sucessivos prismas que constituem uma cobertura no estilo da Adega Ysios. A seguinte imagem foi gerada pela função `cobertura_ysios` usando quatro invocações da função `curva_ysios` como argumentos.



Exercício 9.9.4 Identifique uma combinação de parâmetros para as funções `paredes_ysios` e `curva_ysios` que seja capaz de reproduzir aproximadamente a fachada e a cobertura da adega Ysios, tal como se apresenta na seguinte imagem.

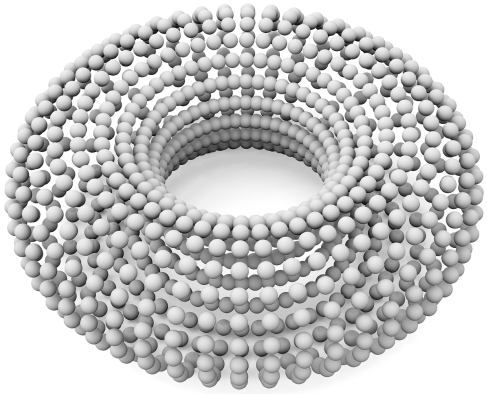


Exercício 9.9.5 A seguinte imagem descreve um toro:



Deduza a equação paramétrica do toro e defina a função Julia `toro` que cria um toro idêntico ao da imagem anterior a partir do centro do toro P, dos raios maior r_0 e menor r_1 e do número de intervalos m e n a considerar em cada dimensão.

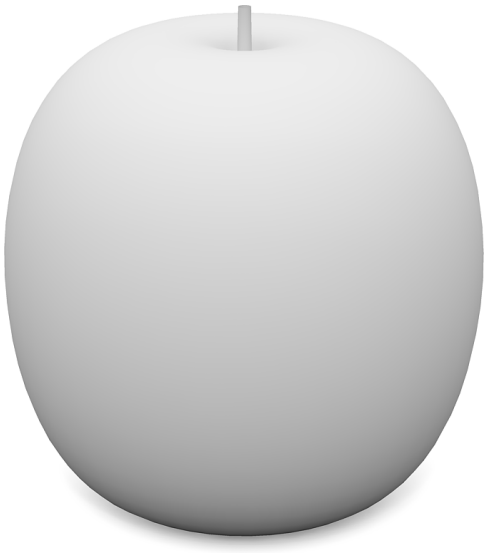
Exercício 9.9.6 Redefina a função anterior para gerar um toro de esferas semelhante ao apresentado na seguinte imagem:



Exercício 9.9.7 A "maçã" apresentada abaixo pode ser descrita pela equação paramétrica

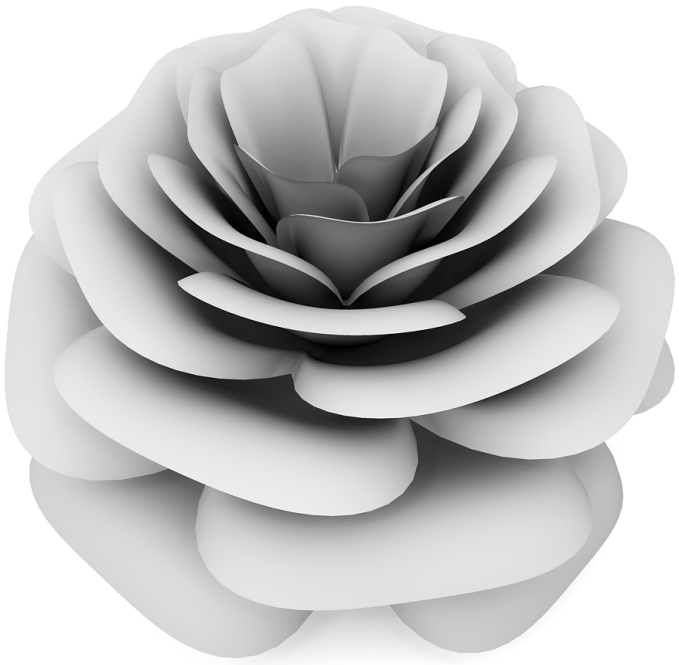
$$
\begin{aligned}
x &= \cos u \cdot (4 + 3.8 \cos v) \\
y &= \sin u \cdot (4 + 3.8 \cos v) \\
z &= (\cos v + \sin v - 1) \cdot (1 + \sin v) \cdot \log(1 - \pi \cdot \frac{v}{10}) + 7.5 \sin v
\end{aligned}
$$

$$
0 \leq u \leq 2\pi; \quad -\pi \leq v \leq \pi;
$$



Escreva uma expressão Julia que reproduz a "maçã" anterior.

Exercício 9.9.8 Considere a "rosa" que se apresenta em seguida:



Esta "rosa" foi gerada a partir de uma equação paramétrica descoberta por Paul Nylander. Pesquise por essa equação e escreva uma expressão Julia capaz de gerar a superfície correspondente.

9.10 Normais a uma Superfície

Quando uma superfície não é plana, mas é *quase*-plana, pode ser difícil a um observador perceber a sua curvatura. Nestes casos, é muito útil dispor de um mecanismo que permita visualizar os efeitos dessa curvatura sem com isso alterar a forma da superfície. A superimposição de vectores normais é um processo simples de fazer essa visualização, tal como se pode constatar nas imagens apresentadas na Figura 9.36. À esquerda, vemos uma superfície onde a curvatura não é perfeitamente óbvia. À direita, vemos a mesma superfície mas com os vectores normais a mostrarem claramente a curvatura.

Para sobrepormos os vectores normais temos de calcular a sua posição e direcção. Admitindo que a superfície está discretizada numa malha (tal como é produzido, por exemplo, pela função `map_division`), essa malha é composta por uma sucessão de quadrângulos que se estendem ao longo de duas direcções. Para vermos a curvatura, podemos posicionar cada vector no centro de cada um destes quadrângulos, segundo a direcção normal a esse quadrângulo.

Para calcularmos o centro de um quadrângulo podemos empregar a abordagem apresentada na Figura 9.37. O centro do quadrângulo definido

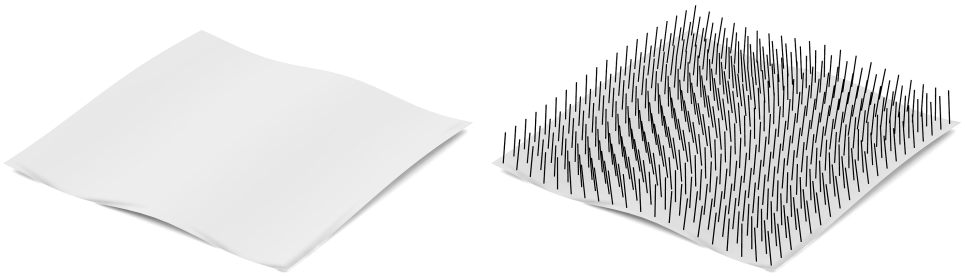


Figura 9.36: A imagem à esquerda mostra uma superfície quase plana. A imagem à direita mostra a mesma superfície com vectores normais sobrepostos, permitindo perceber melhor a sua curvatura.

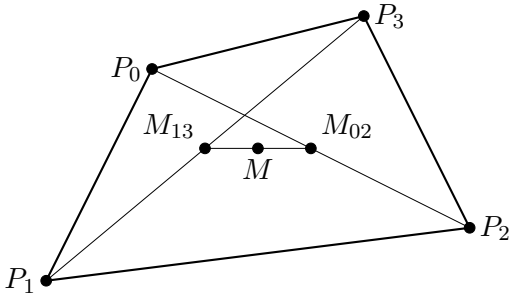


Figura 9.37: O centro de um quadrângulo como sendo o ponto médio dos pontos médios das diagonais.

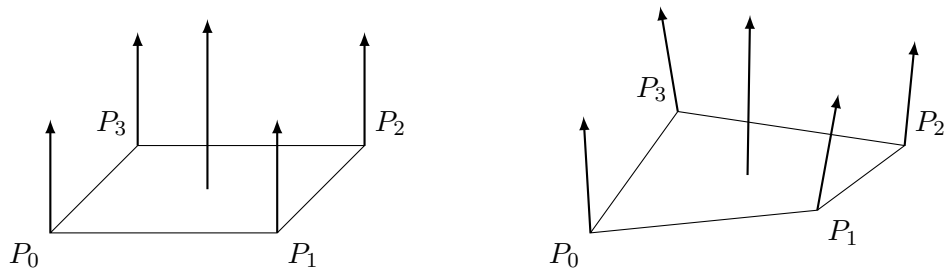


Figura 9.38: Normal no centro de um quadrângulo (planar à esquerda e não-planar à direita).

pelos pontos P_0, P_1, P_2, e P_3 pode ser obtido determinando os pontos médios M_{02} e M_{13} das diagonais do quadrângulo e, finalmente, o ponto médio M deste segmento. Traduzindo isto para Julia temos simplesmente:

```
media_pontos(p0, p1) =
  xyz((p0.x + p1.x)/2,
      (p0.y + p1.y)/2,
      (p0.z + p1.z)/2)

centro_quadrangulo(p0, p1, p2, p3) =
  media_pontos(
    media_pontos(p0, p2),
    media_pontos(p1, p3))
```

O cálculo da normal de um quadrângulo é simples quando o quadrângulo é planar, pois basta realizar o produto externo de duas arestas que se juntam num mesmo vértice, tal como podemos ver à direita da Figura 9.38. No entanto, no caso de um quadrângulo não planar (à direita da Figura 9.38), já essa lógica não é aplicável, pois poderão existir normais diferentes em cada vértice. É contudo possível computar uma normal aproximada usando o método de Newell [**?**] que se baseia no facto de as áreas das projecções de um polígono nos planos Cartesianos serem proporcionais às componentes do vector normal desse polígono. O método de Newell calcula essas áreas e, a partir delas, deriva o vector normal $\vec{N}$. Matematicamente falando, dado um polígono com n vértices $\{v_0, v_1, \ldots, v_n\}$, o método consiste em calcular as componentes do vector normal $\vec{N}$ através do somatório de áreas:

$$N_x = \sum_{i=0}^{n} (v_{i_y} - v_{i+1_y}) \cdot (v_{i_z} + v_{i+1_z})$$

$$N_y = \sum_{i=0}^{n} (v_{i_z} - v_{i+1_z}) \cdot (v_{i_x} + v_{i+1_x})$$

$$N_z = \sum_{i=0}^{n} (v_{i_x} - v_{i+1_x}) \cdot (v_{i_y} + v_{i+1_y})$$

Para transformarmos o vector resultante num vector unitário, basta *normalizá-lo*, i.e., dividir cada componente pelo seu comprimento. A função pré-definida `unitize` faz precisamente isso. Traduzindo para Julia, temos:

```
normal_poligono(vs) =
  unitized(produtos_cruzados([vs..., vs[1]]))

produtos_cruzados(vs) =
  if length(vs) == 1
    vxyz(0, 0, 0)
  else
    cross(vs[1], vs[2]) + produtos_cruzados(vs[2:end])
  end
```

No caso de um quadrângulo, podemos simplesmente definir

```
normal_quadrangulo(p0, p1, p2, p3) =
  normal_poligono([p1 - p0, p2 - p1, p3 - p2, p0 - p3])
```

Na verdade, algumas das operações atrás definidas encontram-se pré-definidas em Khepri, nomeadamente, o produto cruzado (denominado `cross`) e o vector normalizado (denominado `unitize`).

Para representar visualmente a normal vamos empregar um cilindro muito fino cuja dimensão será proporcional à dimensão do quadrângulo, de modo a que ele se adapte automaticamente a diferentes quadrângulos. Naturalmente, é também possível empregar outros esquemas, por exemplo, adoptando dimensões fixas.

```
normal(pt0, pt1, pt2, pt3) =
  let c = centro_quadrangulo(pt0, pt1, pt2, pt3),
      n = normal_quadrangulo(pt0, pt1, pt2, pt3),
      d = distance(pt0, pt2)
    with(current_layer, create_layer("White")) do
      cylinder(c, d/40, c + n*d*1.5)
    end
  end
```

O passo seguinte é percorrer todos os quadrângulos da malha e construir a normal em cada um. Para isso, fazemos:

```
normais(ptss) =
  [[normal(p0, p1, p2, p3)
    for (p0, p1, p2, p3)
    in zip(pts0[1:end-1], pts1[1:end-1], pts1[2:end], pts0[2:end])]
   for (pts0, pts1)
   in zip(ptss[1:end-1], ptss[2:end])]
```

Embora a visualização das normais seja complementar à visualização da superfície, não faz muito sentido ver a primeira sem também ver a segunda. Assim sendo, vamos definir uma função que combina as duas visualizações:

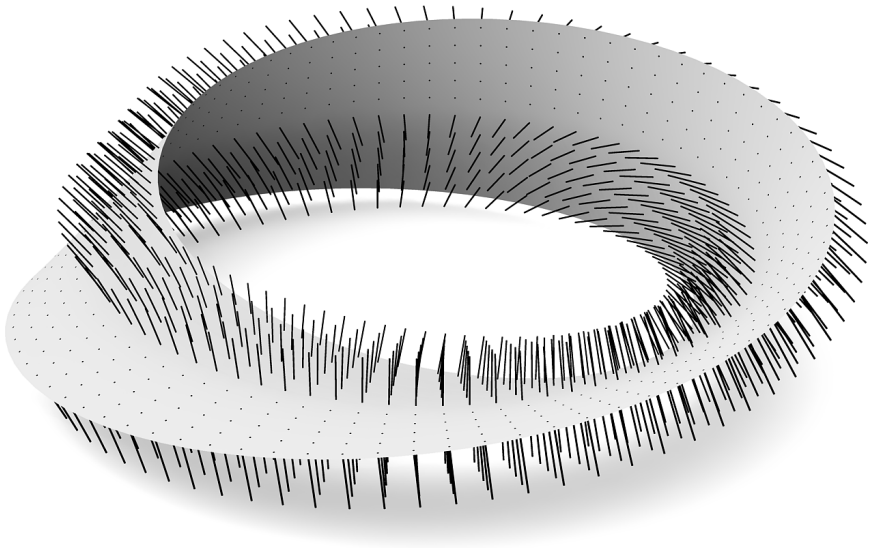


Figura 9.39: A superfície de Möbius com as normais sobrepostas.

```
superficie_e_normais(ptss) =
  begin
    normais(ptss)
    surface_grid(ptss)
  end
```

Para testarmos esta função podemos agora experimentar aplicá-la à geração das normais da faixa de Möbius:

```
superficie_e_normais(faixa_moebius(0, 4*pi, 160, 0, 0.3, 5))
```

A Figura 9.39 mostra o resultado.

9.11 Processamento de Superfícies

Uma observação cuidada da função `normais` mostra que ela itera ao longo dos quadrângulos da superfície, computando a normal em cada um. Naturalmente, podemos generalizar este processo de modo a que seja possível processar uma superfície através da aplicação de uma qualquer operação a todos os seus quadrângulos. A passagem para uma função de ordem superior é feita substituindo a função que calcula a normal por uma função arbitrária:

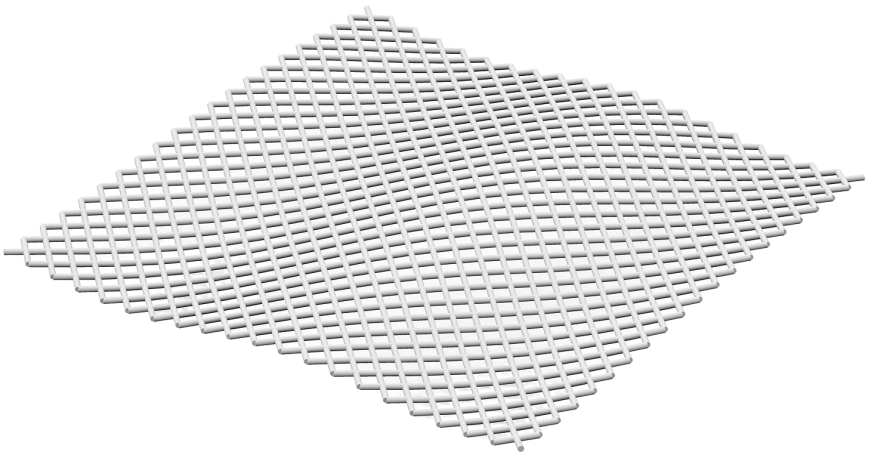


Figura 9.40: Uma rede feita de cilindros cruzados.

```
itera_quadrangulos(f, ptss) =
  [[f(p0, p1, p2, p3)
    for (p0, p1, p2, p3)
    in zip(pts0[1:end-1], pts1[1:end-1], pts1[2:end], pts0[2:end])]
   for (pts0, pts1)
   in zip(ptss[1:end-1], ptss[2:end])]
```

A partir de agora, a computação das normais a uma superfície pode ser tratada como uma mera particularização da função `itera_quadrangulos`:

```
normais(ptss) =
  itera_quadrangulos(normal, ptss)
```

A grande vantagem, contudo, está no facto de podermos agora criar muitas outras formas de processar uma superfície. A título de exemplo, consideremos a criação de uma rede, tal como se apresenta na Figura 9.40. Para este exemplo, a forma da rede é definida pela superfície paramétrica

$$\begin{cases} x(u,v) = u \\ y(u,v) = v \\ z(u,v) = \dfrac{7}{100}(\sin(u) + \sin(2(v-2))) \end{cases}$$

$$0 \leq u \leq 5$$

$$0 \leq v \leq 5$$

A criação da rede anterior pode ser feita simplesmente criando uma cruz ao longo dos quadrângulos de uma superfície. A cruz pode ser feita usando dois cilindros cujas extremidades coincidem com os vértices das diagonais do quadrângulo. Para que os cilindros se adaptem a diferentes

quadrângulos, podemos considerar que o raio dos cilindros é proporcional ao tamanho do quadrângulo. A seguinte função implementa esta abordagem:

```
cruz_cilindros_quad(p0, p1, p2, p3) =
  let r = min(distance(p0, p1), distance(p0, p3))/10
    cylinder(p0, r, p2)
    cylinder(p1, r, p3)
  end
```

Para criarmos a rede da Figura 9.40 basta-nos agora fazer:

```
itera_quadrangulos(
  cruz_cilindros_quad,
  map_division((i, j) -> xyz(i, j, 0.07*(sin(i) + sin(2*(j - 2)))),
               0, 5, 20, 0, 5, 20))
```

A função `itera_quadrangulos` pode assim ser usada quer para diferentes superfícies, quer para diferentes funções a iterar sobre as coordenadas dessas superfícies. A Figura 9.41 mostra diferentes construções realizadas sobre a superfície definida pela seguinte função paramétrica:

$$\begin{cases} x(u,v) = u \\ y(u,v) = v \\ z(u,v) = \dfrac{4}{10}(\sin(u+v) + e^{\frac{|u-1|}{10}} + \sin(v-1)) \end{cases}$$

$$0 \leq u \leq 3$$

$$0 \leq v \leq 6$$

Na Figura 9.42 mostramos o resultado das mesmíssimas funções mas agora iteradas sobre uma outra superfície definida por:

$$\begin{cases} x(u,v) = u \\ y(u,v) = v \\ z(u,v) = \dfrac{4}{10}\sin(u \cdot v) \end{cases}$$

$$-\pi \leq u \leq \pi$$

$$-\pi \leq v \leq \pi$$

Exercício 9.11.1 Para cada um dos exemplos apresentados na Figura 9.41 (ou, equivalentemente, na Figura 9.42), defina a função que recebe quatro pontos como argumento e que, ao ser iterada pela função `itera_quadrangulos` ao longo das coordenadas da superfície, reproduz o modelo.

Exercício 9.11.2 Defina uma função que, dadas as coordenadas dos quatro vértices de um quadrângulo, cria dois cilindros entrelaçados ao longo das diagonais do quadrângulo, tal como se apresenta na seguinte imagem:

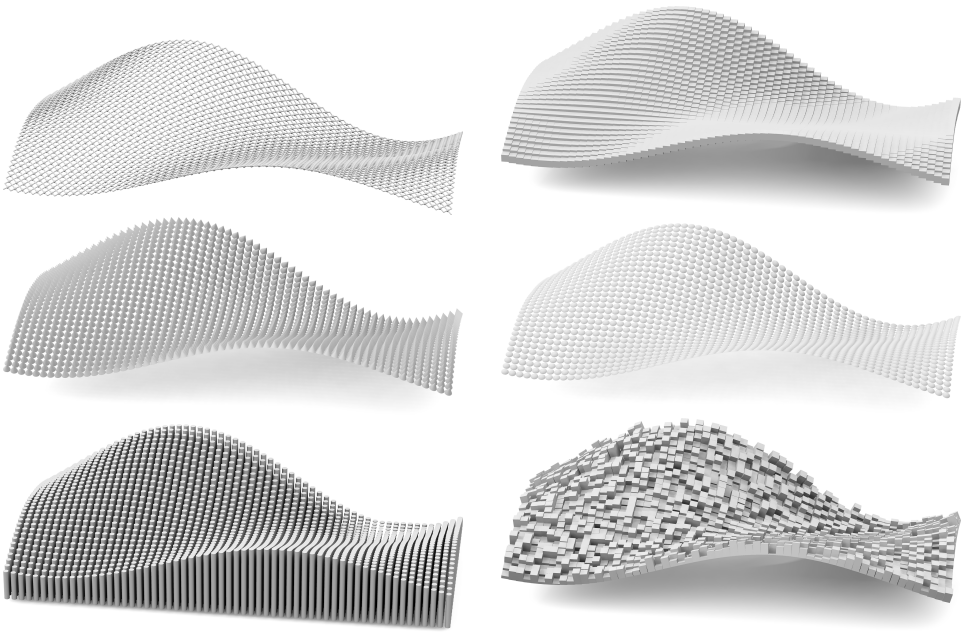


Figura 9.41: Diferentes construções feitas a partir de uma superfície. Da esquerda para a direita e, de cima para baixo, o elemento que se repete é: (1) quatro cilindros unidos a uma esfera; (2) um cubo; (3) um cone; (4) uma calota esférica; (5) um cilindro; (6) um paralelipídedo de altura aleatória.

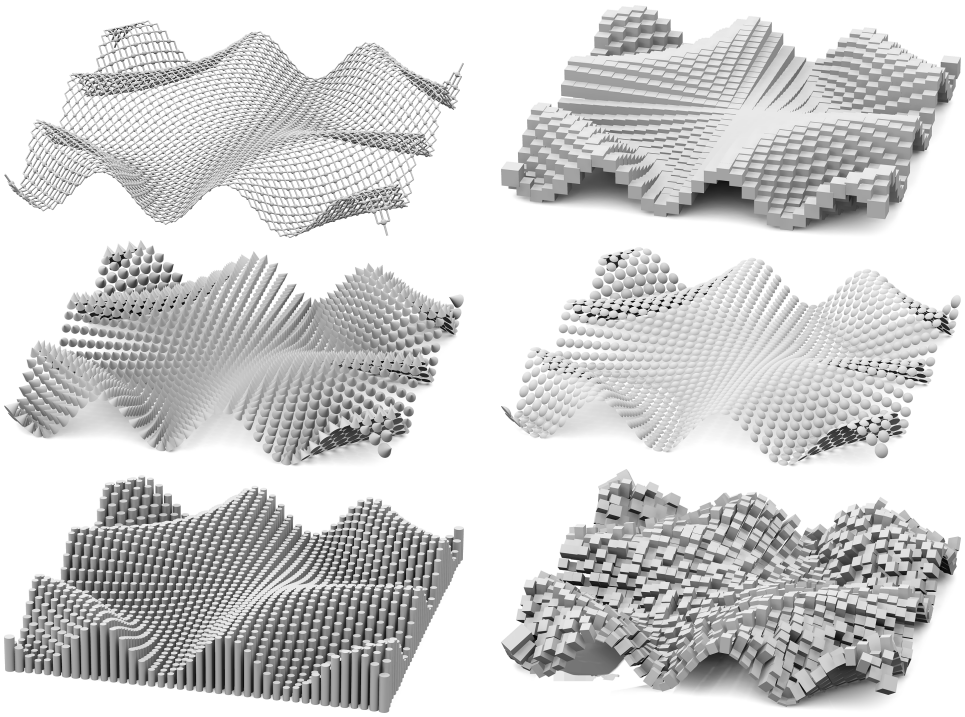
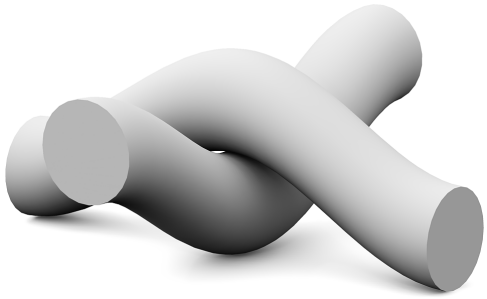
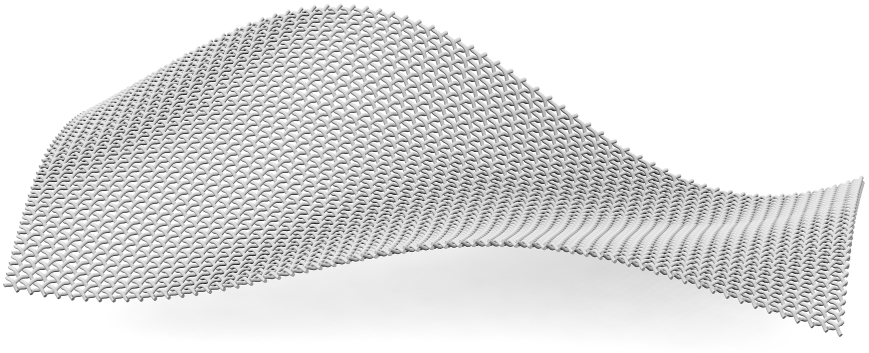


Figura 9.42: Diferentes construções feitas a partir de uma superfície.

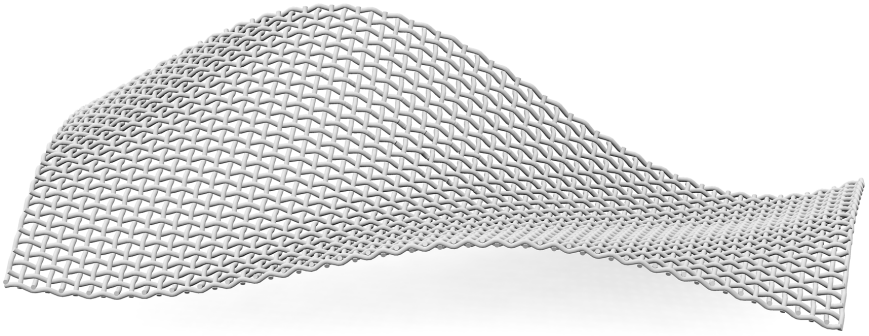


Use a função que criou como argumento de `itera_quadrangulos` para gerar o "tecido" que se apresenta em seguida:



Exercício 9.11.3 A abordagem empregue no exercício anterior não é adequada quando a superfície em questão não é quase-plana pois os cilindros correspondentes a dois quadrângulos adjacentes podem não ficar correctamente alinhados, caso em que apresentarão sobreposições ou, pior, espaços vazios.

Redefina o processo de criação do tecido de modo a que sejam empregues cilindros entrelaçados, mas em que cada cilindro corresponde a um fio completo (e não apenas ao troço contido num quadrângulo). Para simplificar, pode assumir que os fios se desenvolvem ao longos das linhas e colunas da matriz de coordenadas da superfície, de forma a gerar imagens como a que se segue:



Vimos nos exemplos anteriores uma separação entre a geração das coordenadas de uma superfície (usando, por exemplo, a função `map_division`) e o uso dessas coordenadas, por exemplo, para visualização (usando a função `surface_grid`) ou para processamento (usando a função `itera_quadrangulos`).

Esta separação é vantajosa porque nos permite combinar arbitrariamente diferentes funções de geração de coordenadas com diferentes funções que usam essas coordenadas. Uma outra possibilidade interessante é a das funções que processam as coordenadas para produzirem novas coordenadas. Um exemplo trivial seria a aplicação de um efeito de escala sobre uma superfície, simplesmente multiplicando a escala por cada uma das coordenadas, gerando assim um novo conjunto de coordenadas.

Um exemplo um pouco menos trivial, mas bastante mais interessante, é o da criação de treliças cuja forma acompanha uma superfície. Como vimos na secção 5.6, estas treliças são compostas por esferas unidas entre si por barras, formando pirâmides quadrângulares. Para construirmos uma treliça que acompanhe uma superfície podemos simplesmente construir cada uma das pirâmides da treliça com a base assente na superfície e com o topo localizado no vector normal a essa superfície. A função `trelica_espacial` (que desenvolvemos na secção 5.6.3) é perfeitamente capaz de construir estas treliças, desde que receba as posições dos nós da treliça sob a forma de uma lista com um número ímpar de listas de coordenadas, segundo a sequência nós-da-base, nós-do-topo, nós-da-base, nós-do-topo, ..., nós-da-base.

Tendo estes pressupostos em conta, para construirmos uma treliça que acompanha uma superfície podemos começar por gerar as coordenadas dessa superfície, mas convenientemente espaçadas, de modo a servirem de posições aos nós da base de cada pirâmide da treliça. De seguida, para cada quadrângulo de coordenadas, temos de localizar o nó do topo da pirâmide cuja base é esse quadrângulo. Para isso, podemos admitir que as treliças empregarão, tanto quanto possível, barras com a mesma dimensão l. Isto implica que o nó do topo de cada pirâmide estará localizado na normal ao quadrângulo definido pelos nós da base, a uma distância do centro desse quadrângulo dada por $h = \frac{l}{\sqrt{2}}$. Assim sendo, temos de calcular quer o centro, quer a normal do quadrângulo. A seguinte função ilustra esse processo:

```
vertice_piramide_quadrangular(p0, p1, p2, p3) =
  let h = (distance(p0, p1) +
           distance(p1, p2) +
           distance(p2, p3) +
           distance(p3, p0))/4.0/sqrt(2)
    centro_quadrangulo(p0, p1, p2, p3) +
    normal_quadrangulo(p0, p1, p2, p3)*h
  end
```

O passo seguinte será usar esta função para computar as coordenadas dos vértices ao longo da malha de coordenadas da superfície, criando novas sequências de coordenadas que inserimos entre as sequências de coordenadas da malha original, de modo a reproduzir a alternância entre nós-da-base e nós-do-topo da treliça.

As seguintes duas funções realizam essa tarefa:

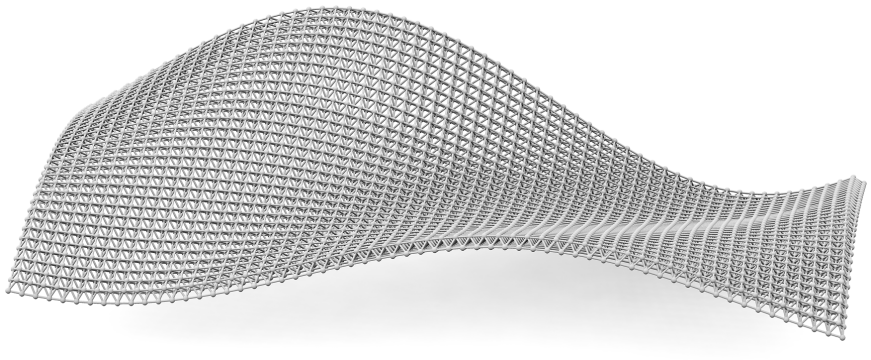


Figura 9.43: Treliça construída sobre uma superfície.

```
insere_vertice_piramide(ptss) =
  if length(ptss) == 1
    ptss
  else
    [ptss[1],
     insere_vertice_piramide_2(ptss[1], ptss[2]),
     insere_vertice_piramide(ptss[2:end])...]
  end

insere_vertice_piramide_2(pts0, pts1) =
  [vertice_piramide_quadrangular(pts0[1], pts1[1], pts1[2], pts0[2]),
   (length(pts0) == 2 ?
     [] :
     insere_vertice_piramide_2(pts0[2:end], pts1[2:end]))...]
```

Finalmente, dada uma malha qualquer, apenas temos de encadear a criação dos vértices com a criação da treliça, i.e.:

```
trelica_espacial(insere_vertice_piramide(malha))
```

A Figura 9.43 mostra o resultado da criação de uma treliça sobre a mesma superfície usada na Figura 9.41. Naturalmente, podemos combinar a construção de treliças com qualquer outra superfície. A Figura 9.44 mostra uma treliça construída sobre a mesma superfície usada na Figura 9.42.

Finalmente, a título de curiosidade, a Figura 9.45 mostra uma treliça construída sobre uma faixa de Möbius. Esta treliça foi produzida pela avaliação da seguinte expressão:

```
trelica_espacial(
    insere_vertice_piramide(
        faixa_moebius(0, 4*pi, 160, 0, 0.3, 5)))
```

Exercício 9.11.4 Se observar atentamente a treliça irá notar um "defeito." Identifique-o e explique-o.

Exercício 9.11.5 Considere o "ouriço" e o "cacto" apresentados na imagem seguinte:

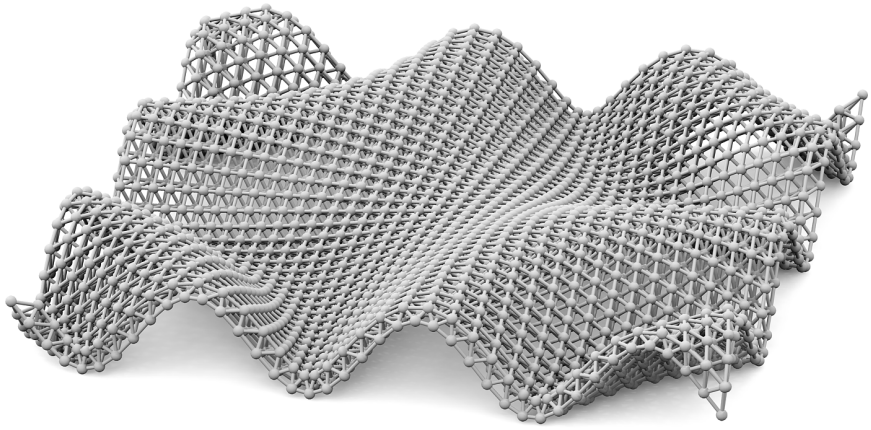


Figura 9.44: Treliça construída sobre uma superfície.

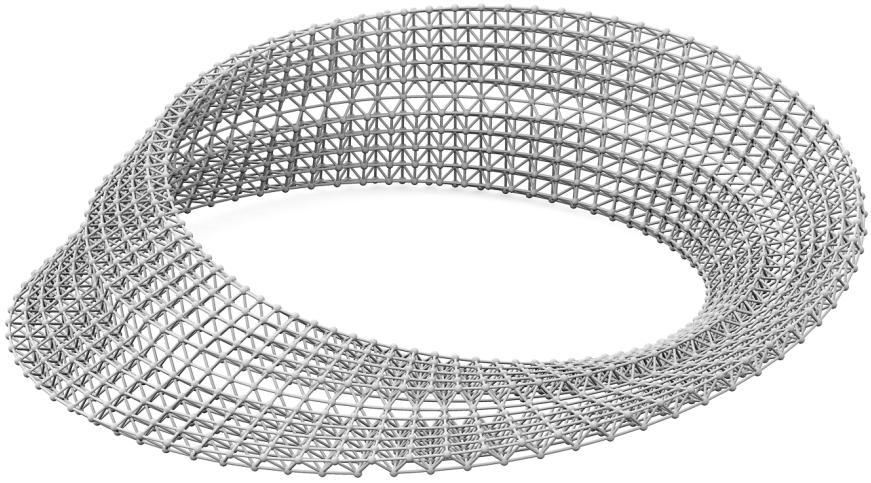
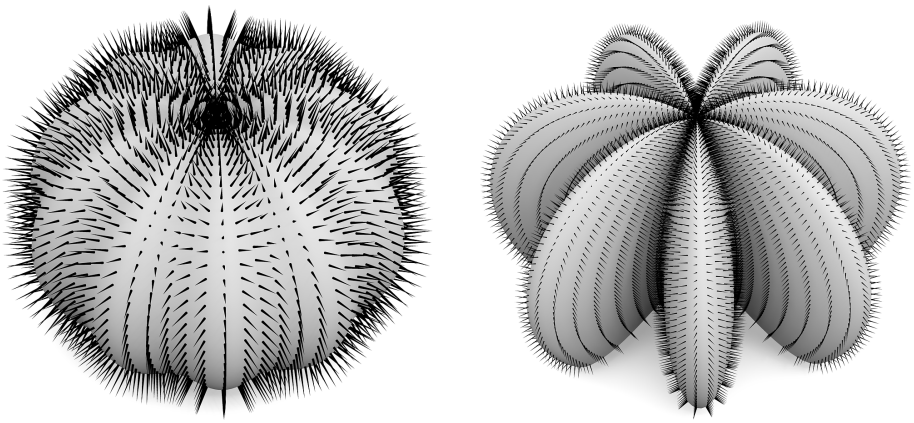


Figura 9.45: Uma treliça com a forma de uma faixa de Möbius.



Defina uma função que, convenientemente parametrizada, seja capaz de construir ouriços e cactos como os anteriores.

Epílogo

Vimos, ao longo desta obra, como a programação se pode tornar numa importante ferramenta para o arquitecto. A programação permite-nos formalizar e transmitir um pensamento, permite-nos modelar rigorosamente formas que só existem na nossa imaginação, permite-nos automatizar tarefas monótonas e repetitivas, permite-nos, inclusive, gerar formas em que nem sequer tínhamos pensado. Todas estas capacidades justificam que se dê à programação a atenção que ela merece.

É precisamente essa atenção que os grandes ateliers de arquitectura têm vindo a dar no passado recente e, na verdade, o domínio da programação começa a ser um requisito frequente para a admissão.

Aprender a programar é, portanto, um imperativo. À semelhança de muitas outras disciplinas, é uma aprendizagem que exige esforço, dedicação, rigor, e atenção ao pormenor. Mas é também uma aprendizagem com um enorme retorno do investimento realizado. Aqueles que dominam a programação tornam-se mais capazes e mais eficientes.

Optámos, nesta obra, por ensinar a programar no contexto da Arquitectura. Para isso, seleccionámos os tópicos que tivessem uma aplicabilidade imediata à Arquitectura e, na verdade, muitos dos exemplos que usámos foram, na realidade, fornecidos por arquitectos.

Apesar da dimensão desta obra, convém termos presente que ela ilustra apenas uma fracção infinitesimal do que podemos considerar como programação. Inúmeras outras obras terão de ser (continuamente) escritas para que essa fracção comece a ter significado. A bibliografia recomendada enumera algumas fontes adicionais de informação para os leitores interessados em aprofundar os conhecimentos.

Bibliografia

[1] The revised revised report on scheme, or an uncommon lisp. MIT AI Memo 848, Massachusetts Institute of Technology, Cambridge, Mass., August 1985.

[2] Revised^3 report on the algorithmic language scheme. *ACM Sigplan Notices*, 21(12), December 1986.

[3] IEEE Std 1178-1990. *Ieee Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[4] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.

[5] Carlos Roberto Barrios Hernandez. Thinking parametric design: introducing parametric gaudi. *Design Studies*, 27(3):309–324, 2006.

[6] Jane Burry and Mark Burry. *The new mathematics of architecture*. Thames & Hudson London, 2010.

[7] Mark Burry. *Scripting cultures: Architectural design and programming*. John Wiley & Sons, 2013.

[8] Cristiano Ceccato, Lars Hesselgren, and Mark Pauly. *Advances in Architectural Geometry 2010*. Springer, 2010.

[9] John Clements, Paul T. Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Little languages and their programming environments, August 22 2001.

[10] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[11] Michael Eisenberg. *Programming in Scheme*. The Scientific Press, Redwood City, CA, 1988.

[12] Michael Eisenberg, William Clinger, and Anne Hartheimer. *Programming in Macscheme*. The Scientific Press, Redwood City, CA, 1990.

[13] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The DrScheme project: an overview. *ACM SIGPLAN Notices*, 33(6):17–23, June 1998.

[14] Matthias Felleisen. Why teach programming languages in this day and age and how to go about it. *ACM SIGPLAN Notices*, 43(11):59–61, November 2008.

[15] Matthias Felleisen. TeachScheme!: a checkpoint. *ACM SIGPLAN Notices*, 45(9):129–130, September 2010.

[16] Matthias Felleisen. Multilingual component programming in racket. In Ewen Denney and Ulrik Pagh Schultz, editors, *Generative Programming And Component Engineering, Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE 2011, Portland, Oregon, USA, October 22-24, 2011*, pages 1–2. ACM, 2011.

[17] Matthias Felleisen. Teachscheme! In Thomas J. Cortina, Ellen Lowenfeld Walker, Laurie A. Smith King, and David R. Musicant, editors, *Proceedings of the 42nd ACM technical symposium on Computer science education, SIGCSE 2011, Dallas, TX, USA, March 9-12, 2011*, pages 1–2. ACM, 2011.

[18] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shiram Krishnamurthi. *How to Design Programs*. The MIT Press, 2003.

[19] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: a programming environment for scheme. *J. Funct. Program*, 12(2):159–182, 2002.

[20] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Drscheme: A pedagogic programming environment for scheme, June 11 1997.

[21] Gary William Flake. *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptations*. The MIT Press, 1998.

[22] Daniel P. Friedman and Matthias Felleisen. *The Little LISPer: Second Edition*. Science Research Associates, Inc., Palo Alto, California, 1986.

[23] Daniel P. Friedman and Matthias Felleisen. *The Little LISPer*. MIT Press, 1987.

[24] Daniel P. Friedman and Matthias Felleisen. *The Seasoned Schemer*. MIT Press, 1996.

[25] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1992.

[26] Michael Hensel, Achim Menges, and Michael Weinstock. *Emergence: morphogenetic design strategies*. Wiley-Academy Chichester, 2004.

[27] Dominik Holzer, Richard Hough, and Mark Burry. Parametric design and structural optimisation for early design exploration. *International Journal of Architectural Computing*, 5(4):625–643, 2007.

[28] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40–, January 1965.

[29] Yehuda E Kalay. *Architecture's new media: principles, theories, and methods of computer-aided design*. The MIT press, 2004.

[30] Branko Kolarevic. Designing and manufacturing architecture in the digital age. *Architectural information management*, 2001.

[31] Branko Kolarevic. *Architecture in the digital age: design and manufacturing*. Taylor & Francis, 2003.

[32] Branko Kolarevic and Kevin R Klinger. *Manufacturing material effects: rethinking design and making in architecture*. Routledge New York, 2008.

[33] Shriram Krishnamurthi and Matthias Felleisen. Lecture notes on the principles of programming languages, August 28 2001.

[34] David Littlefield. *Space Craft: developments in architectural computing*. RIBA Enterprises, 2008.

[35] Michael Meredith and Mutsuro Sasaki. *From control to design: parametric/algorithmic architecture*. Actar-D, 2008.

[36] Farshid Moussavi, Michael Kubo, and Seth Hoffman. *The function of ornament*. Actar, 2006.

[37] Farshid Moussavi and Daniel López. *The function of form*. Actar Barcelona, 2009.

[38] Helmut Pottmann. *Architectural geometry*, volume 10. Bentley Institute Press, 2007.

[39] Helmut Pottmann. Architectural geometry as design knowledge. *Architectural Design*, 80(4):72–77, 2010.

[40] Helmut Pottmann, Michael Hofer, and Axel Kilian. Advances in architectural geometry. In *Proc. of Vienna conference*, 2008.

[41] Casey Reas. *Form+code in design, art, and architecture*. Princeton Architectural Press, 2010.

[42] Casey Reas and Ben Fry. Processing: programming for the media arts. *AI & SOCIETY*, 20(4):526–538, 2006.

[43] Casey Reas and Ben Fry. *Processing: a programming handbook for visual designers and artists*, volume 6812. Mit Press, 2007.

[44] Casey Reas and Ben Fry. *Getting Started with Processing*. O'Reilly, 2010.

[45] Casey Reas and Benjamin Fry. Processing: a learning environment for creating interactive web graphics. In *ACM SIGGRAPH 2003 Web Graphics*, pages 1–1. ACM, 2003.

[46] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual, Fourth Edition*. Yale University Computer Science Department, January 1984.

[47] Stephen Slade. *The T Programming Language*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1987.

[48] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming*. MIT Press and McGraw-Hill, 1989.

[49] Lars Spuybroek. *NOX: machining architecture*. Thames & Hudson, 2004.

[50] Lars Spuybroek. *Research & design: the architecture of variation*. Thames & Hudson, 2009.

[51] Ivan E Sutherland, Robert F Sproull, and Robert A Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys (CSUR)*, 6(1):1–55, 1974.

[52] Helmut Vogel. A better way to construct the sunflower head. *Mathematical biosciences*, 44(3-4):179–189, 1979.

[53] Robert Woodbury et al. *Elements of parametric design*. Taylor & Francis Group, 2010.

Soluções

Solução do Exercício 1.1.1

$$b^n = \begin{cases} 1, & \text{se } n = 0 \\ b \times b^{(n-1)}, & \text{se } n \in \mathbb{N}. \end{cases}$$

Solução do Exercício 1.1.2 Um programa é o conhecimento que se transmite a um computador e que serve para dar ao computador a capacidade para resolver um determinado problema.

Solução do Exercício 1.1.3 Uma linguagem de programação é uma linguagem onde não há margem para qualquer ambiguidade, lacuna ou imprecisão. Isso permite que ela seja usada para descrever rigorosamente os processos que, se forem levados a cabo, produzem os resultados desejados. Por serem rigorosas, as linguagens de programação são compreensíveis por computadores que poderão executar os processos nelas descritos.

Solução do Exercício 1.3.1 O REPL é o *read-eval-print-loop*, um ciclo que o Julia realiza continuamente em que lê uma expressão, avalia-a e escreve o resultado, se existir.

Solução do Exercício 1.3.2

1. $\frac{1}{2} \times 3$
2. $\frac{1}{2-3}$
3. $\frac{1+2}{3}$
4. $\frac{\frac{1}{2}}{3}$
5. $\frac{1}{\frac{2}{3}}$

Solução do Exercício 1.3.3

1. 1.5
2. -1
3. 1.0
4. -4

Solução do Exercício 1.4.1

```
dobro(x) =
  2*x
```

Solução do Exercício 1.4.2

1. `volume_esfera`
2. `primo`
3. `centimetros_para_polegadas`

Solução do Exercício 1.4.3

```
radianos(graus) =
  3.14159*(graus/180.0)
```

Solução do Exercício 1.4.4

```
graus(radianos) =
  180.0*(radianos/3.14159)
```

Solução do Exercício 1.4.5

```
perimetro_circunferencia(raio) =
  2*3.14159*raio
```

Solução do Exercício 1.4.6

```
volume_paralelipedo(comprimento, altura, largura) =
  comprimento*altura*largura
```

Solução do Exercício 1.4.7

```
volume_cilindro(raio, comprimento) =
  area_circulo(raio)*comprimento
```

Solução do Exercício 1.4.8

```
media(x, y) =
  (x + y)/2.0
```

Solução do Exercício 1.5.1

1. `sqrt(1/log(2^abs(3 - 9*log(25))))`
2. `cos(2/sqrt(5))^4/atan(3)`
3. `1/2 + sqrt(3) + sin(2)^(5/2)`

Solução do Exercício 1.5.2

1. $\log \sin(2^4 + \frac{\lfloor \operatorname{atan} \pi \rfloor}{\sqrt{5}})$
2. $\cos^5 \cos \cos 0.5$
3. $\sin \frac{\cos \frac{\sin \frac{\pi}{3}}{3}}{3}$

Solução do Exercício 1.5.3

```
impar(x) =
  x % 2 == 1
```

Solução do Exercício 1.5.4

```
area_pentagono_regular(r) =
  5/8*r^2*sqrt(10 + 2*sqrt(5))
```

Solução do Exercício 1.5.5

```
volume_elipsoide(a, b, c) =
  4/3*pi*a*b*c
```

Solução do Exercício 1.6.1

```
f(x) =
  x - 0.1*(10*x - 10)
```

Solução do Exercício 1.6.2

```
julia> f(5.1)
0.9999999999999991
julia> f(51367.7)
0.999999999992724
julia> f(176498634.7)
0.9999999701976776
julia> f(1209983553611.9)
0.999755859375
julia> f(19843566622234755.9)
0.0
julia> f(553774558711019983333.9)
-65536.0
```

Como o tratamento dos números inexactos empregue em Julia não corresponde ao comportamento matematico usual, surgem erros de arredondamento que explicam os resultados obtidos.

Solução do Exercício 1.6.3 Se um lanço de escada de altura a tem n espelhos então cada espelho tem uma altura de $\frac{a}{n}$. Neste caso, de acordo com a proporção, a largura de cada cobertor será $0.64 - 2\frac{a}{n}$. Uma vez que a n espelhos estão associados $n - 1$ cobertores, o comprimento total do lanço de escadas será, então, $(n - 1)(0.64 - 2\frac{a}{n})$.

```
comprimento_lanco_escadas(altura_lanco, n_espelhos) =
  (n_espelhos - 1)*(0.64 - 2*(altura_lanco/n_espelhos))
```

Solução do Exercício 1.10.1

1. `true`
2. `true`
3. `true`

Solução do Exercício 1.10.2

- Uma expressão condicional é uma expressão cujo valor depende de um ou mais testes, permitindo escolher vias diferentes para a obtenção do resultado.
- Uma expressão lógica é uma expressão cujo valor pode ser interpretado como verdade ou falso.

Solução do Exercício 1.10.3

- Um valor lógico é um valor que é interpretado como verdade ou falso.
- Os valores lógicos usados em Julia são o `true` e o `false`.

Solução do Exercício 1.10.4

- Um predicado é uma função que produz valores lógicos.
- Os operadores > e = são exemplos de predicados.

Solução do Exercício 1.10.5

- Um operador relacional é um operador que compara os seus operandos.
- Os operadores >, = e <= são exemplos de operadores relacionais.

Solução do Exercício 1.10.6

- Um operador lógico é um operador que combina valores lógicos.
- Os operadores `and`, `or` e `not` são exemplos de operadores lógicos.

Solução do Exercício 1.10.7

1. `x < y`
2. `x <= y`
3. `x < y < z`
4. `x < y && x < z`
5. `x <= y <= z`
6. `x <= y && y < z`
7. `x < y && y <= z`

Solução do Exercício 1.12.1

```
soma_maiores(x, y, z) =
  if x >= y
    if y >= z
      x + y
    else
      x + z
    end
  elseif x < z
    y + z
  else
    x + y
  end
```

Solução do Exercício 1.12.2

```
max3(x, y, z) =
  max(x, max(y, z))
```

Solução do Exercício 1.12.3

```
segundo_maior(x, y, z) =
  max3(min(x, y), min(y, z), min(z, x))
```

Solução do Exercício 1.15.1

```
module Hyperbolic
export sinh(x), cosh(x), tanh(x)

sinh(x) =
  (exp(x) - exp(-x))/2

cosh(x) =
  (exp(x) + exp(-x))/2

tanh(x) =
  (exp(x) - exp(-x))/(exp(x) + exp(-x))
```

Solução do Exercício 1.15.2

```
asinh(x) =
  log(x + sqrt(x*x + 1))

acosh(x) =
  log(x + sqrt(x*x - 1))

atanh(x) =
  if abs(x) < 1
    log((1 + x)/(1 - x))/2
  else
    log((x + 1)/(x - 1))/2
  end
```

Solução do Exercício 2.3.1

```
posicao_media(p0, p1) =
  xyz(media(cx(p0), cx(p1)), media(cy(p0), cy(p1)), media(cz(p0), cz(p1)))
```

Solução do Exercício 2.3.2

```
posicoes_iguais(p0, p1) =
  cx(p0) == cx(p1) && cy(p0) == cy(p1) && cz(p0) == cz(p1)
```

Solução do Exercício 2.3.3

```
vector_unitario(v) =
  let l = sqrt(v.x^2 + v.y^2 + v.z^2)
    vxyz(v.x/l, v.y/l, v.z/l)
  end
```

Solução do Exercício 2.3.4

```
vector_simetrico(v) =
  vxyz(-v.x, -v.y, -v.z)
```

Solução do Exercício 2.4.1

```
distancia_ponto_recta(p0, p1, p2) =
  abs((p2.x-p1.x)*(p1.y-p0.y)-(p1.x-p0.x)*(p2.y-p1.y))/sqrt((p2.x-p1.x)^2+(p2.y-p1.y)^2)
```

Solução do Exercício 2.4.2

```
numero_minimo_espelhos(p0, p1) =
  ceil((cy(p1) - cy(p0))/0.18)
```

Solução do Exercício 2.5.1 Este problema obriga-nos a trabalhar com tolerâncias, de modo que as comparações não sejam feitas em termos absolutos mas sim relativamente a uma dada tolerância ϵ, i.e., ao invés de $a = b$, testamos $\|a - b\| \leq \epsilon$. Comecemos então por definir a tolerância para a comparação de coordenadas:

```
tolerancia_posicoes = 1e-15
```

A comparação de coordenadas pode agora ser definida como:

```
posicoes_iguais(p0, p1) =
  distancia(p0, p1) <= tolerancia_posicoes
```

Solução do Exercício 2.6.1

```
circle(xy(0, 0), 4)
circle(xy(sqrt(8), sqrt(8)), 2)
circle(xy(sqrt(18), sqrt(18)), 1)
```

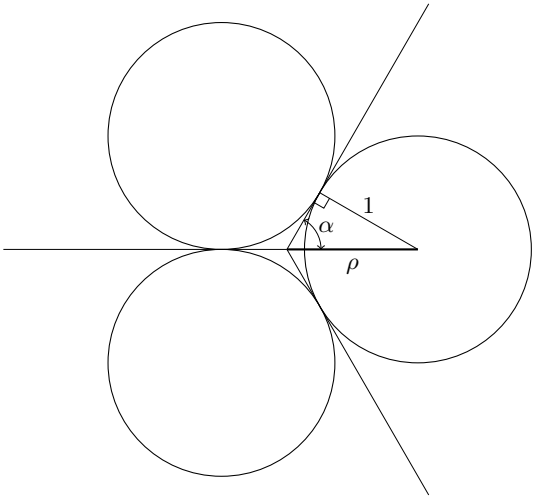
Solução do Exercício 2.6.2

```
circle(xy(-1, 0), 1)
circle(xy(+1, 0), 1)
```

Solução do Exercício 2.6.3

```
circle(xy(-1, -1), 1)
circle(xy(-1, +1), 1)
circle(xy(+1, +1), 1)
circle(xy(+1, -1), 1)
```

Solução do Exercício 2.6.4 Para se determinar a distância do centro das circunferências à origem é necessário empregar alguma trigonometria tal como é ilustrado na seguinte figura:



Da análise da figura depreendemos que a colocação das três circunferências implica a divisão do plano em três secções, cada uma a cobrir um ângulo $2\alpha = \frac{2\pi}{3}$. A distância ρ do centro de cada circunferência ao centro do plano pode então ser obtida pela relação trigonométrica

$$1 = \rho \sin \alpha$$

logo

$$\rho = \frac{1}{\sin \frac{\pi}{3}} = 1.1547$$

Uma vez que o plano está divido em 3 secções, os ângulos dos centros serão, sucessivamente, 0, $\frac{2\pi}{3}$ e $2 \cdot \frac{2\pi}{3} = \frac{4\pi}{3}$. Com base nestes dados é agora simples usar coordenadas polares para criar as três circunferências.

```
circle(pol(1.1547, 0/3*pi), 1)
circle(pol(1.1547, 2/3*pi), 1)
circle(pol(1.1547, 4/3*pi), 1)
```

Solução do Exercício 2.8.1

```
circulo_e_raio(p, r) =
  begin
    circle(p, r)
    text(string("Raio: ", r), p + vpol(r*1.25, 0), r*0.25)
  end
```

Solução do Exercício 2.8.2

```
circulo_e_raio(pol(0, 0), 4)
circulo_e_raio(pol(4, pi/4), 2)
circulo_e_raio(pol(6, pi/4), 1)
```

Solução do Exercício 2.11.1

Solução do Exercício 2.11.2

```
divisao_rectangular(p, comprimento, largura, funcao) =
  begin
    rectangle(p, p + vxy(comprimento, largura))
    text_centered(funcao, p + vxy(comprimento*0.5, largura*0.7), comprimento/8)
    text_centered("Area:${(comprimento*largura)}", p + vxy(comprimento*0.5, largura*0.3), comprimento
  end
```

Solução do Exercício 2.13.1

```
cruzeta(xyz(0, 0, 0), 2, 1, 5)
cruzeta(xyz(0, 0, 0), 0.5, 1, 0.5)
```

Solução do Exercício 2.13.2

```
ampulheta(p, rb, rc, h) =
  begin
    cone_frustum(p, rb, p + vz(h/2), rc)
    cone_frustum(p + vz(h/2), rc, p + vz(h), rb)
  end
```

Solução do Exercício 2.13.3

```
obelisco(p, b, h, bp, hp) =
  let l = b/2,
      l1 = bp/2,
      h0 = h - hp
    regular_pyramid_frustum(4, p, l, 0, h0, l1)
    regular_pyramid(4, p + vz(h0), l1, 0, hp)
  end

obelisco(u0(), 16.8, 169.3, 10.5, 16.9)
```

Solução do Exercício 2.13.4

```
obelisco_perfeito(p, h) =
  let b = h/10
    obelisco(p, b, h, 2/3*b, b)
  end

obelisco_perfeito(u0(), 168)
```

Solução do Exercício 2.13.5

```
prisma(n, p0, r, a, p1) =
  regular_pyramid_frustum(n, p0, r, a, p1, r)
```

Solução do Exercício 2.13.6

```
bloco_sears(p, i, j, l, h) =
  let r = l/2,
      pij = p + vxy(i*l + l/2, j*l + l/2)
    prisma(4, pij, r, pi/4, pij + vz(h))
  end

torre_sears(p, l, h00, h01, h02, h10, h11, h12, h20, h21, h22) =
  begin
    bloco_sears(p, 0, 0, l/3, h00)
    bloco_sears(p, 0, 1, l/3, h01)
    bloco_sears(p, 0, 2, l/3, h02)
    bloco_sears(p, 1, 0, l/3, h10)
    bloco_sears(p, 1, 1, l/3, h11)
    bloco_sears(p, 1, 2, l/3, h12)
    bloco_sears(p, 2, 0, l/3, h20)
    bloco_sears(p, 2, 1, l/3, h21)
    bloco_sears(p, 2, 2, l/3, h22)
  end

torre_sears(u0(), 68.7, 270, 442, 205, 368, 442, 368, 205, 368, 270)
```

Solução do Exercício 2.13.7

```
porta(p, c, h, e) =
  begin
    box(p + vx(-c/2 - e), e, e, h)
    box(p + vx(c/2), e, e, h)
    box(p + vxz(-c/2 - e, h), e + c + e, e, e)
  end

torre(p, cb, lb, ct, lt, h) =
  cuboid(p + vxyz(-cb/2, -lb/2, 0),
         p + vxyz(cb/2, -lb/2, 0),
         p + vxyz(cb/2, lb/2, 0),
         p + vxyz(-cb/2, lb/2, 0),
         p + vxyz(-ct/2, -lt/2, h),
         p + vxyz(ct/2, -lt/2, h),
         p + vxyz(ct/2, lt/2, h),
         p + vxyz(-ct/2, lt/2, h))

pilone(p, cb, lb, ct, lt, h, cp, hp, ep) =
  begin
    torre(p + vx(-(cp + cb)/2), cb, lb, ct, lt, h)
    torre(p + vx(+(cp + cb)/2), cb, lb, ct, lt, h)
    porta(p + vy(-lb/2), cp, hp, ep)
  end

pilone(xyz(0, 0, 0), 50, 20, 30, 10, 50, 6, 20, 8)
```

Solução do Exercício 2.14.1

```
cyl_rho(p) =
  sqrt(cx(p)^2 + cy(p)^2)

cyl_phi(p) =
  atan2(cy(p), cx(p))

cyl_z(p) =
  cz(p)
```

Solução do Exercício 2.14.2

```
degrau(p, r, fi, z) =
  cylinder(p + vcyl(0, 0, z), r, p + vcyl(10*r, fi, z))

escada(p, r, h, a) =
  begin
    cylinder(p, r, p + vxyz(0, 0, 10*h))
    degrau(p, r, a*0, h*1)
    degrau(p, r, a*1, h*2)
    degrau(p, r, a*2, h*3)
    degrau(p, r, a*3, h*4)
    degrau(p, r, a*4, h*5)
    degrau(p, r, a*5, h*6)
    degrau(p, r, a*6, h*7)
    degrau(p, r, a*7, h*8)
    degrau(p, r, a*8, h*9)
  end
```

Solução do Exercício 2.15.1

```
sph_rho(p) =
  sqrt(cx(p)^2 + cy(p)^2 + cz(p)^2)

sph_phi(p) =
  atan2(cy(p), cx(p))

sph_psi(p) =
  atan2(sqrt(cx(p)^2 + cy(p)^2), cz(p))
```

Solução do Exercício 2.15.2

Solução do Exercício 2.16.1

```
abaco(p, a_abaco, l_abaco) =
  box(p + vxyz(-l_abaco/2, -l_abaco/2, 0),
      p + vxyz(l_abaco/2, l_abaco/2, a_abaco))
```

Solução do Exercício 3.1.1

```
ackermann(m, n) =
  if m == 0
    n + 1
  elseif n == 0
    ackermann(m - 1, 1)
  else
    ackermann(m - 1, ackermann(m, n - 1))
  end
```

Solução do Exercício 3.1.2

1. $9 = 8 + 1$
2. $10 = 8 + 2$
3. $19 = 2 \times 8 + 3$
4. $2045 = 2^{(8+3)} - 3$
5. $2^{2^{2^{2^{2^{2^{2}}}}}} - 3$

Solução do Exercício 3.2.1

```
piramide_degraus(p, b, t, h0, h1, d, n) =
  if n == 0
    nothing
  else
    regular_pyramid_frustum(4, p, b, 0, h0, t)
    regular_pyramid_frustum(4, p + vz(h0), t, 0, h1, b - d)
    piramide_degraus(p + vz(h0 + h1), b - d, t - d, h0, h1, d, n - 1)
  end

piramide_degraus(xyz(0, 0, 0), 120, 115, 15, 5, 15, 6)
```

Solução do Exercício 3.2.2

```
arco_falso(p, c, e, de, l) =
  if e <= 0
    box(p + vxy(-c/2, -l/2), p + vxyz(c/2, l/2, l))
  else
    box(p + vxy(-c/2, -l/2), p + vxyz(-e/2, l/2, l))
    box(p + vxy(e/2, -l/2), p + vxyz(c/2, l/2, l))
    arco_falso(p + vz(l), c, e - de - de, de, l)
  end
```

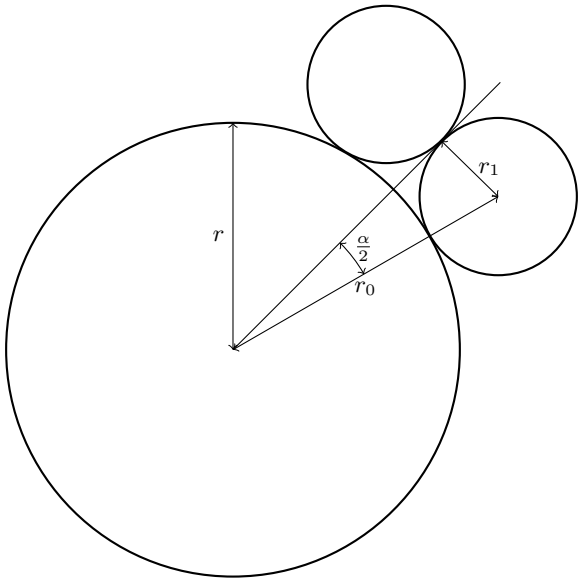
Solução do Exercício 3.2.3

```
equilibrio_circulos(p, r, f) =
  if r < 1
    nothing
  else
    circle(p, r)
    equilibrio_circulos(p + vy((1 + f)*r), f*r, f)
  end
```

Solução do Exercício 3.2.4

```
circulos_radiais(p, n, r0, r1, fi, d_fi) =
  if n == 0
    nothing
  else
    circle(p + vpol(r0, fi), r1)
    circulos_radiais(p, n - 1, r0, r1, fi + d_fi, d_fi)
  end
```

Solução do Exercício 3.2.5 A função `circulos_radiais` definida no exercício anterior simplifica bastante a resolução do problema, sendo apenas necessário calcular os parâmetros correctos. Para isso, podemos reportarmo-nos à figura seguinte:



Da figura depreende-se que

$$r_1 = r_0 \sin \frac{\alpha}{2}$$

Sendo $r_0 = r + r_1$, temos

$$r_1 = (r + r_1) \sin \frac{\alpha}{2}$$

ou seja

$$r_1 = r \frac{\sin \frac{\alpha}{2}}{1 - \sin \frac{\alpha}{2}}$$

Por outro lado, para uma flor de n pétalas, temos que $\alpha = \frac{2\pi}{n}$, o que implica que

$$r_1 = r \frac{\sin \frac{\pi}{n}}{1 - \sin \frac{\pi}{n}}$$

Estamos agora em condições de definir a função `flor`:

```
flor(p, raio, petalas) =
  let coef = sin(pi/petalas),
      r1 = raio*coef/(1 - coef),
      r0 = raio + r1
    circle(p, raio)
    circulos_radiais(p, petalas, r0, r1, 0, 2*pi/petalas)
  end
```

Solução do Exercício 3.2.6

```
circulos(p, raio) =
  if raio < 1
    nothing
  else
    circle(p, raio)
    circulos(p + vx(raio), raio/2)
    circulos(p + vy(raio), raio/2)
  end
```

Solução do Exercício 3.2.7

```
serra(p0, dentes, comprimento, altura) =
  if dentes == 0
    nothing
  else
    let p1 = p0 + vxy(comprimento/2, altura),
        p2 = p0 + vx(comprimento)
      line(p0, p1, p2)
      serra(p2, dentes - 1, comprimento, altura)
    end
  end
```

Solução do Exercício 3.2.8

```
losangos(p, c) =
  if c < 1
    nothing
  else
    let p0 = p + vpol(c, 0),
        p1 = p + vpol(c, pi/2),
        p2 = p + vpol(c, pi),
        p3 = p + vpol(c, 3*pi/2),
        c2 = c/2.0
      line(p0, p1, p2, p3, p0)
      losangos(p0, c2)
      losangos(p1, c2)
      losangos(p2, c2)
      losangos(p3, c2)
    end
  end
```

Solução do Exercício 3.2.9

```
escada_rampa(p, alfa, c, n) =
  if n == 0
    nothing
  else
    let e = c*tan(alfa),
        p1 = p + vy(e),
        p2 = p1 + vx(c)
      line(p, p1, p2)
      escada_rampa(p2, alfa, c, n - 1)
    end
  end
```

Solução do Exercício 3.2.10

```
escada_progressao_geometrica(p, alfa, c, n, f) =
  if n == 0
    nothing
  else
    let e = c*tan(alfa),
        p1 = p + vy(e),
        p2 = p1 + vx(c)
      line(p, p1, p2)
      escada_progressao_geometrica(p2, alfa, c*f, n - 1, f)
    end
  end
```

Solução do Exercício 3.3.1 Dados os pontos inicial P e final Q, é fácil constatarmos que o vector v que mede a separação entre as n colunas é determinado por

$$\vec{v} = \frac{Q - P}{n}$$

Assim, podemos definir uma função que coloca as colunas entre dois pontos quaisquer `p` e `q`:

```
colunas_doricas_entre(p, q, h, n) =
  colunas_doricas(p, h, (q - p)/n, n)
```

Solução do Exercício 3.3.2

```
colunas_tholos(p, n_colunas, raio, fi, d_fi, altura) =
  if n_colunas == 0
    nothing
  else
    coluna_dorica(loc_from_o_phi(p + vpol(raio, fi), fi), altura)
    colunas_tholos(p, n_colunas - 1, raio, fi + d_fi, d_fi, altura)
  end
```

Solução do Exercício 3.3.3

```
torre_tholos(p, n_modulos, n_degraus_topo, n_degraus, rb, dab, drb, n_colunas, rp, ap) =
  if n_modulos == 0
    base_tholos(p, n_degraus_topo, rb, dab, drb)
  else
    tholos(p, n_degraus, rb, dab, drb, n_colunas, rp, ap)
    torre_tholos(p + vxyz(0, 0, n_degraus*dab + ap),
                 n_modulos - 1, n_degraus_topo, n_degraus,
                 rb, dab, drb, n_colunas, rp, ap)
  end

torre_tholos(xyz(0, 0, 0), 6, 10, 3, 7.9, 0.2, 0.2, 20, 7, 4)
```

Solução do Exercício 3.3.4

```
torre_tholos(p, n_modulos, n_degraus_topo, n_degraus, rb, dab, drb,
    n_colunas, rp, ap) =
  if n_modulos == 0
    base_tholos(p, n_degraus_topo, rb, dab, drb)
  else
    tholos(p, n_degraus, rb, dab, drb, n_colunas, rp, ap)
    torre_tholos(p + vxyz(0, 0, n_degraus*dab + ap),
                 n_modulos - 1, n_degraus_topo, n_degraus,
                 0.9*rb, dab, drb, n_colunas, 0.9*rp, ap)
  end

torre_tholos(xyz(0, 0, 0), 6, 10, 3, 8.5, 0.2, 0.2, 20, 7, 4)
```

Solução do Exercício 3.3.5

Solução do Exercício 3.3.6

```
cidade_espacial(p, raio) =
  begin
    sphere(p, raio/8.0)
    if raio < 1
      nothing
    else
      let r2 = raio/2,
          nx = p - vx(r2),
          px = p + vx(r2),
          ny = p - vy(r2),
          py = p + vy(r2),
          nz = p - vz(r2),
          pz = p + vz(r2)
        cylinder(nx, raio/32.0, px)
        cylinder(ny, raio/32.0, py)
        cylinder(nz, raio/32.0, pz)
        cidade_espacial(nx, r2)
        cidade_espacial(px, r2)
        cidade_espacial(ny, r2)
        cidade_espacial(py, r2)
        cidade_espacial(nz, r2)
        cidade_espacial(pz, r2)
      end
    end
  end




cidade_espacial(xyz(0, 0, 0), 8)
```

Solução do Exercício 3.4.1

```
arco_espiral(p, r, a_ini, a_inc) =
  begin
    arc(p, r, a_ini, a_inc)
    rectangle(p, p + vpol(sqrt(2)*r, a_ini + a_inc/2))
  end

espiral(xy(40, 0), 1, pi, pi/2, pi*9, 1.618)
```

Solução do Exercício 3.4.2 Para que os arcos de circunferência possam interligar-se suavemente, é necessário que exista uma relação entre os raios r_0, r_1 e r_2. É facil ver que

$$(r_2 - r_1)\cos\alpha + r_0 = r_2$$

Por outro lado, a altura h do óvulo é dada por

$$h = r_0 + (r_2 - r_1)\sin\alpha + r_1$$

Uma vez que a função não recebe nem o raio r_2 nem o ângulo α, temos de os deduzir. Para isso, vamos resolver a primeira equação em ordem a r_2

$$r_2 = \frac{r_0 - r_1\cos\alpha}{1 - \cos\alpha}$$

e vamos substituir na segunda equação

$$h = r_0 + (\frac{r_0 - r_1\cos\alpha}{1 - \cos\alpha} - r_1)\sin\alpha + r_1$$

Simplificando, obtemos

$$h = r_0 + r_1 + (r_0 - r_1)\frac{\sin\alpha}{1 - \cos\alpha}$$

Empregando agora a igualdade trigonométrica

$$\tan\frac{\alpha}{2} = \frac{1 - \cos\alpha}{\sin\alpha}$$

temos

$$h = r_0 + r_1 + (r_0 - r_1)\frac{1}{\tan\frac{\alpha}{2}}$$

ou seja,

$$\alpha = 2\tan^{-1}\frac{r_0 - r_1}{h - r_0 - r_1}$$

Estamos agora em condições de definir a função pretendida:

```
ovo(p, r0, r1, h) =
  let alfa = 2*atan2(r0 - r1, h - r0 - r1),
      r2 = (r0 - r1*cos(alfa))/(1 - cos(alfa))
    arc(p, r0, 0, -pi)
    arc(p + vx(r0 - r2), r2, 0, alfa)
    arc(p + vx(r2 - r0), r2, pi - alfa, alfa)
    arc(p + vy((r2 - r1)*sin(alfa)), r1, alfa, pi - alfa - alfa)
  end
```

Solução do Exercício 3.4.3

```
piramide_cilindros(p, l, r, f, a, d, da) =
  if r < 0.01
    nothing
  else
    cylinder(p + vpol(l, a + -d), r, p + vpol(l, a + d + pi/2))
    cylinder(p + vpol(l, a + pi - d), r, p + vpol(l, a + d + 3*pi/2))
    piramide_cilindros(p + vz((1 + f)*r), l*f, r*f, f, a + pi/2 + da, d, da)
  end
```

Solução do Exercício 4.4.1

```
folha(topo) =
  sphere(topo, 0.5)

ramo(p, topo) =
  let raio = distance(p, topo)/10.0
    cone_frustum(p, raio, topo, raio*0.9)
  end

arvore(p, c, fi, psi, min_fi, max_fi, min_psi, max_psi, min_f, max_f) =
  let topo = p + vsph(c, fi, psi)
    ramo(p, topo)
    if c < 2
      folha(topo)
    else
      arvore(topo,
             c*random_range(min_f, max_f),
             fi + random_range(min_fi, max_fi),
             psi + random_range(min_psi, max_psi),
             min_fi, max_fi, min_psi, max_psi, min_f, max_f)
      arvore(topo,
             c*random_range(min_f, max_f),
             fi - random_range(min_fi, max_fi),
             psi - random_range(min_psi, max_psi),
             min_fi, max_fi, min_psi, max_psi, min_f, max_f)
    end
  end
```

Solução do Exercício 4.4.2

```
posicao_aleatoria() =
  xyz(random_range(0, 200), random_range(0, 100), random_range(0, 100))


cilindros_aleatorios(n) =
  if n == 0
    nothing
  else
    cylinder(posicao_aleatoria(), random_range(1, 10), posicao_aleatoria())
    cilindros_aleatorios(n - 1)
  end
```

Solução do Exercício 4.4.3

```
caminhada_aleatoria(p, d, n) =
  if n == 0
    nothing
  else
    let p1 = p + vpol(random_range(0, d), random_range(0, 2*pi))
      if p1 != p
        line(p, p1)
      end
      caminhada_aleatoria(p1, d, n - 1)
    end
  end

caminhada_aleatoria(u0(), 5, 100)
```

Solução do Exercício 4.4.4

```
cara_ou_coroa() =
  if random(10) < 3
    "cara"
  else
    "coroa"
  end
```

Solução do Exercício 4.4.5

```
cilindros_esfera(p, r, rc, n) =
  if n == 0
    nothing
  else
    cylinder(p, rc, p + vsph(r, random_range(0, 2*pi), random_range(0, pi)))
    cilindros_esfera(p, r, rc, n - 1)
  end
```

Solução do Exercício 4.4.6

```
round_vector(v) =
  vxyz(round(v.x), round(v.y), round(v.z))

direccao_ortogonal_aleatoria_excepto(v0) =
  let v1 = round_vector(sph(1, random_range(0, 4)*pi/2, random_range(0, 3)*pi/2))
    if v0 == v1 || v0 == v1*-1
      direccao_ortogonal_aleatoria_excepto(v0)
    else
      v1
    end
  end

blocos_conexos(p, v, c, l, n) =
  if n == 0
    nothing
  else
    right_cuboid(p, l, l, p + v*c)
    let v1 = direccao_ortogonal_aleatoria_excepto(v),
        p1 = p + v*(c - l/2.0) + v1*l/2.0
      blocos_conexos(p1, v1, c, l, n - 1)
    end
  end
```

Solução do Exercício 4.5.1

```
predio0(p, l, h) =
  box(p, l, l, random_range(0.1, 1.0)*h)

predio1(p, l, h) =
  cylinder(p + vxy(l/2, l/2), l/2, random_range(0.1, 1.0)*h)
```

Solução do Exercício 4.5.2

```
predio(p, l, h) =
  if random(5) == 0
    predio1(p, l, h)
  else
    predio0(p, l, h)
  end
```

Solução do Exercício 4.5.3

```
predio_blocos(n, p0, c0, l0, h0) =
  if n == 1
    box(p0, c0, l0, h0)
  else
    let c1 = random_range(0.7, 1.0)*c0,
        l1 = random_range(0.7, 1.0)*l0,
        h1 = random_range(0.2, 0.8)*h0,
        p1 = p0 + vxyz((c0 - c1)/2.0, (l0 - l1)/2.0, h1)
      box(p0, c0, l0, h1)
      predio_blocos(n - 1, p1, c1, l1, h0 - h1)
    end
  end

predio0(p, l, h) =
  predio_blocos(random_range(1, 6), p, l, l, h)
```

Solução do Exercício 4.5.4

```
gaussiana_2d(x, y, sigma) =
  exp(-((x/sigma)^2 + (y/sigma)^2))

predio(p, l, h) =
  let h = h*max(0.1, gaussiana_2d(cx(p), cy(p), 25.0*l))
    if random(5) == 0
      predio1(p, l, h)
    else
      predio0(p, l, h)
    end
  end
```

Solução do Exercício 4.5.5

```
predio(p, l, h) =
  let coef = max(gaussiana_2d(500 - cx(p), 500 - cy(p), 11*l),
                 gaussiana_2d(2000 - cx(p), 5500 - cy(p), 14*l),
                 gaussiana_2d(4500 - cx(p), 2500 - cy(p), 13*l))
    if coef > 0.1
      let h = h*max(coef, 0.1)
        if random(5) == 0
          predio1(p, l, h)
        else
          predio0(p, l, h)
        end
      end
    end
  end
```

Solução do Exercício 4.5.6

```
esferas_na_esfera(p, ri, re, rl, n) =
  if n == 0
    nothing
  else
    let r = random_range(ri, re)
      sphere(p + vsph(r, random_range(0, 2*pi), random_range(0, pi)), rl - r)
      esferas_na_esfera(p, ri, re, rl, n - 1)
    end
  end
```

Solução do Exercício 4.5.7

```
delete_all_shapes()

random_element(l) =
  l[random(length(l))+1]

misterio(ps, n) =
  let p = ps[0]
    for i in range(n):
      let p = intermediate_loc(p, random_element(ps))
        surface_circle(p, 1)
      end
  end

misterio([xy(0, 0), xy(200, 0), xy(100, 200)], 5000)
```

Solução do Exercício 5.3.1

```
elemento_aleatorio(lista) =
  lista[random(length(lista))+1]
```

Solução do Exercício 5.3.2

```
um_de_cada(listas) =
  if listas == []
    []
  else
    [elemento_aleatorio(listas[1]), um_de_cada(listas[2:end])...]
  end
```

Solução do Exercício 5.3.3

```
elementos_aleatorios(n, lista) =
  if n == 0
    []
  else
    let i = random(length(lista))+1
      [lista[i], elementos_aleatorios(n-1, [lista[1:i-1]..., lista[i+1:end]...])...]
    end
  end
```

Solução do Exercício 5.3.4

Solução do Exercício 5.3.5

```
iota(b, i) =
  enumera(0, b-1, i)
```

Solução do Exercício 5.3.6

```
pertence(num, lista) =
  if lista == []
    false
  elseif num == lista[1]
    true
  else
    pertence(num, lista[2:end])
  end
```

Solução do Exercício 5.3.7

```
elimina1(num, lista) =
  if lista == []
    []
  elseif num == lista[1]
    lista[2:end]
  else
    [lista[1], elimina1(num, lista[2:end])...]
  end
```

Solução do Exercício 5.3.8

```
elimina(num, lista) =
  if lista == []
    []
  elseif num == lista[1]
    elimina(num, lista[2:end])
  else
    [lista[1], elimina(num, lista[2:end])...]
  end
```

Solução do Exercício 5.3.9

```
substitui(novo, velho, lista) =
  if lista == []
    []
  elseif velho == lista[1]
    [novo, substitui(novo, velho, lista[2:end])...]
  else
    [lista[1], substitui(novo, velho, lista[2:end])...]
  end
```

Solução do Exercício 5.3.10

```
remove_duplicados(lista) =
  if lista == []
    []
  elseif pertence(lista[1], lista[2:end])
    remove_duplicados(lista[2:end])
  else
    [lista[1], remove_duplicados(lista[2:end])...]
  end
```

Solução do Exercício 5.3.11

```
ocorrencias(num, lista) =
  if lista == []
    0
  elseif num == lista[1]
    1+ocorrencias(num, lista[2:end])
  else
    ocorrencias(num, lista[2:end])
  end
```

Solução do Exercício 5.3.12

```
posicao(num, lista) =
  if num == lista[1]
    0
  else
    1+posicao(num, lista[2:end])
  end
```

Solução do Exercício 5.4.1

```
divide_poligono(pts, i, j) =
  [[pts[1:i+2]..., pts[j+1:end]...], pts[i+1:j+2]]
```

Solução do Exercício 5.4.2

```
ponto_intermedio(p0, p1, f) =
  p0+(p1-p0)*f

bisseccao_poligono(pts, i, j, fi, fj) =
  let pti = ponto_intermedio(pts[i+1], pts[i+1+1], fi),
      ptj = ponto_intermedio(pts[j+1], [pts..., pts...][j+2], fj)
    [[pts[1:i+1+1]..., pti, ptj, pts[j+2:end]...], [ptj, pti, pts[i+2:j+2]...]]
  end
```

Solução do Exercício 5.4.3

```
bisseccao_aleatoria_poligono(pts) =
  let l = length(pts),
      ij = enumera(0, l, 1),
      i = ij[random(l) + 1],
      j = [e for e in ij if e != i][random(l - 1) + 1]
    bisseccao_poligono(pts,
                       min(i, j),
                       max(i, j),
                       random_range(0.0, 1.0),
                       random_range(0.0, 1.0))
  end
```

Solução do Exercício 5.4.4

```
divisao_aleatoria_poligonos(pols, n) =
  if pols == []
    []
  else
    [divisao_aleatoria_poligono(pols[1], n)...,
     divisao_aleatoria_poligonos(pols[2:end], n)...]
  end


divisao_aleatoria_poligono(pts, n) =
  if n == 0
    [pts]
  else
    divisao_aleatoria_poligonos(bisseccao_aleatoria_poligono(pts), n-1)
  end
```

Solução do Exercício 5.5.1

```
pontos_arco_sinusoidal(p, r, a, c, fi, dfi, n) =
  if n == 0
    []
  else
    [p + vpol(r + a*sin(c*fi), fi),
     pontos_arco_sinusoidal(p, r, a, c, fi+dfi, dfi, n-1)...]
  end


pontos_sinusoide_circular(p, ri, re, c, n) =
  pontos_arco_sinusoidal(p, (ri+re)/2, (re-ri)/2, c, 0, 2*pi/n, n)
```

Solução do Exercício 5.5.2

```
pontos_arco_raio_aleatorio(p, r0, r1, fi, dfi, n) =
  if n == 0
    []
  else
    [p+vpol(random_range(r0, r1), fi),
     pontos_arco_raio_aleatorio(p, r0, r1, fi+dfi, dfi, n-1)...]
  end

pontos_circulo_raio_aleatorio(p, r0, r1, n) =
  pontos_arco_raio_aleatorio(p, r0, r1, 0, 2*pi/n, n)
```

Solução do Exercício 5.6.1

```
coordenadas_x(p, l, n) =
  if n == 0
    []
  else
    [p, coordenadas_x(p+vx(l), l, n-1)...]
  end

trelica_recta(p, h, l, n) =
  trelica(coordenadas_x(p, l, n),
          coordenadas_x(p+vxyz(l/2, l/2, h), l, n-1),
          coordenadas_x(p+vy(l), l, n))
```

Solução do Exercício 5.6.2 Relativamente à figura anterior, a linha $c_i a_{i+1}$ tem comprimento $\sqrt{l^2 + l^2} = \sqrt{2}l$. A distância entre o ponto c_i e o centro da base é, consequentemente, $\frac{\sqrt{2}l}{2}$. Para que as barras tenham todas o mesmo tamanho, a barra $c_i b_i$ terá também de ter comprimento l. Ora como esta barra é a hipotenusa do triângulo com catetos de comprimento h e $\frac{\sqrt{2}l}{2}$, temos que $l^2 = h^2 + (\frac{\sqrt{2}l}{2})^2$, ou seja

$$h = \frac{l}{\sqrt{2}}$$

Assim, podemos definir:

```
trelica_modulo(p, l, n) =
  trelica_recta(p, l/sqrt(2), l, n)
```

Solução do Exercício 5.6.3

```
trelica_plana(ais, bis) =
  begin
    nos_trelica(ais)
    nos_trelica(bis)
    barras_trelica(ais, bis)
    barras_trelica(bis, ais[2:end])
    barras_trelica(ais, ais[2:end])
    barras_trelica(bis, bis[2:end])
  end
```

Solução do Exercício 5.6.4

```
media(a, b) =
  (a+b)/2

ponto_medio(p, q) =
  xyz(media(cx(p), cx(q)), media(cy(p), cy(q)), media(cz(p), cz(q)))

pontos_medios(ps, qs) =
  if ps == []
    []
  else
    [ponto_medio(ps[1], qs[1]), pontos_medios(ps[2:end], qs[2:end])...]
  end

trelica_especial(ais, bis, cis) =
  let dis = pontos_medios(ais, cis)
    nos_trelica(ais)
    nos_trelica(bis)
    nos_trelica(cis)
    nos_trelica(dis)
    barras_trelica(ais, cis)
    barras_trelica(bis, dis)
    barras_trelica(bis, dis[2:end])
    barras_trelica(ais, ais[2:end])
    barras_trelica(cis, cis[2:end])
    barras_trelica(bis, bis[2:end])
  end
```

Solução do Exercício 5.6.5 O primeiro passo consiste na dedução das fórmulas que relacionam a largura e e o ângulo inicial ψ_0 com o raio r e com o número n_ϕ de treliças pretendidas.

Para isso, através da observação da figura anterior é fácil constatar que $r_0 \sin \psi_0 = r \cos \alpha$. Sendo $e = 2r \sin(\frac{\alpha}{2})$ e $\alpha = \frac{2\pi}{n_\phi}$, temos

$$e = 2r \sin \frac{\pi}{n_\phi}$$

e ainda

$$\psi_0 = \text{asin} \frac{r \cos \frac{\pi}{n_\phi}}{r_0}$$

O segundo passo consiste em iterar a função `trelica_arco` para sucessivos valores do ângulo ϕ a variar entre 0 e 2π em incrementos de $\frac{2pi}{n_\phi}$:

```
abobada_trelicas(p, rac, rb, rf, n, n_fi) =
  let e = 2*rf*sin(pi/n_fi),
      psi0 = asin(rf*cos(pi/n_fi)/rac)
    [trelica_arco(p, rac, rb, fi, psi0, pi/2, e, n) for fi in division(0, 2*pi, n_fi, false)]
  end
```

Solução do Exercício 5.6.6

```
escada_de_mao(ais, bis) =
  begin
    nos_trelica(ais)
    nos_trelica(bis)
    barras_trelica(ais, bis)
    barras_trelica(ais, ais[2:end])
    barras_trelica(bis, bis[2:end])
  end
```

Solução do Exercício 5.6.7

```
coordenadas_helice(p, r, fi, dfi, dz, n) =
  if n == 0
    []
  else
    [p+vpol(r, fi),
     coordenadas_helice(p+vz(dz), r, fi+dfi, dfi, dz, n-1)...]
  end


genoma(p, r, dfi, dz, n) =
  escada_de_mao(coordenadas_helice(p, r, 0, dfi, dz, n),
                coordenadas_helice(p, r, pi, dfi, dz, n))
```

Solução do Exercício 5.6.8

```
trelica(ais, bis, cis) =
  trelica_espacial([ais, bis, cis])
```

Solução do Exercício 5.6.9

```
coordenadas_trelica_aleatoria(p, h, l, m, n, r) =
  if m == 0
    [coordenadas_linha_aleatoria(p, l, n, r)]
  else
    [coordenadas_linha_aleatoria(p, l, n, r),
     coordenadas_linha_aleatoria(p+vxyz(l/2, l/2, h), l, n-1, r),
     coordenadas_trelica_aleatoria(p+vy(l), h, l, m-1, n, r)...]
  end

coordenadas_linha_aleatoria(p, l, n, r) =
  if n == 0
    [p+vxyz_aleatorio(r)]
  else
    [p+vxyz_aleatorio(r), coordenadas_linha_aleatoria(p+vx(l), l, n-1, r)...]
  end

vxyz_aleatorio(r) =
  vxyz(random_range(-r, +r), random_range(-r, +r), random_range(-r, +r))
```

Solução do Exercício 5.6.10

```
coordenadas_piramides_arco(p, rac, rb, l, fi, psi0, psi1, dpsi, m) =
  if m == 0
    [pontos_arco(p+vpol(l/2, fi-pi/2), rac, fi, psi0, psi1, dpsi)]
  else
    [pontos_arco(p+vpol(l/2, fi-pi/2), rac, fi, psi0, psi1, dpsi),
     pontos_arco(p, rb, fi, psi0+dpsi/2, psi1-dpsi/2, dpsi),
     coordenadas_piramides_arco(p+vpol(l, fi+pi/2),
                                rac, rb, l, fi, psi0, psi1, dpsi, m-1)...]
  end

trelica_espacial_arco(p, rac, rb, l, n, fi, psi0, psi1, m) =
  trelica_espacial(
    coordenadas_piramides_arco(p, rac, rb, l, fi, psi0, psi1, (psi1-psi0)/n, m))
```

Solução do Exercício 5.6.11

```
coordenadas_trelica_ondulada(p, rac, rb, l, fi, psi0, psi1, dpsi, alfa0, alfa1, d_alfa, d_r) =
  if alfa0 >= alfa1
    [pontos_arco(p+vpol(l/2.0, fi-pi/2), rac+d_r*sin(alfa0), fi, psi0, psi1, dpsi)]
  else
    [pontos_arco(p+vpol(l/2.0, fi-pi/2), rac+d_r*sin(alfa0), fi, psi0, psi1, dpsi),
     pontos_arco(p, rb+d_r*sin(alfa0), fi, psi0+dpsi/2, psi1-dpsi/2, dpsi),
     coordenadas_trelica_ondulada(p+vpol(l, fi+pi/2), rac, rb, l, fi, psi0, psi1, dpsi, alfa0+d_alfa, alfa1, d_a
  end

trelica_ondulada(p, rac, rb, l, n, fi, psi0, psi1, alfa0, alfa1, d_alfa, d_r) =
    trelica_espacial(
        coordenadas_trelica_ondulada(
            p, rac, rb, l,
            fi, psi0, psi1, (psi1 - psi0)/n,
            alfa0, alfa1, d_alfa, d_r))
```

Solução do Exercício 6.2.1

```
bacia(p, r, e) =
  let re = r+e
    subtraction(box(p-vxyz(re, re, re),
                    p+vxyz(re, re, 0)),
                sphere(p, r))
  end
```

Solução do Exercício 6.2.2

```
banheira(p, r, e, l) =
  let re = r + e
    subtraction(box(p - vxyz(re, re, re),
                    p + vxyz(l + re, re, 0)),
                union(sphere(p, r),
                      cylinder(p, r, p + vx(l)),
                      sphere(p + vx(l), r)))
  end
```

Solução do Exercício 6.4.1

$$(R^C)^C = R$$

$$(R_0 \cup R_1)^C = R_0^C \cap R_1^C$$

$$(R_0 \cap R_1)^C = R_0^C \cup R_1^C$$

$$R \cup R^C = U$$

$$R \cap R^C = \varnothing$$

$$\varnothing^C = U$$

$$U^C = \varnothing$$

$$R_0 \cap R_1^C = R_0 \setminus R_1$$

$$R_0^C \cup R_1 = (R_0 \setminus R_1)^C$$

Solução do Exercício 6.4.2

```
is_complemento(objecto) =
  isinstance(objecto, tuple) && objecto[1] is "-"

complemento(regiao) =
  if is_empty_shape(regiao)
    universal_shape()
  elseif is_universal_shape(regiao)
    empty_shape()
  elseif is_complemento(regiao)
    regiao[2:end]
  else
    "-", regiao
  end
```

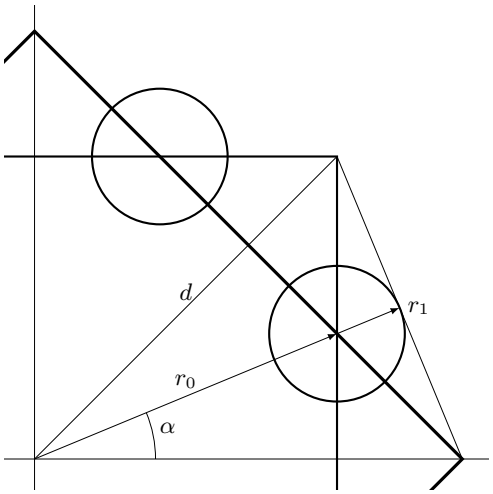
Solução do Exercício 6.4.3

```
uniao(r0, r1) =
  if is_complemento(r0) && is_complemento(r1)
    complemento(intersection(complemento(r0), complemento(r1)))
  elseif is_complemento(r0)
    complemento(subtraccao(complemento(r0), r1))
  elseif is_complemento(r1)
    complemento(subtraccao(r0, complemento(r1)))
  else
    union(r0, r1)
  end


interseccao(r0, r1) =
  if is_complemento(r0) && is_complemento(r1)
    complemento(uniao(complemento(r0), complemento(r1)))
  elseif is_complemento(r0)
    subtraccao(r1, complemento(r0))
  elseif is_complemento(r1)
    subtraccao(r0, complemento(r1))
  else
    intersection(r0, r1)
  end


subtraccao(r0, r1) =
  if is_complemento(r0) && is_complemento(r1)
    subtraction(complemento(r1), complemento(r0))
  elseif is_complemento(r0)
    complemento(uniao(complemento(r0), r1))
  elseif is_complemento(r1)
    complemento(uniao(r0, complemento(r1)))
  else
    subtraction(r0, r1)
  end
```

Solução do Exercício 6.4.4

```
poligonos_recursivos(p, r, fi, alfa_r, n, nivel) =
  if nivel == 0
    empty_shape()
  else
    let pontos = regular_polygon_vertices(n, p, r, fi, true)
      union([surface_polygon(pontos),
             lista_poligonos_recursivos(pontos, r*alfa_r, fi, alfa_r, n, nivel-1)...])
    end
  end

lista_poligonos_recursivos(pontos, r, fi, alfa_r, n, nivel) =
  if pontos == []
    []
  else
    [poligonos_recursivos(pontos[1], r, fi, alfa_r, n, nivel),
     lista_poligonos_recursivos(pontos[2:end], r, fi, alfa_r, n, nivel)...]
  end
```

Solução do Exercício 6.4.5 Uma vez que toda a geometria da secção se define a partir da largura l, é necessário deduzir os restantes parâmetros a partir de l. O seguinte esquema mostra as relações entre os vários parâmetros.



Pela observação do esquema é evidente que $\alpha = \frac{\pi}{8}$. Também sabemos pela definição do cosseno que $r_0 \cos\alpha = \frac{l}{2}$, ou seja, $r_0 = \frac{l}{2\cos\alpha}$. Para o cálculo de r_1 podemos considerar o triângulo cujo cateto é $r_0 + r_1$ e cuja hipotenusa é d. O ângulo entre estes dois lados do triângulo é α e a hipotenusa é $d = l/\sqrt{2}$. Logo, temos $r_0 + r_1 = d\cos\alpha$, i.e., $r_1 = l(\frac{\cos\alpha}{\sqrt{2}} - \frac{1}{2\cos\alpha})$.

Com base nestas equações podemos definir a função `seccao_petronas`:

```
seccao_petronas(p, l) =
  let d = l/sqrt(2),
      cos_a = cos(pi/8),
      r0 = l/2*cos_a,
      r1 = l*(cos_a/sqrt(2)-1/2*cos_a)
    union(surface_regular_polygon(4, p, d, 0),
          surface_regular_polygon(4, p, d, pi/4),
          uniao_circulos(p, r0, pi/8, pi/4, r1, 8))
  end
```

Solução do Exercício 6.4.6

```
tronco_cone_petronas(p, r, dr, h, n) =
  if n == 0
    p
  else
    seccao_petronas(p, r)
    tronco_cone_petronas(p+vz(h), r+dr, dr, h, n-1)
  end

bloco_petronas(p, r0, r1, r2, h, n) =
  let p = tronco_cone_petronas(p, r0, (r1-r0)/1.0/n, h, n)
    seccao_petronas(p, r2)
    p+vz(h)
  end

torre_petronas(p) =
  let p = bloco_petronas(p, 37, 37, 37, 5, 6),
      p = bloco_petronas(p, 35, 35, 36, 4, 52),
      p = bloco_petronas(p, 32, 32, 33, 4, 12),
      p = bloco_petronas(p, 28, 27, 28, 4, 8),
      p = bloco_petronas(p, 24, 21, 22, 4, 5),
      p = bloco_petronas(p, 18, 15, 16, 4, 4),
      p = bloco_petronas(p, 14, 5, 4, 3, 5)
  end

torre_petronas(xy(0, 0))
torre_petronas(xy(96, 0))
```

Solução do Exercício 6.4.7

```
malha_tubos(p, c, r, n, m) =
  if m == 0
    empty_shape()
  else
    union(linha_tubos(p, c, r, n),
          malha_tubos(p+vz(2*r), c, r, n, m-1))
  end


linha_tubos(p, c, r, n) =
  if n == 0
    empty_shape()
  else
    union(subtraction(cylinder(p, r, p+vy(c)),
                      cylinder(p, 0.9*r, p+vy(c))),
          linha_tubos(p+vx(2*r), c, r, n-1))
  end


cobertura_tubos(p, h, n) =
  let r1 = h/2.0/n,
      r0 = h-r1
    subtraction(malha_tubos(p+vxyz(-r0, 0, r1), h, r1, 2*n, n),
                sphere(p, r0))
  end


cobertura_tubos(xyz(0, 0, 0), 5, 9)
```

Solução do Exercício 6.4.8

```
malha_tubos(p, c, r, n, m) =
  if m == 0
    empty_shape()
  else
    union(linha_tubos(p, c, r, n),
          malha_tubos(p+vz(2*r), c, r, n, m-1))
  end

linha_tubos(p, c, r, n) =
  if n == 0
    empty_shape()
  else
    union(subtraction(cylinder(p, r, p+vx(c)),
                      cylinder(p, 0.9*r, p+vx(c))),
          linha_tubos(p+vy(2*r), c, r, n-1))
  end

cobertura_tubos(p, h, n) =
  let r1 = h/2.0/n,
      r0 = h-r1
    subtraction(malha_tubos(p+vxyz(-h, r1, r1), 2*h, r1, n, n),
                sphere(p, r0))
  end
```

Solução do Exercício 6.4.9

```
malha_tubos(p, c, r, n, m) =
  if m == 0
    empty_shape()
  else
    union(linha_tubos(p, c, r, n),
          malha_tubos(p+vy(2*r), c, r, n, m-1))
  end

linha_tubos(p, c, r, n) =
  if n == 0
    empty_shape()
  else
    union(subtraction(cylinder(p, r, p+vz(c)),
                      cylinder(p, 0.9*r, p+vz(c))),
          linha_tubos(p+vx(2*r), c, r, n-1))
  end

cobertura_tubos(p, h, n) =
  let r1 = h/2.0/n,
      r0 = h-r1
    subtraction(malha_tubos(p+vxyz(-r0, r1, 0), h, r1, 2*n, n),
                sphere(p, r0))
  end
```

Solução do Exercício 6.4.10 Uma vez que se pretende uma "esfera," o melhor será usar coordenadas esféricas para os centros das bases dos cones.

```
[cone(sph(1, pi*fi, pi*psi), 0.1*sin(pi*psi), u0())
 for fi in 0:1/10:2
 for psi in 1/30:7/75:1]
```

Solução do Exercício 6.4.11

```
for psi in 0:1/10:1
  let n = floor(15*sin(pi*psi))
    for i in 0:n
      cone(sph(1, i*2*pi/n, pi*psi), 0.1, u0())
    end
  end
end
```

Solução do Exercício 6.4.12

```
casca_esferica_perfurada(p, r, e, rc, n) =
  subtraction([sphere(p, r),
               sphere(p, r-e),
               [cone(p+vsph(1.1*r, i*2*pi/n, pi*psi), 0.1, p)
                for psi in 0:1/10:1
                for i in 0:1:floor(15*sin(pi*psi))+1]...])
```

Solução do Exercício 6.4.13

```
coordenada_aleatoria(p0, p1) =
  xyz(random_range(cx(p0), cx(p1)),
      random_range(cy(p0), cy(p1)),
      random_range(cz(p0), cz(p1)))


lista_coordenadas_aleatorias(p0, p1, n) =
  [coordenada_aleatoria(p0, p1) for i in 1:n]


lista_valores_aleatorios(a, b, n) =
  [random_range(a, b) for i in 1:n]


uniao_esferas(cs, rs, e) =
  union([sphere(c, r-e) for (c, r) in zip(cs, rs)])


uniao_cascas_esfericas_perfuradas(cs, rs, e, rc, n) =
  union([casca_esferica_perfurada(c, r, e, rc, n) for (c, r) in zip(cs, rs)])


cascas_esfericas_perfuradas_aleatorias(p0, p1, r0, r1, e, n, rc, m) =
  let cs = lista_coordenadas_aleatorias(p0, p1, n),
      rs = lista_valores_aleatorios(r0, r1, n)
    subtraction(uniao_cascas_esfericas_perfuradas(cs, rs, e, rc, m),
                uniao_esferas(cs, rs, e))
  end
```

Solução do Exercício 6.4.14

```
cubo_furado(p, lado) =
  for i in 0:2
    for j in 0:2
      for k in 0:2
        if !(i == j == 1 || i == k == 1 || j == k == 1)
          cubo_vertice(p+vxyz(i*lado, j*lado, k*lado), lado)
        end
      end
    end
  end

cubo_vertice(p, lado) =
  box(p, p+vxyz(lado, lado, lado))
```

Solução do Exercício 6.4.15

```
esponja_menger(p, lado, nivel) =
  if nivel == 0
    cubo_vertice(p, lado)
  else
    let lado = lado/3.0
      for i in 0:2
        for j in 0:2
          for k in 0:2
            if !(i == j == 1 || i == k == 1 || j == k == 1)
              esponja_menger(p+vxyz(i*lado, j*lado, k*lado), lado, nivel-1)
            end
          end
        end
      end
    end
  end
```

Solução do Exercício 6.4.16

```
raios_cilindro(p, r, a, da, rc, n) =
  union([cylinder(p, rc, p+vpol(r, a+i*da)) for i in 0:n-1])

cobertura_arcos_romanos(p, r, e, n) =
  let da = 2*pi/n,
      lc = r*cos(da/2),
      rc = r*sin(da/2)
    subtraction(raios_cilindro(p, lc, 0, da, rc, n),
                raios_cilindro(p, r, 0, da, rc-e, n),
                cylinder(p, r, p+vz(-rc)))
  end
```

Solução do Exercício 6.5.1

```
octaedro_estrelado(p, l) =
  union(tetraedro(p+vxyz(l/-2, l/-2, l/-2),
                  p+vxyz(l/2, l/2, l/-2),
                  p+vxyz(l/-2, l/2, l/2),
                  p+vxyz(l/2, l/-2, l/2)),
        tetraedro(p+vxyz(l/2, l/-2, l/-2),
                  p+vxyz(l/-2, l/2, l/-2),
                  p+vxyz(l/-2, l/-2, l/2),
                  p+vxyz(l/2, l/2, l/2)))
```

Solução do Exercício 6.5.2

```
meio(a, b) =
  (a+b)/2.0

meio_xyz(p0, p1) =
  xyz(meio(p0.x, p1.x), meio(p0.y, p1.y), meio(p0.z, p1.z))

sierpinski(p0, p1, p2, p3, n) =
  if n == 0
    tetraedro(p0, p1, p2, p3)
  else
    let p0p1 = meio_xyz(p0, p1),
        p0p2 = meio_xyz(p0, p2),
        p0p3 = meio_xyz(p0, p3),
        p1p2 = meio_xyz(p1, p2),
        p1p3 = meio_xyz(p1, p3),
        p2p3 = meio_xyz(p2, p3)
      sierpinski(p0, p0p1, p0p2, p0p3, n-1)
      sierpinski(p0p1, p1, p1p2, p1p3, n-1)
      sierpinski(p0p2, p1p2, p2, p2p3, n-1)
      sierpinski(p0p3, p1p3, p2p3, p3, n-1)
    end
  end

sierpinski(xyz(-1, -1, -1),
           xyz(+1, +1, -1),
           xyz(+1, -1, +1),
           xyz(-1, +1, +1), 5)
```

Solução do Exercício 6.5.3

Solução do Exercício 6.6.1

Solução do Exercício 6.6.2

```
laje(p, a, omega, fi, lx, dx, ly, lz) =
  let pontos = pontos_sinusoide(p, a, omega, fi, 0, lx, dx)
    extrusion(surface(spline(pontos),
                      line(pontos[1],
                           p + vxy(0, ly),
                           p + vxy(lx, ly),
                           pontos[end])),
              lz)
  end
```

Solução do Exercício 6.6.3

```
corrimao(p, a, omega, fi, lx, dx, l_corrimao, a_corrimao) =
  parede_sinusoidal(p, a, omega, fi, 0, lx, dx, l_corrimao, a_corrimao)
```

Solução do Exercício 6.6.4

```
prumos(p, a, omega, fi, lx, dx, altura, raio) =
  for ponto in pontos_sinusoide(p, a, omega, fi, 0, lx, dx)
    cylinder(ponto, raio, ponto+vxyz(0, 0, altura))
  end
```

Solução do Exercício 6.6.5

```
guarda(p, a, omega, fi, lx, dx,
       a_guarda, l_corrimao, a_corrimao, d_prumos) =
  begin
    corrimao(p + vxyz(0, l_corrimao/-2.0, a_guarda),
             a, omega, fi, lx, d_prumos, l_corrimao, a_corrimao)
    prumos(p, a, omega, fi, lx, d_prumos, a_guarda, l_corrimao/3.0)
  end
```

Solução do Exercício 6.6.6

```
piso(p, a, omega, fi, lx, dx, ly,
     a_laje, a_guarda, l_corrimao, a_corrimao, d_prumos) =
  begin
    laje(p, a, omega, fi, lx, d_prumos, ly, a_laje)
    guarda(p + vxyz(0, l_corrimao, a_laje),
           a, omega, fi, lx, dx, a_guarda, l_corrimao, a_corrimao, d_prumos)
  end
```

Solução do Exercício 6.6.7

```
predio(p, a, omega, fi, lx, dx, ly,
       a_laje, a_guarda, l_corrimao, a_corrimao,
       d_prumos, a_andar, n_andares) =
  if n_andares == 0
      nothing
  else
      piso(p, a, omega, fi, lx, dx, ly,
           a_laje, a_guarda, l_corrimao, a_corrimao, d_prumos)
      predio(p + vxyz(0, 0, a_andar),
             a, omega, fi, lx, dx, ly,
             a_laje, a_guarda,
             l_corrimao, a_corrimao,
             d_prumos,
             a_andar,
             n_andares - 1)
  end
```

Solução do Exercício 6.6.8

```
predio(p, a, omega, fi, lx, dx, ly,
       a_laje, a_guarda, l_corrimao, a_corrimao,
       d_prumos, a_andar, n_andares, dfi) =
  if n_andares == 0
    nothing
  else
    piso(p, a, omega, fi, lx, dx, ly,
         a_laje, a_guarda, l_corrimao, a_corrimao, d_prumos)
    predio(p + vxyz(0, 0, a_andar),
           a, omega, fi + dfi, lx, dx, ly,
           a_laje, a_guarda,
           l_corrimao, a_corrimao,
           d_prumos,
           a_andar,
           n_andares - 1,
           dfi)
  end
```

Solução do Exercício 6.6.9 As imagens foram geradas com `d_fi` a variar ao longo da sequência $[0, \pi, \pi/2, \pi/4]$.

Solução do Exercício 6.6.10

```
corrimao(p, a, omega, fi, lx, dx, l_corrimao, a_corrimao) =
  sweep(spline(pontos_sinusoide(p, a, omega, fi, 0, lx, dx)),
        rectangle(xy(-l_corrimao/2.0, -a_corrimao/2.0), l_corrimao, a_corrimao))
```

Solução do Exercício 6.6.11

```
parede_pontos(espessura, altura, pontos_curva) =
  let curva = spline(pontos_curva),
      seccao = surface_rectangle(xy(-espessura/2, -altura/2), espessura, altura)
    sweep(curva, seccao)
  end
```

Solução do Exercício 6.8.1

```
sinusoide(a, omega, fi, x) =
  a*sin(omega*x + fi)

pontos_sinusoide(p, a, omega, fi, x0, x1, dx) =
  if x0 > x1
    []
  else
    [p + vxy(x0, sinusoide(a, omega, fi, x0)),
     pontos_sinusoide(p, a, omega, fi, x0 + dx, x1, dx)...]
  end

tubo_sinusoidal(p, r, a, omega, fi, lx, dx) =
  revolve(spline(pontos_sinusoide(p + vxy(0, -r), a, omega, fi, 0, lx, dx)),
          p,
          vx(1))

tubo_sinusoidal(xyz(0, 0, 0), 10, 1, 1, 0, 60, 0.2)
```

Solução do Exercício 6.8.2 Para que os arcos de circunferência possam interligar-se suavemente, é necessário que exista uma relação entre os raios r_0, r_1 e r_2. É facil ver que

$$(r_2 - r_1)\cos\alpha + r_0 = r_2$$

Por outro lado, a altura h do óvulo é dada por

$$h = r_0 + (r_2 - r_1)\sin\alpha + r_1$$

Uma vez que a função não recebe nem o raio r_2 nem o ângulo α, temos de os deduzir. Para isso, vamos resolver a primeira equação em ordem a r_2

$$r_2 = \frac{r_0 - r_1\cos\alpha}{1 - \cos\alpha}$$

e vamos substituir na segunda equação

$$h = r_0 + (\frac{r_0 - r_1\cos\alpha}{1 - \cos\alpha} - r_1)\sin\alpha + r_1$$

Simplificando, obtemos

$$h = r_0 + r_1 + (r_0 - r_1)\frac{\sin\alpha}{1 - \cos\alpha}$$

Empregando agora a igualdade trigonométrica

$$\tan\frac{\alpha}{2} = \frac{1 - \cos\alpha}{\sin\alpha}$$

temos

$$h = r_0 + r_1 + (r_0 - r_1)\frac{1}{\tan\frac{\alpha}{2}}$$

ou seja,

$$\alpha = 2\tan^{-1}\frac{r_0 - r_1}{h - r_0 - r_1}$$

Estamos agora em condições de definir a função pretendida:

```
ovo(p, r0, r1, h) =
  let alfa = 2*atan2(r0-r1, h-r0-r1),
      r2 = (r0-r1*cos(alfa))/(1-cos(alfa))
    revolve(union(arc(p, r0, -pi/2, pi/2),
                  arc(p+vx(r0-r2), r2, 0, alfa),
                  arc(p+vy((r2-r1)*sin(alfa)), r1, alfa, (pi-alfa-alfa)/2)),
            p,
            vy())
  end
```

Solução do Exercício 6.8.3

```
perfil_barril(p, r0, r1, h, e) =
  surface(line(p + vcyl(r0, 0, 0),
               p,
               p + vcyl(0, 0, e),
               p + vcyl(r0 - e, 0, e)),
          spline(p + vcyl(r0 - e, 0, e),
                 p + vcyl(r1 - e, 0, h/2),
                 p + vcyl(r0 - e, 0, h - e)),
          line(p + vcyl(r0 - e, 0, h - e),
               p + vcyl(r0 - e, 0, h),
               p + vcyl(r0, 0, h)),
          spline(p + vcyl(r0, 0, h),
                 p + vcyl(r1, 0, h/2),
                 p + vcyl(r0, 0, 0)))
```

Solução do Exercício 6.8.4

```
barril(p, r0, r1, h, e) =
  revolve(perfil_barril(p, r0, r1, h, e), p)
```

Solução do Exercício 6.8.5

```
tabua_barril(p, r0, r1, h, e, alfa, d_alfa) =
  revolve(perfil_barril(p, r0, r1, h, e),
          p,
          vz(1),
          alfa,
          d_alfa)
```

Solução do Exercício 6.8.6

```
tabuas_barril(p, r0, r1, h, e, n, s) =
  let delta_alfa = 2*pi/n
    for alfa in division(0, 2*pi, n, false)
      tabua_barril(p, r0, r1, h, e, alfa, delta_alfa-s)
    end
  end
```

Solução do Exercício 6.9.1

```
for n in 5:5:25
  loft([surface(
          closed_spline(
            pontos_circulo_raio_aleatorio(xyz(n/5.0, 0, 0), 0.3, 0.5, n))),
        surface(
          closed_spline(
            pontos_circulo_raio_aleatorio(xyz(n/5.0, 0, 1), 0.2, 0.4, n)))])
end
```

Solução do Exercício 7.6.1

```
quarto_toro(re, ri) =
  slice(slice(torus(u0(), re, ri),
              u0(), vx()),
        u0(), vy())

meio_cilindro(ri, l) =
  cylinder(u0(), ri, xyz(l/2.0, 0, 0))

quarto_elo(re, ri, l) =
  union(move(quarto_toro(re, ri), -vx()*l/2.0),
        move(meio_cilindro(ri, l), -vy(re)-vx(l/2.0)))

elo(re, ri, l) =
  union_mirror(union_mirror(quarto_elo(re, ri, l),
                            u0(), vy()),
               u0(), vx())
```

Solução do Exercício 7.6.2

```
corrente(n, re, ri, l) =
  if n == 1
    elo(re, ri, l)
  else
    union(elo(re, ri, l),
          rotate(move(corrente(n-1, re, ri, l),
                      vx()*(l+2*(re-ri))),
                 pi/2,
                 u0(),
                 vx()))
  end
```

Solução do Exercício 7.6.3

```
elo_rodado(n, re, ri, l) =
  if n%2 == 0
    elo(re, ri, l)
  else
    rotate(elo(re, ri, l), pi/2, u0(), vx())
  end

corrente_fechada(n, r, a, da, re, ri, l) =
  if n == 0
    nothing
  else
    move(rotate(elo_rodado(n, re, ri, l),
                a+pi/2,
                u0(),
                vz()),
         vcyl(r, a, 0))
    corrente_fechada(n-1, r, a+da, da, re, ri, l)
  end
```

Solução do Exercício 8.4.1

```
laje(p, f, x0, x1, dx, ly, lz) =
  let pontos = pontos_funcao(p + vxy(-x0, -ly), f, x0, x1, dx),
      p0 = pontos[1],
      p1 = pontos[end]
    extrusion(surface(spline(pontos), line(p0, p, p + vx(x1 - x0), p1)), lz)
  end

corrimao(p, f, x0, x1, dx, ly, l_corrimao, a_corrimao) =
  sweep(spline(pontos_funcao(p + vxy(-x0, -ly), f, x0, x1, dx)),
        surface_rectangle(xy(-l_corrimao/2.0, -a_corrimao/2.0), l_corrimao, a_corrimao))


prumos(p, f, x0, x1, dx, ly, altura, raio) =
  for ponto in pontos_funcao(p + vxy(-x0, -ly), f, x0, x1, dx)
    cylinder(ponto, raio, ponto + vz(altura))
  end


guarda(p, f, x0, x1, dx, ly, a_guarda, l_corrimao, a_corrimao, d_prumos) =
  begin
    corrimao(p + vz(a_guarda), f, x0, x1, d_prumos, ly, l_corrimao, a_corrimao)
    prumos(p, f, x0, x1, d_prumos, ly, a_guarda, l_corrimao/2.0)
  end


varanda(p, f, x0, x1, dx, ly, a_laje, a_guarda, l_corrimao, a_corrimao, d_prumos) =
  begin
    laje(p, f, x0, x1, d_prumos, ly, a_laje)
    guarda(p + vxyz(0, l_corrimao, a_laje), f, x0, x1, dx, ly, a_guarda, l_corrimao, a_corrimao, d_prumos)
  end
```

Solução do Exercício 8.4.2

```
sinusoide_limitada(a, omega, fi, x) =
  a*max(-0.6, min(0.6, sin(omega*x + fi)))

serra(a, b, c, x) =
  abs(a + b*(x - c))

varanda(xyz(0, 0, 0),
        x -> serra(0, 0.3, 3*pi, x),
        0, 4*pi, 0.5,
        4, 0.2, 1, 0.04, 0.02, 0.4)
varanda(xyz(8, 8, 0),
        x -> oscilatorio_amortecido(-2, 0.1, 2, x),
        0, 4*pi, 0.2,
        4, 0.2, 1, 0.04, 0.02, 0.2)
varanda(xyz(16, 16, 0),
        x -> sinusoide_limitada(1.0, 1.0, 0, x),
        0, 4*pi, 0.2,
        4, 0.2, 1, 0.04, 0.02, 0.4)
```

Solução do Exercício 8.4.3

```
roundv(v) =
  vxyz(round(v.x), round(v.y), round(v.z))


caminho_de_tubos(p0, v0, l, n) =
  if n == 0
    nothing
  else
    let v1 = roundv(vsph(1, random_range(0, 4)*pi/2, random_range(0, 3)*pi/2))
      if v0 == -v1
        caminho_de_tubos(p0, v0, l, n)
      else
        let p1 = p0 + v1*random_range(0.2, 1.0)*l
          cylinder(p0, 0.02*l, p1)
          sphere(p1, 0.04*l)
          caminho_de_tubos(p1, v1, l, n - 1)
        end
      end
    end
  end
```

Solução do Exercício 8.4.4

```
forma_de_tubos(p0, v0, l, n, aceitavel) =
  if n == 0
    nothing
  else
    let v1 = roundc(sph(1, random_range(0, 4)*pi/2, random_range(0, 3)*pi/2))
      if v0 == -v1
        forma_de_tubos(p0, v0, l, n, aceitavel)
      else
        let p1 = p0 + v1*random_range(0.2, 1.0)*l
          if aceitavel(p1)
            cylinder(p0, 0.02*l, p1)
            sphere(p1, 0.04*l)
            forma_de_tubos(p1, v1, l, n - 1, aceitavel)
          else
            forma_de_tubos(p0, v0, l, n, aceitavel)
          end
        end
      end
    end
  end

cubo_de_tubos(p, r, l, n) =
  let no_cubo(p1) =
        let v = p1 - p
          abs(v.x) < r && abs(v.y) < r && abs(v.z) < r
        end
    forma_de_tubos(p, xyz(1, 0, 0), l, n, no_cubo)
  end

cilindro_de_tubos(p, r, l, n) =
  let no_cilindro(p1) =
        let v = p1 - p
          cyl_rho(v) < r && abs(cyl_z(v)) < r
        end
    forma_de_tubos(p, xyz(1, 0, 0), l, n, no_cilindro)
  end

esfera_de_tubos(p, r, l, n) =
  forma_de_tubos(p, xyz(1, 0, 0), l, n, p1 -> distance(p, p1) < r)
```

Solução do Exercício 8.5.1

```
acumulatorio(combina, f, inicial, a, suc, b) =
  if a > b
    inicial
  else
    combina(f(a), acumulatorio(combina, f, inicial, suc(a), suc, b))
  end

somatorio(f, a, b) =
  acumulatorio((a, b) -> a + b, f, 0, a, i -> i + 1, b)

produtorio(f, a, b) =
  acumulatorio((a, b) -> a*b, f, 1, a, i -> i + 1, b)
```

Solução do Exercício 8.5.2

```
aproxima_pi(n) =
  acumulatorio((a, b) -> a + b,
               x -> 8.0/(x*(x + 2)),
               0,
               1,
               x -> x + 4, n)

julia> aproxima_pi(2000)
3.140592653839793
```

Solução do Exercício 8.5.3

```
sucessao(a, b, f) =
  if a >= b
    []
  else
    [a, sucessao(f(a), b, f)...]
  end
```

Solução do Exercício 8.5.4

```
enumera(a, b, i) =
  sucessao(a, b, ai -> ai + i)
```

Solução do Exercício 8.8.1

```
superficie_interpolacao(pontos) =
  loft(mapeia(spline, pontos))
```

Solução do Exercício 8.9.1

```
quantos(f, lst1, lst2) =
  if lst1 == [] || lst2 == []
    0
  elseif f(lst1[1], lst2[1])
    1 + quantos(f, lst1[2:end], lst2[2:end])
  else
    quantos(f, lst1[2:end], lst2[2:end])
  end
```

Solução do Exercício 8.9.2

```
ponto_no_poligono(p, vs) =
  let q = xy(reduce(max, map(cx, vs)) + 1, p.y),
      l = quantos((vi, vj) -> interseccao_segmentos(p, q, vi, vj),
                  vs,
                  [vs[2:end]..., vs[1]])
    l % 2 == 1
  end
```

Solução do Exercício 8.9.3

```
media(lista) =
  reduce((a, b) -> a + b, lista)/length(lista)

cria_edificios(pontos, poligonos) =
  for poligono in poligonos
    let pts = pontos_no_poligono(pontos, poligono),
        n_pts = length(pts)
      if n_pts == 0
        nothing
      elseif n_pts == 1
        cria_edificio(poligono, pts[1].z)
      else
        cria_edificio(poligono, media(mapeia(cz, pts)))
      end
    end
  end
```

Solução do Exercício 9.4.1

```
espiral_arquimedes(p, alfa, t0, t1, n) =
  map(t -> p + vpol(alfa*t, t), division(t0, t1, n))
```

Solução do Exercício 9.4.2

```
parede_curva(espessura, altura, curva) =
  let seccao = surface_rectangle(xy(0, 0), xy(espessura, altura))
    sweep(curva, seccao)
  end

parede_espiral_arquimedes(espessura, altura, p, alfa, t0, t1, n) =
  parede_curva(espessura,
               altura,
               spline(espiral_arquimedes(p, alfa, t0, t1, n)))

parede_espiral_arquimedes(4, 10.0, xy(0, 0), 2, 0, 16*pi, 150)
```

Solução do Exercício 9.4.3

```
cilindros_espiral_arquimedes(r, h, p, alfa, t0, t1, n) =
  [cylinder(p + vpol(alfa*t, t), r, h)
   for t in division(t0, t1, n)]
```

Solução do Exercício 9.4.4 Para desenharmos separadamente a parte positiva e negativa da curva, vamos parametrizá-la, definindo

```
meia_lemniscata_gerono(a, x0, x1, n) =
  map_division(x -> xy(x, a*sqrt(x^2 - x^4)), x0, x1, n)
```

Agora, é possível definir:

```
grafico_lemniscata_gerono(a, x0, x1, n) =
  closed_spline([meia_lemniscata_gerono(+a, x0, x1, n)[2:end]...,
                 meia_lemniscata_gerono(-a, x1, x0, n)[2:end]...])

grafico_lemniscata_gerono(1.0, -1.0, 1.0, 200)
```

Solução do Exercício 9.4.5

Solução do Exercício 9.4.6

```
tanque_circular(p, raio, espessura, altura) =
  parede_curva(espessura, altura, circle(p, raio))

tanques_circulares(p, raio, espessura, altura, a, b, n, pts) =
  for t in division(-pi, pi, pts, false)
    tanque_circular(superelipse(p, a, b, n, t), raio, espessura, altura)
  end
```

Solução do Exercício 9.4.7

```
tanque_sergels(p) =
  begin
    tanque_superelipse(p, 0.4, 1.0, 20, 22, 2.5, 100)
    tanques_circulares(p, 1.0, 0.2, 0.8, 18, 20, 2.5, 32)
    tanques_circulares(p, 1.0, 0.2, 0.8, 15, 17, 2.5, 32)
  end
```

Solução do Exercício 9.4.8

```
tanque_sergels(p) =
  begin
    tanque_superelipse(p, 0.4, 1.0, 22, 22, 2, 100)
    tanques_circulares(p, 1.0, 0.2, 0.8, 20, 20, 2, 32)
    tanques_circulares(p, 1.0, 0.2, 0.8, 17, 17, 2, 32)
  end
```

Solução do Exercício 9.4.9

```
tanque_sergels(p) =
  begin
    tanque_superelipse(p, 0.4, 1.0, 20, 22, 1.3, 100)
    tanques_circulares(p, 1.0, 0.2, 0.8, 18, 20, 1.3, 20)
    tanques_circulares(p, 0.8, 0.2, 0.8, 15, 17, 1.3, 20)
    tanques_circulares(p, 0.6, 0.2, 0.8, 12, 14, 1.3, 20)
    tanques_circulares(p, 0.4, 0.2, 0.8, 10, 12, 1.3, 20)
  end
```

Solução do Exercício 9.4.10

```
tanque_superelipse(p, espessura, altura, a, b, n, pts) =
  for s in division(-pi, pi, 4, false)
    parede_curva(espessura,
                 altura,
                 spline([superelipse(p, a, b, n, t)
                         for t in division(s, s + pi/2, pts/4, true)]))
  end

tanque_sergels(p) =
  begin
    tanque_superelipse(p, 0.4, 1.0, 22, 22, 0.7, 100)
    tanques_circulares(p, 1.0, 0.2, 0.8, 17, 17, 0.7, 8)
    tanques_circulares(p, 1.0, 0.2, 0.8, 12, 12, 0.7, 12)
  end
```

Solução do Exercício 9.6.1

```
map_division_adapt(f, t0, t1, dt, epsilon) =
  let p0 = f(t0),
      p1 = f(t1),
      tm = (t0 + t1)/2.0,
      pm = f(tm)
    if t1 - t0 < dt && aproximadamente_colineares(p0, pm, p1, epsilon)
      [p0, pm, p1]
    else
      [map_division_adapt(f, t0, tm, dt, epsilon)...,
       map_division_adapt(f, tm, t1, dt, epsilon)[2:end]...]
    end
  end
```

Solução do Exercício 9.6.2

```
map_division_adapt(f, t0, t1, dt, epsilon) =
  let loop(t0, t1, p0, p1, d01, pts) =
        let tm = (t0 + t1)/2.0,
            pm = f(tm),
            d0m = distance(p0, pm),
            dm1 = distance(pm, p1)
          if t1 - t0 < dt && area_triangulo(d0m, dm1, d01) < epsilon
            [pm, pts...]
          else
            loop(t0, tm, p0, pm, d0m, [pm, loop(tm, t1, pm, p1, dm1, pts)...])
          end
        end
      endp0 = f(t0)
      p1 = f(t1)
    [p0, loop(t0, t1, p0, p1, distance(p0, p1), [p1])...]
  end
```

Solução do Exercício 9.8.1

```
rede_moebius(v, n, mn, m, mm) =
  begin
    for mi in v/m/4:v/m:v + v/m/2
      surface_grid(faixa_moebius(0, 4*pi, mn, mi, mi + v/m/2, mm))
    end
    for mi in v/m/4 + v/m/2:v/m:v
      for ni in 0:4*pi/n:4*pi
        surface_grid(faixa_moebius(ni, ni + 2*pi/n, mn/n/2, mi, mi + v/m/2, mm))
      end
    end
  end

rede_moebius(0.4, 80, 160, 5, 4)
```

Solução do Exercício 9.8.2

```
surface_grid(
  map_division((u, v) -> xyz(u, v, sin(u*v)),
               -pi, pi, 200,
               -pi, pi, 200))
```

Solução do Exercício 9.8.3

Solução do Exercício 9.8.4

```
breather(p, a, u0, u1, m, v0, v1, n) =
  let r = 1 - a^2,
      w = sqrt(r)
      f(u, v) =
        let sinv = sin(v),
            cosv = cos(v),
            sinwv = sin(w*v),
            coswv = cos(w*v),
            sinhau = sinh(a*u),
            coshau = cosh(a*u),
            d = a*((w*coshau)^2 + (a*sinwv)^2)
          p+vxyz(-u + 2*r*coshau*sinhau/d,
                 2*w*coshau*(-(w*cosv*coswv) - sinv*sinwv)/d,
                 2*w*coshau*(-(w*sinv*coswv) + cosv*sinwv)/d)
        end
    map_division(f, u0, u1, m, v0, v1, n)
  end
```

Solução do Exercício 9.8.5

```
elipsoide(a, b, c, n) =
  surface_grid(
    map_division((fi, psi) -> xyz(a*sin(psi)*cos(fi),
                                  b*sin(psi)*sin(fi),
                                  c*cos(psi)),
                 0, 2*pi, n,
                 0, pi, n))
```

Solução do Exercício 9.9.1

```
prisma_poligonal(pts, dx) =
  loft_ruled([surface_polygon(pts), surface_polygon(map(pt -> pt + vx(dx), pts))])
```

Solução do Exercício 9.9.2

```
cobertura_ysios(pts0, pts1, pts2, pts3) =
  if length(pts0) == 1
    []
  else
    [prisma_poligonal([pts0[1], pts2[1], pts3[1], pts1[1]],
                      pts0[2].x - pts0[1].x),
     cobertura_ysios(pts0[2:end], pts1[2:end], pts2[2:end], pts3[2:end])...]
  end
```

Solução do Exercício 9.9.3

```
curva_ysios(p, a, b, c, d) =
  let f(t) =
        if -pi/2 <= t <= pi/2
          p + vxyz(t, c*cos(t), d*cos(t))
        else
          p + vxyz(t, a*cos(t), b*cos(t))
        end
    map_division(f, -7*pi, 7*pi, 100)
  end
```

Solução do Exercício 9.9.4

```
surface_grid(
  map_division(
    (x, z) -> xyz(14*pi/100/2, -17, 0.2) +
              vxyz(x,
                   -pi/2 <= x <= pi/2 ?
                     (z + 3)*-0.8*cos(x) :
                     -cos(x),
                   -pi/2 <= x <= pi/2 ?
                     z*(1 + 0.4*cos(x)) :
                     z*(1 + 0.2*cos(x))),
    -7*pi, 7*pi, 100,
    0, 5, 10))

cobertura_ysios(
  curva_ysios(xyz(0, -20, 5.0), 0.0, 1.0, -4.0, 1.0),
  curva_ysios(xyz(0, -20, 5.7), 0.0, 1.0, -6.0, 2.8),
  curva_ysios(xyz(0, 20, 5.0), 0.0, -1.0, 0.0, -1.0),
  curva_ysios(xyz(0, 20, 5.7), 0.0, -1.0, 0.0, -1.0))
```

Solução do Exercício 9.9.5 O toro é definido parametricamente por:

$$\begin{aligned} x &= \cos v(r_0 + r_1 \cos u) \\ y &= \sin v(r_0 + r_1 \cos u) \\ z &= r_1 \sin u \end{aligned}$$

$$0 \leq u \leq 2\pi; \quad 0 \leq v \leq 2\pi;$$

A definição da função fica então:

```
toro(p, r0, r1, m, n) =
  surface_grid(
    map_division((fi, psi) -> p+vpol(r0, fi)+vsph(r1, fi, psi),
                 0, 2pi, m, false,
                 0, 2pi, n, false),
    true,
    true)

toro(u0(), 10, 2, 50, 20)
```

Solução do Exercício 9.9.6

```
toro_esferas(p, r0, r1, m, n) =
  for u in division(0, 2*pi, m)
    for v in division(0, 2*pi, n)
      sphere(p + vcyl(r0 + r1*cos(v), u, r1*sin(v)), 0.1)
    end
  end
```

Solução do Exercício 9.9.7

```
maca(p)=
  surface_grid(
    map_division((u,v) -> p+vxyz(cos(u)*(4+3.8*cos(v)),
                                 sin(u)*(4+3.8*cos(v)),
                                 (cos(v)+sin(v)-1)*(1+sin(v))*log(1-pi*v/10)+7.5*sin(v)),
                 0, 2pi, 50,
                 -pi, pi, 50))
```

Solução do Exercício 9.9.8

```
rosa(x1, phi) =
  let phi = pi/2*exp(-(phi/8/pi)),
      x = 1 - 1/2*(5/4*(1 - divmod(3.6*phi, 2*pi)[2]/pi) ^ 2 - 1/4) ^ 2,
      y1 = 1.9565284531299512*x1^2*(1.2768869870150188*x1 - 1)^2*sin(phi),
      r = x*(x1*sin(phi) + y1*cos(phi))
    xyz(r*sin(phi), r*cos(phi), x*(x1*cos(phi) - y1*sin(phi)))
  end

surface_grid(map_division(rosa, 1e-06, 1, 100, -2*pi, 15*pi, 1000))
```

Solução do Exercício 9.11.1

```
cruz_esfera_quad(p0, p1, p2, p3) =
  let r = min(distance(p0, p1), distance(p0, p3))/15,
      p = centro_quadrangulo(p0, p1, p2, p3)
    cylinder(p, r, p0)
    cylinder(p, r, p1)
    cylinder(p, r, p2)
    cylinder(p, r, p3)
    sphere(p, 3*r)
  end

cubo_quad(p0, p1, p2, p3) =
  let p = centro_quadrangulo(p0, p1, p2, p3),
      r = min(distance(p0, p1), distance(p0, p3))
    box(p + vxyz(-(r/2), -(r/2), -(r/2)), r, r, r)
  end

cone_quad(p0, p1, p2, p3) =
  let p = centro_quadrangulo(p0, p1, p2, p3),
      n = normal_quadrangulo(p0, p1, p2, p3),
      d = min(distance(p0, p1), distance(p0, p3))
    cone(p, d/2, p + n*d)
  end

calota_quad(p0, p1, p2, p3) =
  let d = min(distance(p0, p1), distance(p0, p3)),
      h = d/4.0,
      p = centro_quadrangulo(p0, p1, p2, p3),
      n = normal_quadrangulo(p0, p1, p2, p3)
    calota_esferica(p, p + n*h, d)
  end

cylinder_base_quad(p0, p1, p2, p3) =
  let d = 0.7*min(distance(p0, p1), distance(p0, p3)),
      p = centro_quadrangulo(p0, p1, p2, p3)
    cylinder(p + vz(0.3), d/2, p + vz(-0.3 - p.z))
  end

paralelipipedo_quad(p0, p1, p2, p3) =
  let d = min(distance(p0, p1), distance(p0, p3)),
      h = random_range(d/10.0, 2*d)
    box(loc_from_o_vx_vy(p0, p1 - p0, p3 - p0), d, d, h)
  end
```

Solução do Exercício 9.11.2

```
trama_tecido_quad(p0, p1, p2, p3) =
  let n02 = p2 - p0,
      n13 = p3 - p1,
      c = centro_quadrangulo(p0, p1, p2, p3),
      n = normal_quadrangulo(p0, p1, p2, p3),
      r = min(distance(p0, p1), distance(p0, p3))/3.0
    loft(map(surface,
             [circulo_normal(p0, n02, r),
              circulo_normal(c + n*r, n02, r),
              circulo_normal(p2, n02, r)]))
    loft(map(surface,
             [circulo_normal(p1, n13, r),
              circulo_normal(c + n*-r, n13, r),
              circulo_normal(p3, n13, r)]))
  end


itera_quadrangulos(trama_tecido_quad, malha_teste())
```

Solução do Exercício 9.11.3

```
oscila_curvas(ptss, r) =
  [oscila_faixa(ptss[1], ptss[2], r),
   (length(ptss) == 2 ?
   [] :
   oscila_curvas(ptss[2:end], -r))...]

oscila_faixa(pts0, pts1, r) =
  [oscila_centro(pts0[1], pts1[1], pts1[2], pts0[2], r),
   (length(pts0) == 2 ?
   [] :
   oscila_faixa(pts0[2:end], pts1[2:end], -r))...]

oscila_centro(p0, p1, p2, p3, r) =
  centro_quadrangulo(p0, p1, p2, p3) + normal_quadrangulo(p0, p1, p2, p3)*r

tecido(malha) =
  let p0 = malha[1][1],
      p1 = malha[2][1],
      p2 = malha[2][2],
      p3 = malha[1][2:end][1],
      d = min(distance(p0, p1), distance(p0, p3)),
      r = d/6.0
    map(curva -> sweep(spline(curva),
                       surface(circle(xyz(0, 0, 0), r))),
        [oscila_curvas(malha, r)...,
         oscila_curvas(matriz_transposta(malha), r)...])
  end
```

Solução do Exercício 9.11.4 O "defeito" é a falta de uma fileira de barras na parte de baixo da treliça. A explicação está no facto de a treliça ser, na realidade, uma fita comprida em que extremidades estão coincidentes. A falta da fileira de barras é, na verdade, o início e fim da fita.

Solução do Exercício 9.11.5

```
pontos_ourico(p, r, a, h, f, n) =
  map_division(
    (fi, psi) -> p+vsph((r+a*sin(f*fi)*sin(psi))*(h+sin(psi)),
                        fi,
                        psi),
    0, 2*pi, n,
    0, 0.7*pi, n/2)

agulha_quad(p0, p1, p2, p3) =
  let p = centro_quadrangulo(p0, p1, p2, p3),
      n = normal_quadrangulo(p0, p1, p2, p3)
    cone(p, 0.2, p + n*-5)
  end

ourico(p, r, a, h, f, n, n_agulhas) =
  begin
    surface_grid(pontos_ourico(p, r, a, h, f, n))
    with(current_layer, create_layer("Black")) do
      itera_quadrangulos(agulha_quad, pontos_ourico(p, r, a, h, f, n_agulhas))
    end
  end
```

Índice