Introdução à linguagem de programação

RAIMUNDO NONATO DINIZ COSTA FILHO
MARINA DA SILVA MIRANDA
MILLENA MARINHO ROCHA
ANDRÉ SANTOS NASCIMENTO

EDUFMA

Introdução à linguagem de programação

julia

RAIMUNDO NONATO DINIZ COSTA FILHO
MARINA DA SILVA MIRANDA
MILLENA MARINHO ROCHA
ANDRÉ SANTOS NASCIMENTO

Introdução a linguagem Julia

São Luís
EDUFMA
2023

Projeto gráfico, Diagramação e Capa: Marina da Silva Miranda e Millena Marinho Rocha.

Dados Internacionais de Catalogação na Publicação (CIP)

Introdução a linguagem de programação Julia [recurso eletrônico]./Raimundo Nonato Diniz Costa Filho... [et al.].— São Luís:EDUFMA, 2023.

108p.:il.

ISBN 978-65-5363-188-5
Modo de acesso: world wide web.

1. Linguagem de programação - Julia. 2. Linguagem de programação - Variáveis. 3.Ambiente de programação-Jupyter.I. Costa, Raimundo Nonato Diniz. II.Miranda, Marina da Silva. III. Rocha, Millena Marinho. IV. Nascimento, André Santos.

CDD 005.13

Ficha catalográfica elaborada pela Diretoria Integrada de Bibliotecas- DIB/UFMA
Bibliotecária: Gracelynne Oliveira Santos -
CRB 13/520

| EDUFMA | EDITORA DA UNIVERSIDADE FEDERAL DO MARANHÃO
Av. dos Portugueses 1966 | Vila Bacanga
CEP: 65080-805 | São Luís | MA | Brasil
Telefone: (98) 3272-8157
www.edufma.ufma.br | edufma.sce@ufma.br

“O verdadeiro discípulo é aquele que supera o mestre”

(Aristóteles)

Dedicado aos nossos pais

Raimundo e Maria Jucileide

Adão e Arni

José Nilson e Aurileth

Antônio e Francisca

AGRADECIMENTOS

A todos aqueles que contribuíram direta ou indiretamente para realização deste livro.

A Universidade Federal do Maranhão (UFMA) pelas bolsas do Projeto de Ensino do Foco Acadêmico.

SUMÁRIO

SINOPSE

O livro aborda o básico de linguagem Julia e foi dividido em 10 capítulos. Resumidamente, os capítulos são:

- CAPÍTULO 1: Apresenta o histórico da linguagem Julia, ambiente de desenvolvimento, funções básicas de impressão em tela.
- CAPÍTULO 2: um capítulo dedicado aos tipos de variáveis.
- CAPÍTULO 3: refere-se aos comandos de decisão, a exemplo, if, if-else e if-else-if.
- CAPÍTULO 4: O referido capítulo é exclusivo para os laços.
- CAPÍTULO 5: As funções são apresentadas neste capítulo. Há vários exemplos com códigos no final deste capítulo.
- CAPÍTULO 6: Pacotes e a execução de códigos (ou scripts) são estudados neste capítulo.
- CAPÍTULO 7: Neste capítulo é apresentado algumas funções para manipulação de arquivos.
- CAPÍTULO 8: O capítulo 8 é dedicado aos gráficos. A linguagem Julia tem vários pacotes para traçar gráficos de diversos tipos.
- CAPÍTULO 9: As interfaces gráficas do usuário (GUI- *Graphic User Interface*) são estudadas no capítulo 9.
- CAPÍTULO 10: Os códigos implementados nos capítulos 1 a 9 são executados através do REPL (interface de linha de comando), entretanto para otimizar mais os códigos podemos utilizar os IDE (Ambientes de programação), logo o objetivo deste capítulo é apresentar o ambiente de programação Jupyter. Alguns exemplos de códigos em Julia são executados no Jupyter.

Espero que o estilo de escrita deste livro agrade a todos os leitores, pois procuramos fazer um livro que seja de fácil leitura, com códigos simples e textos bem detalhados. Caso os leitores encontrem erros, quiser fazer sugestões/comentários sobre o livro ou conversar sobre a linguagem Julia, nós, autores, ficaremos felizes em receber seus e-mails através de raimundo.diniz@ufma.br.

PREFÁCIO

Há quase duas décadas que eu, Raimundo Diniz, professor da Universidade Federal do Maranhão (UFMA/Balsas), implementei meu primeiro programa em linguagem C, o famoso "Olá Mundo" ou em inglês "Hello World!". O tempo foi passando e, com o incentivo e orientação do professor Leonardo Paucar (UFMA/São Luís), comecei a estudar outras linguagens de programação, como por exemplo, Matlab, Python, C# e Java. No ano de 2018 conheci a linguagem Julia e me atraiu muito pela sua simples sintaxe e alto desempenho. Logo, comecei a aplicar em problemas práticos da área de otimização e sistemas elétricos de potência. Paralelamente, a Julia foi apresentada aos meus alunos de graduação, André, Millena e Marina, através de um projeto de ensino da Universidade Federal do Maranhão (UFMA). Na função de Coordenador do projeto de ensino, observei que existiam poucos materiais sobre a referida linguagem de programação em português e pensei em um livro nomeado de "Introdução a Linguagem Julia". Assim, com ajuda dos citados discentes de graduação, a ideia deste livro foi concretizada.

1. INTRODUÇÃO

1.1 A HISTÓRIA DA LINGUAGEM JULIA E SUAS CARACTERÍSTICAS

Um típico problema enfrentado pelos pesquisadores e programadores é encontrar uma única linguagem de programação que atenda todas as suas necessidades. Neste contexto, em 2009, Karpinski, juntamente com Viral Shah, Alan Edelman e Jeff Bezanson, co-fundadores da *startup Julia Computing*, desenvolveram a Julia, uma linguagem de programação de tipagem dinâmica de alto nível, ou seja, suas variáveis podem receber qualquer tipo de dado e sua sintaxe se aproxima mais da linguagem humana do que da linguagem de máquina. A primeira versão foi lançada em agosto de 2014 e desde então a comunidade Julia tem crescido bastante, assim como o desenvolvimento de seus pacotes, de tal forma que, atualmente (2022) a linguagem encontra-se na posição 25 no ranking do Índice TIOBE [1], [2], [3].

A referida linguagem foi pensada como uma linguagem de programação para computação científica suficientemente rápida, tal como as linguagens C e Fortran, mas igualmente fácil de aprender como o MATLAB®. É escrito em C, C++ e Scheme e a biblioteca padrão é escrita utilizando a própria linguagem Julia. Ademais, a Julia é inspirada em outras linguagens, como MATLAB®, Lisp, C, Fortran, Mathematica®, Python, R, Ruby, Lua, além de compartilhar muitas características de Dylan e Fortress [3].

De acordo com o manual da linguagem Julia [4], as principais vantagens são:

- Livre e *open source*;
- Tipos definidos pelo usuário são rápidos e compactos como tipos nativos;
- Ausência da necessidade de vetorizar códigos por desempenho:códigos não vetorizados são rápidos;
- Projetado para computação paralela e distribuída;
- Lightweight "*Green*" threading (encadeamento de execução);
- Sistemas de tipos não obstrutivos, mas poderoso;
- Conversão e promoção de tipos numéricos e outros de forma elegante e extensível;

- Suporte eficiente para Unicode, incluindo, mas não limitado ao UTF-8;
- Chamadas de funções em C de forma direta (sem necessidade de *wrappers* ou API especial);
- Capacidade semelhante à de uma poderosa Shell para gerenciar outros processos;
- Macros de forma parecida a Lisp e outras facilidades de metaprogramação;

1.2 AMBIENTE DE DESENVOLVIMENTO INTEGRADO

O Ambiente de Desenvolvimento Integrado (IDE, em inglês) é uma ferramenta utilizada com o intuito de combinar as atividades comuns de escrita de um software, sendo composta por: edição de código-fonte, construção de executáveis (compilador) e um depurador. Para ter acesso ao programa Julia, primeiramente deve-se fazer o download do instalador através do link: https://julialang.org/downloads/. Além de ser executável em uma máquina, é possível também executar os códigos no próprio navegador do usuário, como é retratado na Figura 1, o usuário pode acessar através do link https://julialang.org/learning/tryjulia/. Para a elaboração deste livro, os autores utilizaram a versão 1.6.3 no sistema operacional Windows 10. Após a execução do instalador do Julia é gerado um ícone no processo de instalação, que aparecerá em uma tela (terminal do Julia) como destaca a Figura 2, esse ícone é denominado de Julia REPL (*Read- Evaluate- Print- Loop*), excetuando o capítulo 10 deste livro, todos os códigos serão implementados neste ambiente de programação integrado.

Figura 1: Execução do Julia no navegador.

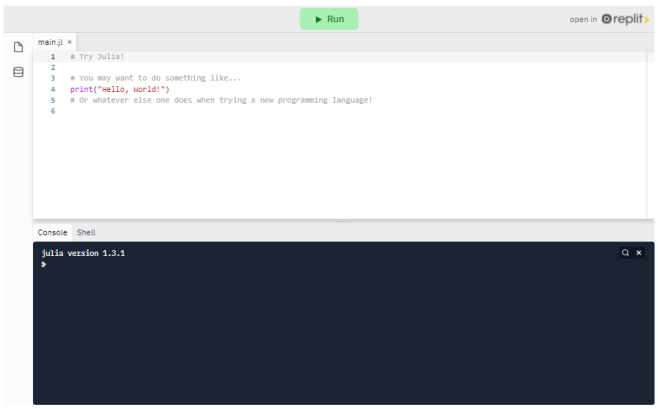
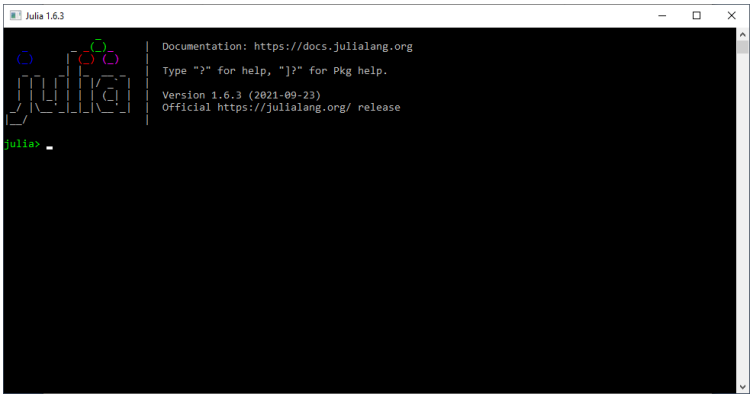


Figura 2: REPL do Julia.



Para atualizar as versões dos pacotes disponíveis, é necessário que após a instalação sejam executadas as linhas de comando “importPkg” e “Pkg.update()”. Sendo assim, depois de realizar esse processo, será executado o referido comando pelo Julia REPL representado pela Figura 3, ou seja, automaticamente o download dos pacotes e atualizações serão realizados.

Figura 3: Atualização do Julia via terminal.

Outra forma de implementar os códigos em linguagem Julia é utilizando editores (IDE- *Integrated Development Environment*) ao invés do Julia REPL. Alguns IDEs são sugeridos na página da linguagem Julia (link: https://julialang.org/), como exemplos, têm-se o VS Code, Jupyter Notebook, Pluto e Vim. Devido à simplicidade e vasta documentação, foi utilizado, no capítulo 10, o Jupyter Notebook. A forma mais comum de sua manipulação e/ou instalação é através da plataforma Anaconda. A referida plataforma é muito utilizada na Linguagem Python, que além de disponibilizar o Jupyter Notebook, oferece acesso a bibliotecas como Numpy, Pandas e outras.

Com a instalação do pacote Anaconda (versão 64 bits 2021.05) concluída com sucesso, através do link: https://www.anaconda.com/products/individual, uma tela da pasta do pacote Anaconda aparecerá, e dentro dela deve ser selecionado o instalador Jupyter Notebook, como destacado na Figura 4. Após clicar no ícone do Jupyter Notebook aparecerá o IDE no navegador.

Figura 4: Tela do pacote Anaconda e destaque do Jupyter Notebook.

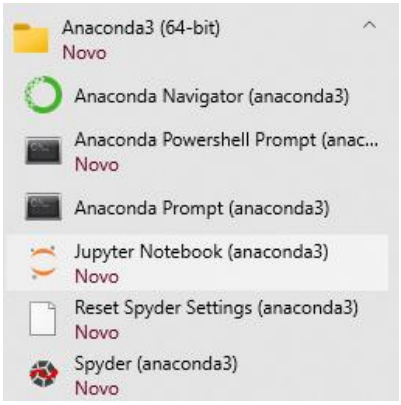
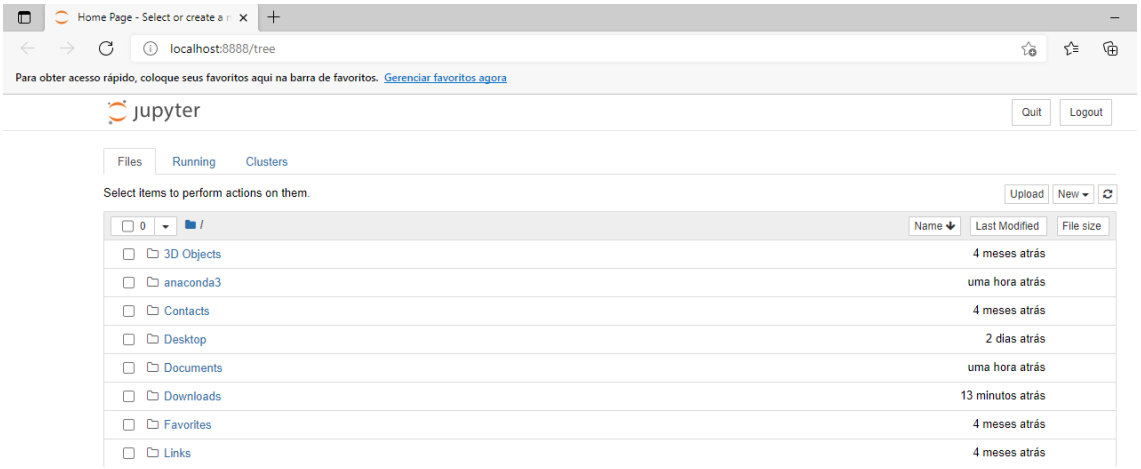


Figura 5: Tela do Jupyter Notebook.



Acessando a aba *New* da tela do Jupyter, pode-se observar que o Julia ainda não se encontra instalado, como visto na Figura 6. Para fazer a instalação do Julia no Jupyter podemos acessar o Julia REPL e digitar os códigos “using Pkg” e “Pkg.add(“IJulia”) ou apertando a tecla “]” do teclado e digitando a linha de código “addIJulia”, e caso o usuário deseje voltar ao modo “normal” do Julia REPL, deverá pressionar a tecla backspace, delete ou shift C, como retrata a Figura 7. Após instalar o Julia no Jupyter, é possível implementar os scripts no referido IDE.

Figura 6: Aba New do Jupyter.

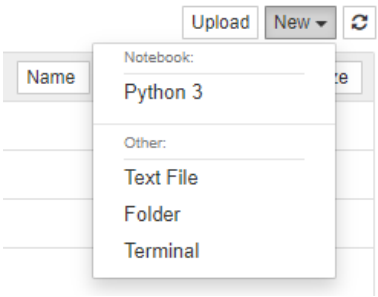


Figura 7: Instalação do Julia no Jupyter.

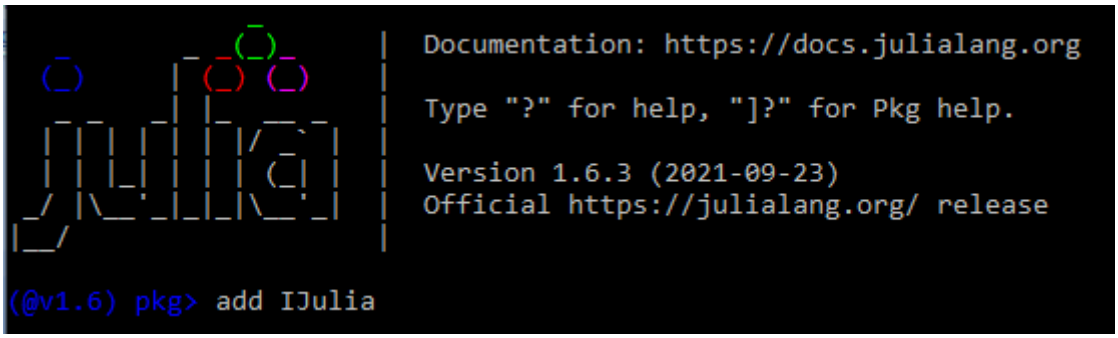
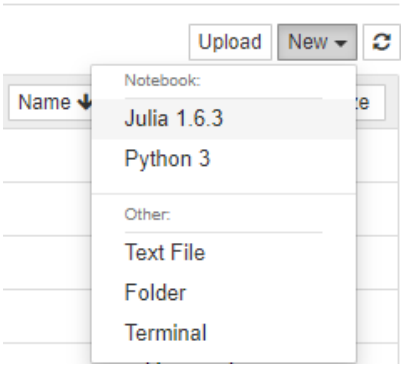


Figura 8: Julia instalado no Jupyter.



1.3 PRIMEIRO PROGRAMA

Ao abrir a interface da maioria dos softwares de programação, é exibida na tela inicial uma mensagem típica como “Olá Mundo” ou “Alô Mundo”. Este geralmente é o primeiro contato com um programa de um usuário, pois é utilizado como um teste ou como um primeiro exemplo de código em vários livros de linguagens de programação. Neste livro, foi realizada uma comparação do programa “Olá Mundo” em quatro linguagens diferentes de programação: C [5], Python [5], Java [5] e Julia,

respectivamente representadas nas Figuras 9 a 12. Para realizar comentários em Julia deverá ser utilizado o caractere "#". A função println() será abordada posteriormente, na seção 1.4. Quando o usuário estiver desenvolvendo programas mais elaborados, a função do Julia REPL será menor, e os códigos podem ser escritos em um arquivo com extensão .jl e executado no REPL.

Figura 9: Programa "Olá Mundo" em linguagem C.

```c
#include <stdio.h>
int main() {
 printf("Olá, Mundo!\n");
 return 0;
}
```

Figura 10: Programa "Olá Mundo" em linguagem Python.

```python
#!/usr/bin/env python
print("Olá, Mundo!")
```

Figura 11: Programa "Olá Mundo" em linguagem Java.

```java
class OlaMundo {
    public static void main(String args[]) {
        System.out.println("Olá, Mundo!");
    }
}
```

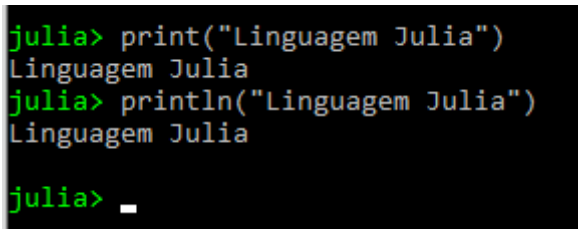
Figura 12: Programa "Olá Mundo" em linguagem Julia.

```
julia> # Olá Mundo

julia> println("Olá, Mundo!")
Olá, Mundo!
```

1.4 FUNÇÃO PRINTLN E PRINT

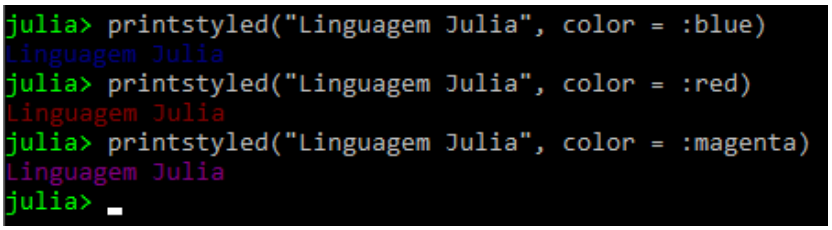
As funções mais comuns para imprimir a saída de um programa em linguagem Julia é o print () e println (). A principal diferença é que a função println () adiciona uma nova linha ao final da saída [6]. Veja a diferença na Figura 13. Para executar este comando, basta pressionar “Enter” no teclado.

Figura 13: Funções print() e println().



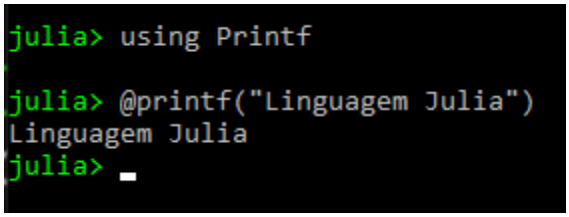
Em Julia, outras funções podem ser utilizadas com o mesmo propósito, mas possuindo características diferentes. A função printstyled () ajuda a imprimir mensagens em cores diferentes, como mostrado na Figura 14.

Figura 14: Função printstyled().



Julia também suporta a função printf () que é usada na linguagem C para imprimir a saída no console. Entretanto, a macro Julia (que é usada precedendo-a com o sinal @) fornece a função printf() e que precisa ser importado para ser usado como apresenta a Figura 15.

Figura 15: Função printf().



1.5 REFERÊNCIAS

[1] https://www.forbes.com/sites/suparnadutt/2017/09/20/this-startup-created-a-new-programming-language-now-used-by-the-worlds-biggest-companies/?sh=44da1107de2a

[2] https://www.tiobe.com/tiobe-index/

[3] J. M. Perreira, M. B. B. de Siqueira, "Linguagem de programação Julia: uma alternativa open source e de alto desempenho ao Matlab", Revista Principia, no. 34, pp. 132- 140, 2017.

[4] The Julia Language, The Julia Project, Julia 1.6 Documentation, 2021.

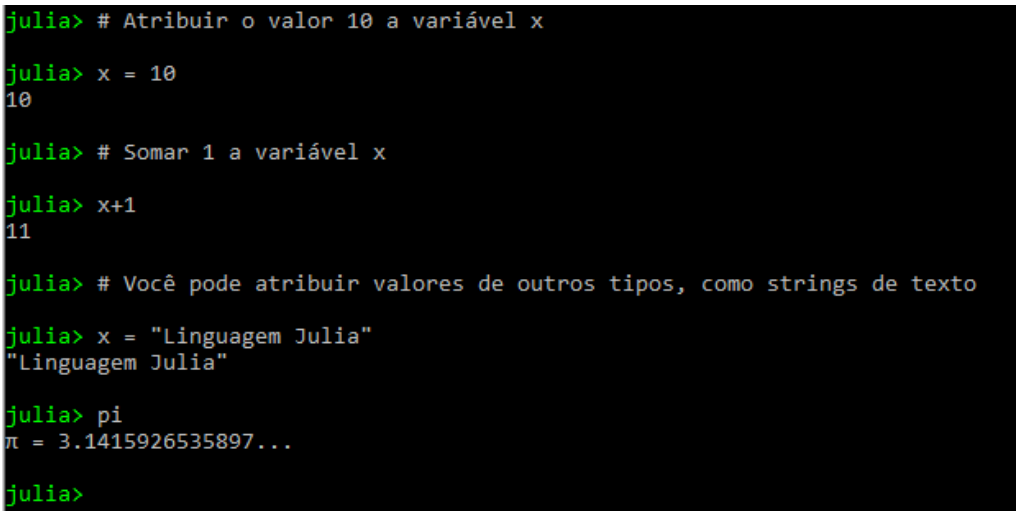
[5] https://terminalroot.com.br/2019/10/linguagem-de-programacao.html

[6] http://leandro.iqm.unicamp.br/leandro/shtml/didatico/simulacoes/tutorial-Julia.pdf

2. VARIÁVEIS

A variável é um aspecto primordial em qualquer linguagem de programação, podendo ser definida como um nome associado a um valor, a variável também é capaz de armazenar valores para uso posterior. É recomendado não atribuir nomes de variáveis iguais às constantes já pré-definidas na linguagem Julia, por exemplo, a constante pi (π), sendo apresentada na Figura 16.

Figura 16: Exemplo de manipulação de variáveis em Julia.



2.1 DECLARAÇÕES DE VARIÁVEIS

Embora a linguagem de programação Julia possua poucas restrições para nomes válidos de variáveis, é oportuno mencionar algumas restrições impostas [1]:

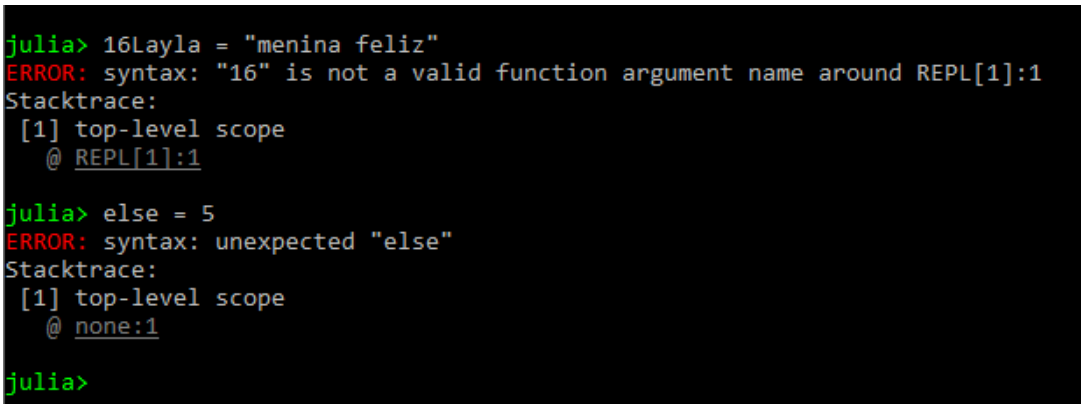
- Os nomes de variáveis em Julia devem começar com um sublinhado, uma letra (A-Z ou a-z) ou um caractere Unicode maior que 00A0 (NBSP – *No Break SPace*);
- Os nomes das variáveis também podem conter dígitos (0-9) ou !, mas não devem começar com eles;
- Operadores como (+, ^, etc.) também podem ser usados para nomear uma variável;
- Nomes de variáveis também podem ser escritos como palavras separadas por sublinhado, mas essa não é uma boa prática e deve ser evitada, a menos que seja necessário;

- O RELP de Julia utiliza palavras para reconhecer a estrutura do programa, logo essas palavras chave não devem ser usadas como nomes de variáveis.

As principais palavras chave utilizadas na linguagem Julia são: **abstract type**, **baremodule**, **begin**, **break**, **catch**, **const**, **continue**, **do**, **else**, **elseif**, **end**, **export**, **false**, **finally**, **for**, **function**, **global**, **if**, **import**, **let**, **local**, **macro**, **module**, **mutable struct**, **struct**, **primitive type**, **quote**, **return**, **true**, **using**, **while**.

Na Figura 17 apresenta alguns erros que podem ser cometidos ao nomear uma variável em Julia. O primeiro erro diz respeito que não é possível nomear uma variável iniciando com caracteres numéricos e o segundo erro é relativo às palavras chave em Julia.

Figura 17: Exemplo de erros em nomes de variáveis em Julia.



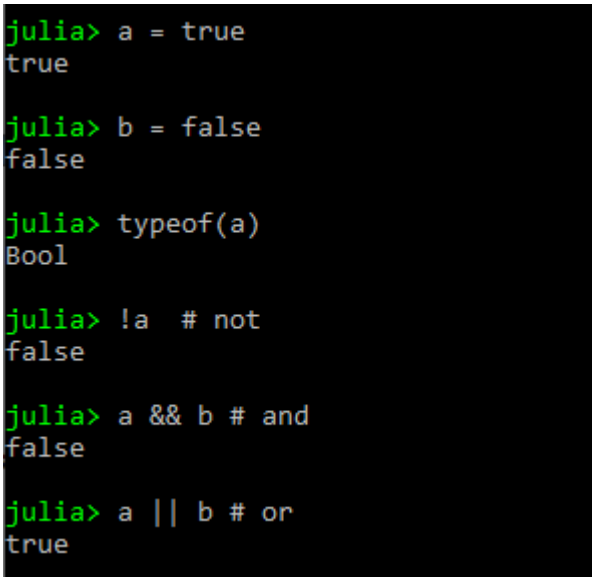
2.2 VALORES E TIPOS

A linguagem Julia possui uma variedade de tipos básicos que podem representar dados como: valores lógicos, números, strings, arrays, tuplas e dicionários. Além disso, existe uma função pré-definida em Julia denominada de typeof(), essa função possui como argumento de entrada o nome da variável e retorna o seu tipo de dado. A linguagem Julia permite que os usuários também possam definir seus próprios tipos. Nos itens a seguir serão abordados os valores e tipos de variáveis para a linguagem Julia.

a) Booleano (valores lógicos):

O tipo booleano em Julia, escrito Bool, inclui os valores true (verdadeiro) e false (falso), podendo atribuir esses valores às variáveis. Também são apresentadas as operações básicas da álgebra Booleana: not (!), and (&&) e or (||). A Figura 18 demonstra aplicações da função typeof() e as operações básicas envolvendo a álgebra Booleana.

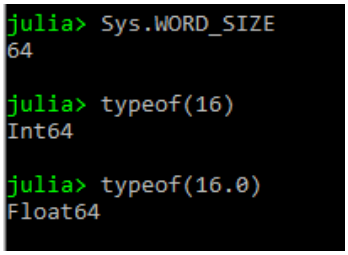
Figura 18: Tipo Bool e operações Booleanas.



b) Números:

Julia suporta números inteiros e pontos flutuantes (representação dos números reais em um computador, em inglês *float*). A Figura 19 demonstra os números representados na linguagem Julia (em uma máquina de 32 bits, o inteiro 16, por exemplo, seria interpretado como Int32). Caso o usuário deseje verificar se sua máquina é de 32 bits ou 64 bits, ele deverá utilizar a variável interna Sys.WORD_SIZE.

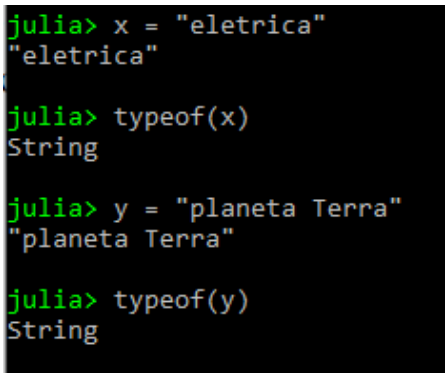
Figura 19: Representações de números em Julia.



c) Strings:

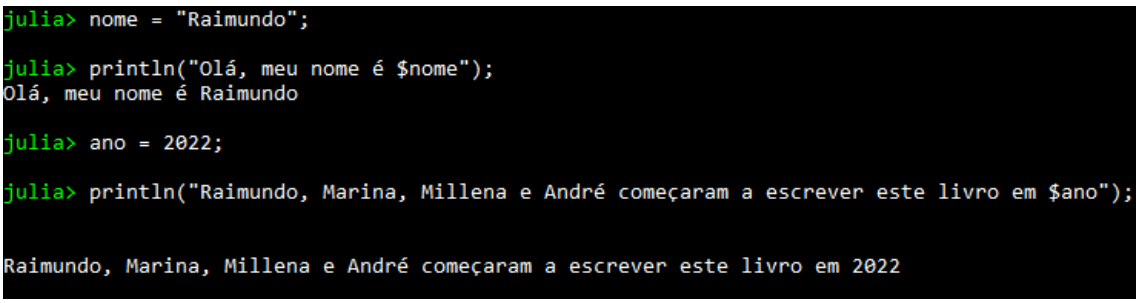
Em Julia as variáveis do tipo String (sequência de caracteres) são empregadas para armazenar e manipular textos como palavras, nomes e sentenças. Uma variável do tipo String pode ser construída utilizando “nome”. Como exemplo, temos a Figura 20.

Figura 20: String em Julia.



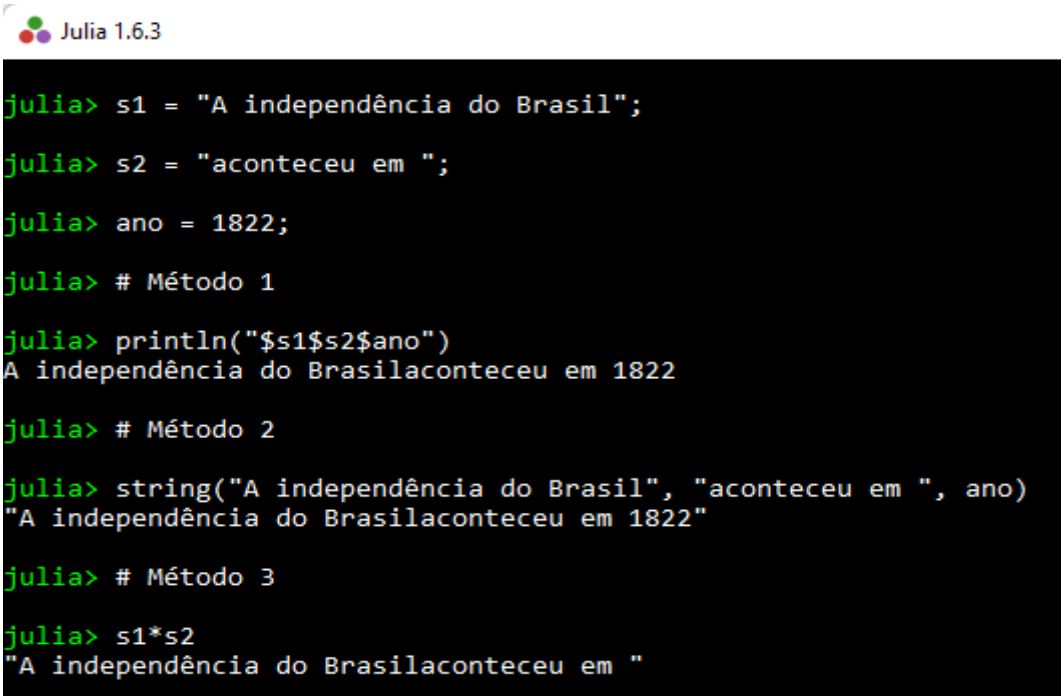
Uma observação interessante, em Julia o símbolo $ é utilizado para inserir variáveis (independente do tipo) em uma string. Conforme descrito na Figura 21.

Figura 21: Uso do símbolo $ em Julia.



Em Julia existe a interessante possibilidade de concatenar (juntar) strings. Neste contexto, há três maneiras de fazer isso (veja a Figura 22): (1) usando a interpolação de strings, (2) usando o método string() que também pode ser usado para converter outras entradas não string em strings e (3) utilizando o operador *, este por sua vez só é utilizado com strings, representando o método mais restrito dentre os citados anteriormente.

Figura 22: métodos de concatenação de strings em Julia.

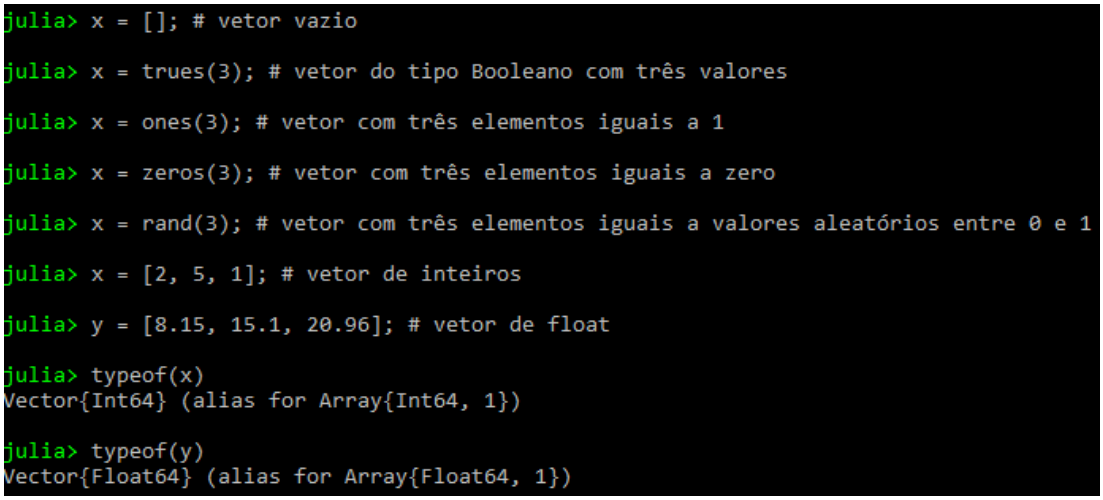


d) Matrizes e vetores:

Matrizes são estruturas em formato de tabelas. Os dados armazenados nas matrizes, sejam estes numéricos ou não, são organizados em linhas e colunas. Logo, uma matriz genérica, por exemplo, possui m colunas e n linhas. Dessa forma, os vetores são um tipo particular de matriz que possuem 1 linha e m colunas (vetores-linha) ou n linhas e 1 coluna (vetores-coluna).

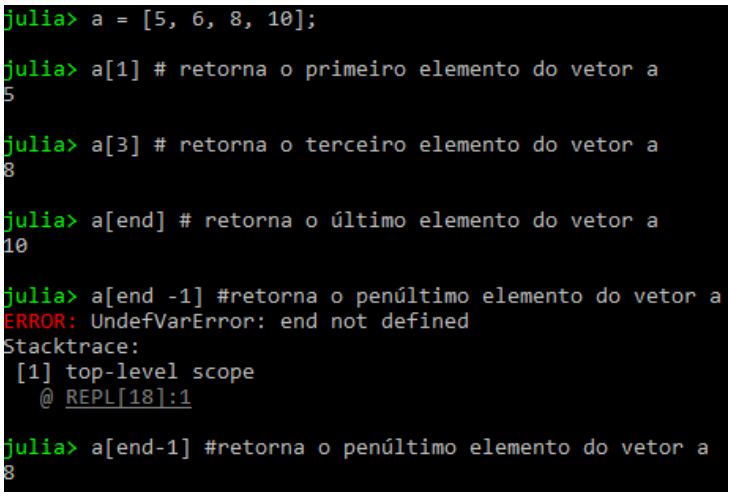
Usualmente na linguagem Julia, o vetor (matriz unidimensional) é representado entre colchetes, onde os elementos são separados por vírgulas. Nessa estrutura de dados pode-se armazenar uma determinada quantidade de valores no mesmo tipo. Conforme o algoritmo a seguir, os pontos e vírgulas no final de cada linha foram utilizados com o objetivo de não mostrar o resultado do comando na linha seguinte. Como pode ser visto na Figura 23, é permitido empregar a função typeof para retornar o tipo da estrutura (vetor é um *array* tipo 1).

Figura 23: Vetores em Julia.



A indexação de cada elemento de um vetor pode ser realizada utilizando colchetes. Na Figura 24 é apresentado a indexação de vetores em Julia e um típico erro cometido por programadores que ao usar a indexação do tipo end-1, utilizá-la sem espaços, pois o REPL da Julia irá entender o espaço como um comando.

Figura 24: Indexação de vetores em Julia.



Para extrair uma variedade de elementos de um vetor os intervalos são especificados utilizando uma notação de dois pontos. Na Fig. 25, uma variedade de extração de elementos de um vetor é apresentada. O comando y[1:2:end] é executado da seguinte maneira: começa com o índice 1 do vetor y, ou seja, y(1) e soma-se 2 a, logo o próximo elemento a ser impresso é y(3), depois soma-se 2, assim o próximo elemento a ser impresso é y(5), e assim continua até o final do vetor y.

Figura 25: Outras formas de indexação de vetores em Julia.

```
julia> print(y[1:3]) # retorna os três primeiros elementos de y
[2, 2, 8]
julia> print(y[1:2:end]) # retorna os elementos com índices ímpares do vetor y
[2, 8, 11]
julia> print(y[end:-1:1]) # retorna os elementos de y do último ao primeiro
[5, 11, 10, 8, 2, 2]
```

Várias operações podem ser realizadas com vetor em Julia, desde soma até potenciação, como descrito na Figura 26.

Figura 26: Operações com vetores em Julia.

```
julia> x = [4, 5, 8];

julia> y = [2, 1, -4];

julia> length(x) # retorna a tamanho do vetor x
3

julia> print(x+y) # adição de vetores
[6, 6, 4]
julia> print(x.*y) # multiplicação elemento por elemento de vetores
[8, 5, -32]
julia> print(x.^2) # potenciação elemento por elemento de vetores
[16, 25, 64]
julia> print(sqrt.(x)) # raiz quadrada elemento por elemento de vetores
[2.0, 2.23606797749979, 2.8284271247461903]
```

Quanto se trata de operações realizadas com matrizes, Julia oferece ferramentas para manipulação dessas estruturas. Alguns exemplos estão descritos na Figura 27.

Figura 27: Manipulação de matrizes em Julia.

```
julia> x = [1 2 3; 4 5 6; 7 8 9] # matriz com 3 linhas e 3 colunas
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> typeof(x) # tipo de x
Matrix{Int64} (alias for Array{Int64, 2})

julia> x[3,2] # elemento da terceira linha e segunda coluna de x
8

julia> print(x[1,:]) # retorna os elementos da primeira linha de x
[1, 2, 3]
julia> print(x[:,3])# retorna os elementos da terceira coluna de x
[3, 6, 9]
julia> size(x) # retorna o tamanho da matriz
(3, 3)

julia> print(x[1:2,1:2]) # retorna a submatriz com elementos x[1,1], x[1,2], x[2,1] e x[2,2]
[1 2; 4 5]
```

Algumas operações úteis em álgebra linear envolvendo matrizes e vetores podem ser realizadas em Julia, mas para realizar essas operações deve-se utilizar o pacote ***LinearAlgebra*** (falaremos de pacotes em Julia mais adiante). A Figura 28 apresenta diversos exemplos, onde demonstra que é possível realizar as operações de transposta de uma matriz, traço (soma dos elementos da diagonal principal de uma matriz), determinante, autovalores e inversa na linguagem Julia.

Figura 28: Operações com matrizes em Julia.

```
julia> using LinearAlgebra

julia> A = [4 5; 2 1];

julia> M = transpose(A) # retorna a tranposta de A
2×2 transpose(::Matrix{Int64}) with eltype Int64:
 4  2
 5  1

julia> det(A) # retorna o determinante da matriz A
-6.0

julia> eigvals(A) # retorna os autovalores da matriz A
2-element Vector{Float64}:
 -1.0
  6.0

julia> tr(A) # retorna o traço da matriz A
5

julia> inv(A) # retorna a inversa da matriz
2×2 Matrix{Float64}:
 -0.166667   0.833333
  0.333333  -0.666667
```

e) Tuplas:

As tuplas em Julia são uma coleção de valores imutáveis e distintos, ou seja, quando uma tupla é criada não é possível adicionar, alterar ou remover seus elementos. Além disso, diferentemente das matrizes e vetores, as tuplas são uma coleção heterogênea de valores. A sequência de valores armazenados em uma tupla pode ser de qualquer tipo e são indexados por inteiros. Os valores de uma tupla são separados sintaticamente por vírgulas. Em outros casos, embora não seja necessário, é mais frequente definir uma tupla fechando a sequência de valores entre parênteses. A Figura 30 apresenta alguns exemplos com tuplas.

Figura 29: Tuplas em Julia.

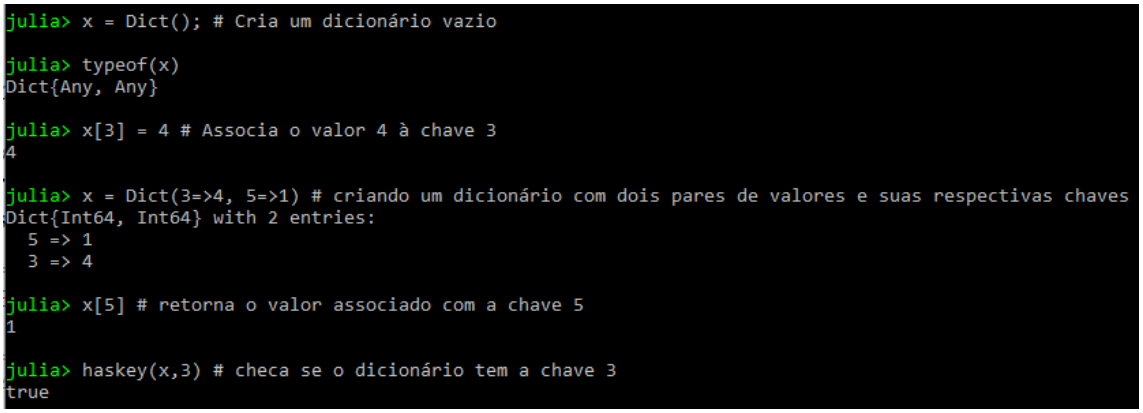
```
julia> trupla1 = () # Criando uma tupla vazia
()

julia> typeof(trupla1)
Tuple{}

julia> tupla1 = () # Criando uma tupla vazia
()

julia> typeof(tupla1)
Tuple{}

julia> tupla2 = (1, 2, 3, 4, 5) # Criando uma tupla com valores numéricos
(1, 2, 3, 4, 5)

julia> tupla3 = (1, 2, 3, "Julia") # Criando uma tupla com valores mistos
(1, 2, 3, "Julia")
```

f) Dicionários:

O dicionário em Julia é uma coleção de pares de valores-chave, onde cada valor no dicionário pode ser acessado com sua chave. Esses pares de valores-chave não precisam ser do mesmo tipo de dados, o que significa que uma chave digitada em String pode conter um valor de qualquer tipo, como inteiro, ponto-flutuante, string, etc. As chaves de um dicionário nunca podem ser as mesmas, cada chave deve ser única. Isso não se aplica aos valores, os valores podem ser os mesmos, conforme a necessidade. Os dicionários, por padrão, são uma coleção não ordenada de dados, ou seja, não mantém a ordem em que as chaves são inseridas [3], [4].

O conceito de dicionário se assemelha bastante com o de uma matriz, divergindo apenas em um ponto: no dicionário os índices podem ser de qualquer tipo, por outro lado em uma matriz, os índices devem ser apenas inteiros. Cada chave em um dicionário é mapeada para um valor. Como as chaves precisam ser únicas, duas chaves podem ser mapeadas com os mesmos valores, mas dois valores diferentes não podem ser mapeados com uma única chave [3]. Na Figura 30 é apresentado alguns comandos envolvendo dicionários. Para criar um dicionário, deve-se utilizar o comando **Dict**. A função **haskey** retorna um valor Booleano true ou false se o dicionário possui determinada chave [4].

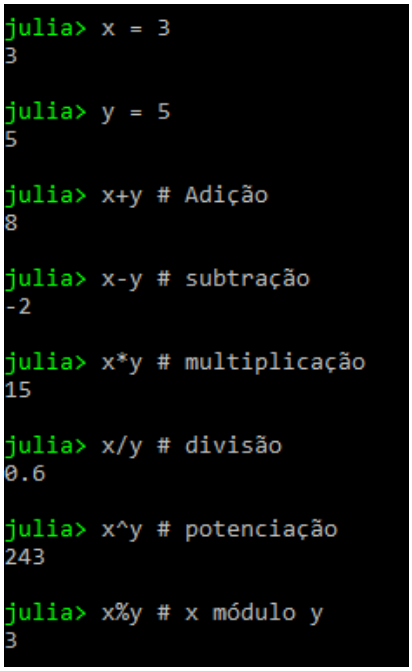
Figura 30: Dicionários em Julia.



2.3 OPERAÇÕES MATEMÁTICAS E FUNÇÕES ELEMENTARES

Na linguagem Julia também é possível realizar algumas operações matemáticas básicas: adição, subtração, multiplicação, divisão, potenciação e módulo (encontrar o resto da divisão de dois números). A Figura 31 apresenta as operações básicas em Julia. Existe em Julia (assim como em Matlab) a função rem que é similar a operação (%). Como observação, na operação módulo (%) quando o primeiro número é menor do que o segundo, retornará o primeiro número (veja que x<y na Figura 31).

Figura 31: Operações matemáticas básicas.



2.4 TIPOS COMPOSTOS

Os tipos compostos são objetos definidos pelo usuário, possuindo a finalidade de armazenamento de dados estruturados, sendo definidos em Julia com a construção *struct*. O Censo do IBGE e outras pesquisas semelhantes têm uma estrutura próxima a esta, só que com informações ainda mais detalhadas; por exemplo: O Censo Demográfico do IBGE (Instituto Brasileiro de Geografia e Estatística) é uma pesquisa voltada para coletar dados sobre a população brasileira com o objetivo de traçar um perfil socioeconômico para o país. Dentro da pesquisa existem vários itens que o entrevistado deve responder, aos quais, como exemplo tem-se: município (um string), nome do morador (um string), idade (um inteiro) e rendimento mensal da família (um flutuante). Ou seja, é realizado o uso de "enes" itens compostos por variáveis do tipo inteiras, string e ponto flutuante. Conforme abordado no exemplo do Censo do IBGE, e em alguns conjuntos de dados, as informações podem até conter caracteres mais complexos, como: imagens, mapas e unicode complexos. Para estes casos, Julia oferece a possibilidade da construção de arrays para armazenar essas informações [5].

A sintaxe em Julia para definir uma estrutura (struct) é dada pela Figura 32. O operador "::" define o tipo que é cada campo. Contudo, o uso deste operador não é obrigatório, pois um campo sem este operador é capaz de receber qualquer tipo de valor.

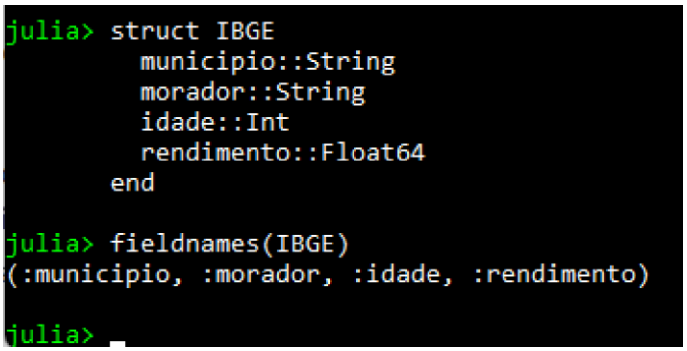
Figura 32: Struct em Julia.

```
struct nome
    campo 1:: Int
    campo 2::Float64
    campo 3::String
    campo 4
end
```

Implementando o exemplo citado anteriormente, a respeito de pesquisas que englobam vários dados, pode-se fazer uma analogia estrutural com a linguagem de programação, como: O nome do município (um string), morador (um string), idade (um inteiro) e média de rendimento mensal (um ponto flutuante). No entanto, a pesquisa pode conter mais detalhes, e para acessar a lista de nomes dos campos definidos para a

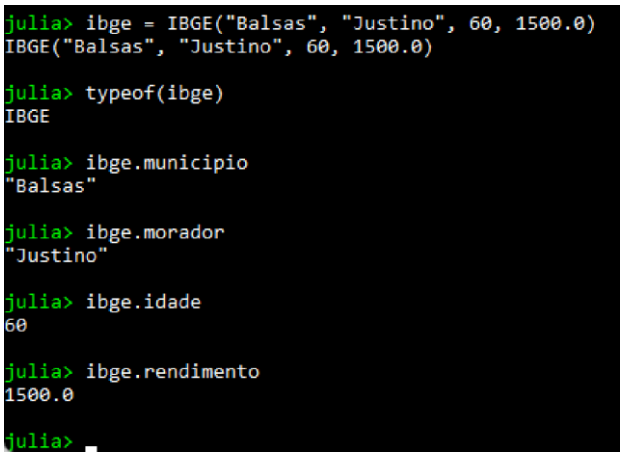
"estrutura IBGE" a função **fieldnames** poderá ser utilizada. A Figura 33 apresenta a estrutura IBGE implementada no REPL do Julia.

Figura 33: Estrutura IBGE implementada em Julia.



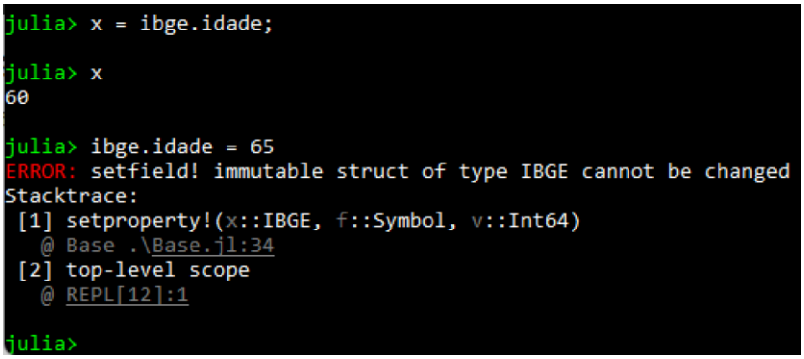
Com objetivo de fornecer valores aos campos da "estrutura IBGE" pode-se criar uma instância (variável) ibge conforme a Figura 34 aponta. O tipo da variável ibge é IBGE. Os valores da estrutura podem ser acessados utilizando a conhecida notação "." após o nome da variável seguida do campo. Por exemplo, para acessar o valor da idade, deverá ser chamado por ibge.idade.

Figura 34: Instâncias em Julia.



Podemos também atribuir valores dos campos da variável ibge a outra variável. Por exemplo, uma variável qualquer x recebe o valor do campo idade da variável ibge (veja Figura 35). Porém, se queremos mudar o valor do campo idade da variável ibge, aparece a mensagem de erro da Figura 35. Isso significa que a estrutura IBGE é imutável. Por padrão, as estruturas criadas em Julia são imutáveis, mas podem ser alteradas e este assunto é algo fora do escopo deste livro.

Figura 35: Erro com estrutura imutável em Julia.



2.5 EXERCÍCOS PROPOSTOS

1. De acordo com o conhecimento adquirido sobre variáveis e seus tipos, considere as seguintes afirmativas a seguir sobre sintaxe e marque a(s) que considerar correta(s).

( ) Os valores de uma tupla são separados sintaticamente por hashtags (#)

( ) A indexação de cada elemento de um vetor pode ser realizada utilizando colchetes.

( ) Para extrair uma variedade de elementos de um vetor os intervalos são especificados utilizando uma notação de dois pontos.

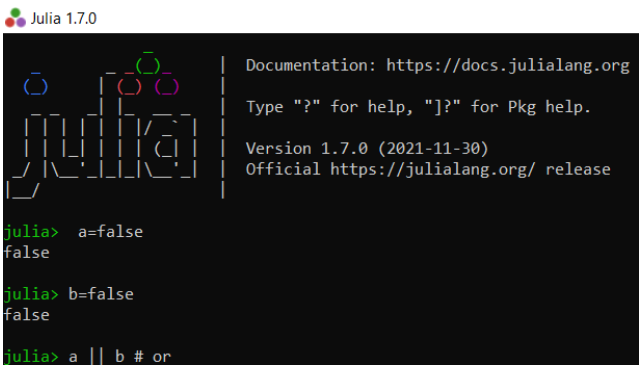
( ) A indexação de cada elemento de um vetor pode ser realizada utilizando &.

2. Cite as três possibilidades de concatenar strings em Julia.

3. Na linguagem Julia, os vetores estão representados entre _____ onde os elementos são separados por _____.

4. Qual será à saída do programa apresentado na Figura 36?

Figura 36: Referente ao exemplo 4- Capítulo 2.



5. Identifique os tipos das variáveis:

a) 1946

b) /0

c) 37.11

d) &&

e) UFMA

6. Quais os itens não são operadores booleanos?

a) %%

b) &&

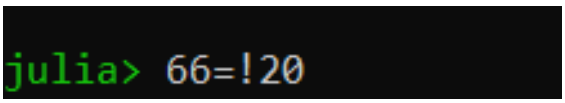
b) $$

c) !!

d) II

7. Aponte o erro da linha de código apresentada na Figura 37 e corrija-o na linha abaixo

Figura 37: Referente ao exemplo 7- Capítulo 2.



_____

8. Quais operadores são aritméticos a seguir?

a) ::

b) *

c) °

d) =

e) +

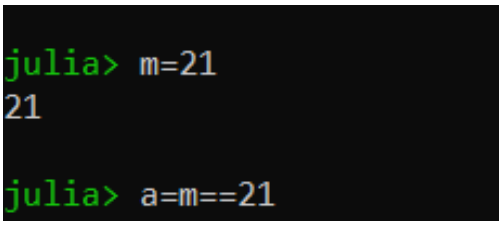
9. Identifique qual a saída da afirmação realizada na linha de código apresentada na Figura 38:

Figura 38: Referente ao exemplo 9- Capítulo 2.

_____

10. De acordo com a linha de código da Figura, qual será o valor de a?

Figura 39: Referente ao exemplo 10- Capítulo 2.



2.6 REFERÊNCIAS

[1] https://www.geeksforgeeks.org/variables-in-julia/.

[2] https://juliaintro.github.io/JuliaIntroBR.jl/#chap02.

[3] https://www.geeksforgeeks.org/julia-dictionary/.

[4] M. J. Kochenderfer; T. A. Wheeler, “Algorithms for optimization”, MIT Press, London, 2019.

[5] https://www.sas.upenn.edu/~jesusfv/Chapter_HPC_8_Julia.pdf

3. COMANDOS DE DECISÕES CONDICIONAIS

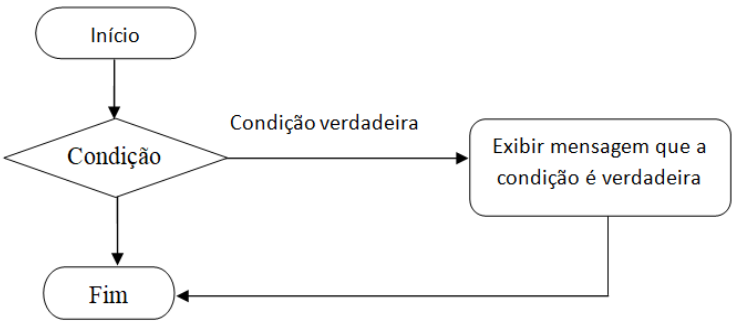
Dentre os aspectos mais relevantes das linguagens de programação, as estruturas condicionais são as que mais se destacam. Elas representam decisões que o algoritmo deverá tomar de acordo com as condições previamente estabelecidas. As decisões condicionais permitem a tomada de ações que serão executadas, podendo ser representadas através de expressões lógicas (verdadeiro ou falso). Na linguagem de programação Julia, ao checar se um valor é verdadeiro ou falso deve-se seguir o seguinte critério: se for igual a 0 (zero) a condição é falsa, caso contrário, a condição é verdadeira. As tomadas de decisões também são conhecidas como controle de fluxo. Algumas estruturas condicionais presente na linguagem Julia são citadas a seguir:

- IF;
- IF-ELSE;
- IF-ELSE-IF.

3.2 COMANDO IF

O comando IF pode ser interpretado como um comando de decisão simples, tem como principal característica o uso de apenas uma condição. A sintaxe do comando IF ocorre da seguinte maneira: a condição é avaliada primeiramente, e antes de executar qualquer bloco dentro da estrutura, é verificado através de identificadores lógicos, se o valor da expressão é diferente de 0 (zero), ou seja, o bloco de comando só será executado se a condição for verdadeira, caso ao contrário a execução será invalidada. O diagrama da Figura 40 representa o funcionamento da estrutura IF:

Figura 40: Representação do comando IF.



A implementação da função IF na linguagem de programação Julia é exemplificada a seguir na Figura 41.

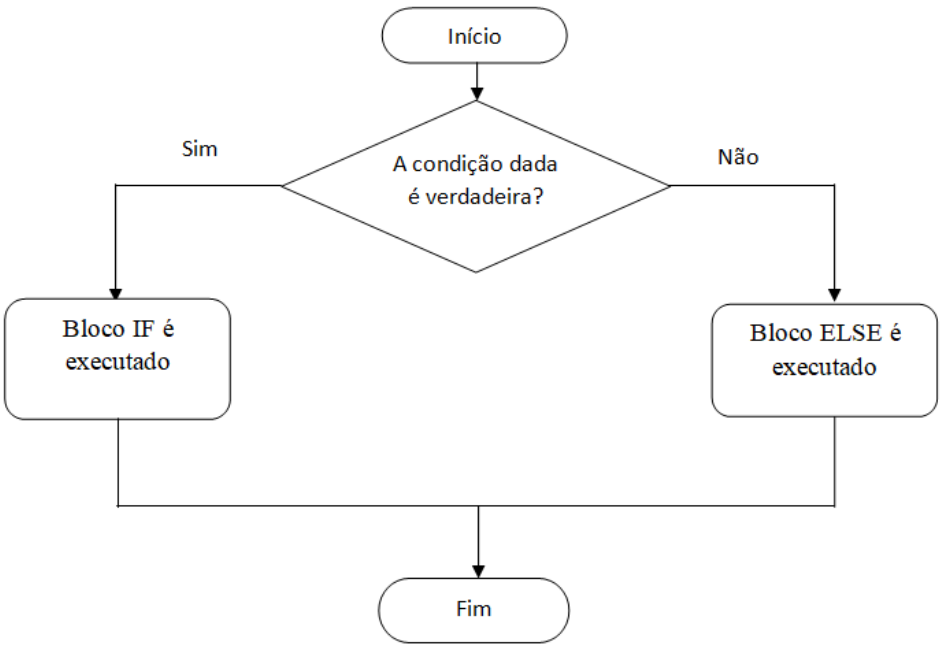
Figura 41: Implementação do comando IF em Julia.

```
julia> i = 47
47

julia> if (i>62)
       println("47 é maior que 62")
       end

julia> println("afirmação falsa")
afirmação falsa
```

3.3 COMANDO IF-ELSE

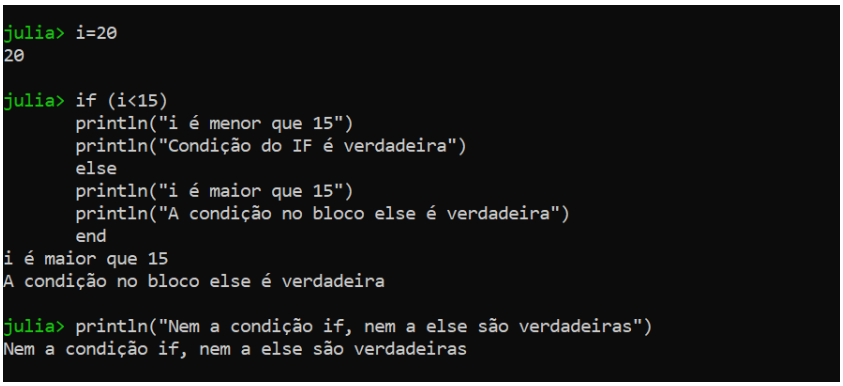
Por outro lado, o comando IF-ELSE representa um comando de decisão composta, destacando-se pela seguinte propriedade: entre dois blocos decidir qual será executado. Seguindo a condição: se o valor da expressão-teste for verdadeiro, ou seja, se o valor for diferente de 0, o bloco IF será executado. Caso contrário, se o valor da expressão for igual a 0 (falso) o comando ELSE será executado. A estrutura permite apenas que um bloco de comandos IF ou ELSE seja executado e não ambos. O fluxograma da Figura 42 seguir retrata o funcionamento do comando IF-ELSE.

Figura 42: Representação do comando IF-ELSE.



A implementação da função IF-ELSE na linguagem de programação Julia é exemplificada na Figura 43.

Figura 43: Implementação do comando IF-ELSE em Julia.



3.4 COMANDO IF-ELSE-IF

Um IF alinhado é um IF que é objeto de outro IF ou ELSE. Assim que uma expressão for verdadeira, o comando associado a ele é executado e desvia do fluxo de condições IF-ELSE-IF, caso todas as expressões sejam falsas, será executado os comandos do bloco ELSE. É semelhante à instrução IF-ELSE, aqui a única diferença é que uma instrução IF é anexada a ELSE. Se a condição fornecida para o bloco IF for verdadeira, então a instrução dentro do bloco IF será executada; ELSE-IF a outra condição fornecida for verificada e se for verdadeira, a instrução dentro do bloco será executada. O fluxograma e um exemplo em Julia do comando IF-ELSE-IF é apresentado nas Figuras 44 e 45, respectivamente.

Figura 44: Representação do comando IF-ELSE-IF.

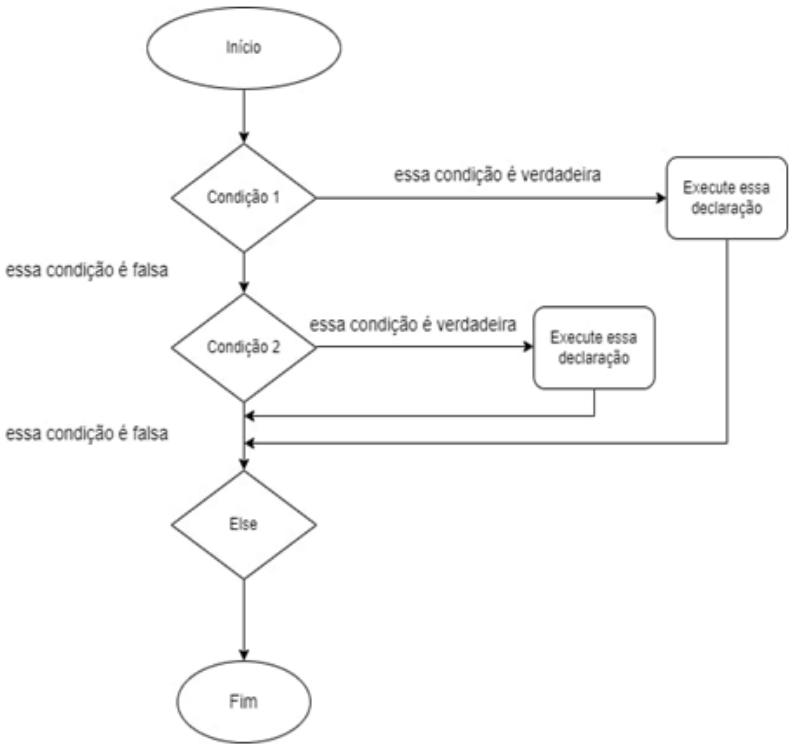


Figura 45: Implementação do comando IF-ELSE-IF em Julia.

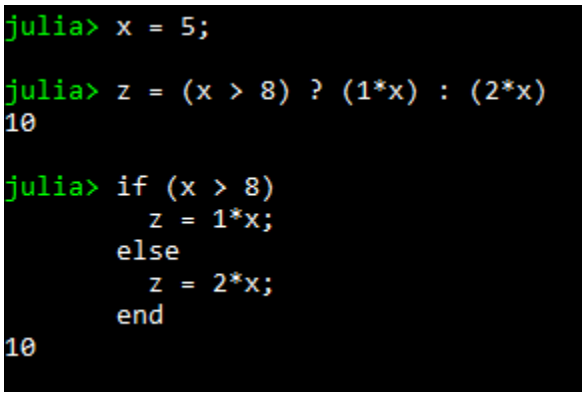
```
julia> i=27
27

julia> if (i==27)
       if (i<30)
       println(" i é menor que 30 ")
       if (i<25)
       println("i é menor que 25")
       else
       println("i está entre 25 e 30")
       end
       else
       println("i é maior que 30")
       end
       end
 i é menor que 30
i está entre 25 e 30
```

3.4 OPERADOR TERNÁRIO

O operador ternário “?:”, presente em várias linguagens de programação, é uma alternativa para substituir os comandos de decisões e sua sintaxe é bem simples. Há uma condição que dever ser validada como verdadeira ou falsa. A sintaxe é **[a] ? [b] : [c]** e significa “se **a** for verdadeiro, então avalie **b**, caso contrário, avalie **c**”. Os espaços ao redor de “?” e “:” são obrigatórios. A Figura 46 apresenta um exemplo da utilização do

operador ternário em Julia. A mesma resposta pode ser implementada com if-else, entretanto o código com o operador ternário é bem mais enxuto comparado com o if-else.

Figura 46: Operador ternário em Julia.



3.5 EXERCÍCIOS PROPOSTOS

1. O voto no Brasil é obrigatório e secreto, sendo coletado através da urna eletrônica. Diante disso, escreva um programa em Julia que leia a idade de um eleitor e verifique se ele pode ou não votar. Se ele puder votar, informe se o seu voto é facultativo. É ciente que 16 e 18 anos e os maiores de 70 anos não são obrigadas a votar, mas têm permissão se desejarem participar da eleição.

2. Escrever um programa em Julia que leia três valores e encontre o maior dos valores lidos. Escrever o maior valor ao final. Caso um valor lido já tenho sido digitado o programa deverá solicitar um novo valor ao usuário.

3. Faça um programa em Julia que receba um número inteiro e verifique se é par ou ímpar.

4. Faça um programa em Julia que leia os valores A, B, C e imprima na tela se a soma de A + B é menor que C.

5. Encontrar o dobro de um número caso ele seja positivo e o seu triplo caso seja negativo, imprimindo o resultado.

6. Faça um programa em Julia para checar se um número é positivo, negativo ou zero.

7. Faça um programa em Julia que receba quatro números e mostre-os em ordem decrescente. Suponha que o usuário digitará quatro números diferentes.

8. Faça um programa em Julia que leia uma variável e some 3 caso seja par ou some 6 caso seja ímpar, imprimindo o resultado desta operação.

9. Dados cinco números distintos, desenvolver um programa em Julia que determine e imprima a soma dos quatro menores.

10. Desenvolva um programa em Julia para ler três valores (considere que não serão lidos valores iguais) e escrever o maior deles.

3.6 REFERÊNCIAS

[1] https://www.geeksforgeeks.org/decision-making-in-julia-if-if-else-nested-if-if-elseif-else-ladder/

[2] DIEHL, A. S. Algoritmos. Departamento de física: UFPEL, 2017. Disponível em: https://wp.ufpel.edu.br/diehl/files/2017/09/lec3_algo.

[3] https://juliateachingctu.github.io/Julia-for-Optimization-and Learning/stable/lecture_03/conditions/

4. LAÇOS

Laços (*loop*, em inglês) são ferramentas importantes em linguagem de programação. Para a linguagem de programação Julia existem dois tipos de laços disponíveis: for e while. Do ponto de vista da sintaxe, os laços da linguagem Julia são similares com os da linguagem Python. A seguir será abordado detalhadamente os laços na linguagem Julia.

4.2 LAÇO FOR

Geralmente, o laço **for** é utilizado quando o número de iterações está pré-definido. A sintaxe em Julia deste tipo de laço é implementada conforme descrito na Figura 47. Na referida figura, "for" é a palavra-chave que indica que o usuário está fazendo uso do laço, em sequência a palavra-chave **in** é utilizada para definir um intervalo no processo de iteração e por fim, a palavra-chave **end** significa o fim do laço for.

Figura 47: Sintaxe do laço for em Julia.

```
for variável in lista
    instruções
end
```

Um exemplo da utilização do laço **for** é apresentado em Figura 48. Neste exemplo, a lista é representada pela instrução 1:5, ou seja, corresponde a uma lista de números sendo eles, 1, 2, 3, 4 e 5. O laço se inicia atribuindo esses valores, iniciando em 1, para variável i, e por fim, o comando println apresenta na tela do usuário os valores encontrados na lista. Dessa forma, o mesmo laço pode ser feito substituindo a palavra-chave **in** pelo símbolo = (**igual**), os resultados são os mesmos conforme a comparação apresentada na Figura 49. Entretanto, para fins de entendimento e clareza do código, fazer o uso do símbolo = (igual), facilita na compreensão do código, pois é a mesma sintaxe que a linguagem Matlab utiliza.

Figura 48: Exemplo de laço for em Julia.

```
julia> for i in 1:5
       println(i)
       end
1
2
3
4
5
```

Figura 49: Mesmo exemplo da Figura 48, porém de outra forma.

```
julia> for i = 1:5
       println(i)
       end
1
2
3
4
5
```

A linguagem Julia permite a criação de laços dentro de outros laços, essa estrutura é denominada como laços aninhados. A Figura 50 apresenta a implementação de laços for aninhados onde o programa gera uma lista de números de 1 a 5 em um formato triangular. A distribuição é realizada da seguinte maneira: O laço **for** externo (da variável **i**) faz o controle das linhas e o laço **for** interno (da variável **j**) faz o controle das colunas.

Figura 50: Laços for aninhados em Julia.

```
julia> for i in 1:5
          for j in 1:i
            print(i," ")
          end
          println()
       end
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

Além disso, os laços do tipo **for aninhados** ainda podem ser escritos de maneira mais simplificada em Julia. Por exemplo, o laço implementado na Figura 50 pode ser reescrito de maneira simplificada como mostra a Figura 51. Para o segundo caso, dois

laços **for** foram agrupados em um único laço, e para seguir para a próxima linha foi utilizado uma lógica envolvendo um **if**.

Figura 51: Laços for aninhados simplificados.

```
julia> for i in 1:5, j in 1:i
           print(i," ")
           if j == i
             println()
           end
       end
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

A função zip(), disponível na linguagem Julia, tem como finalidade percorrer dois ou mais arranjos (arrays) simultaneamente, por esse motivo, a sintaxe da função zip() é um pouco diferente da maioria das linguagens de programação. Para compreender melhor, um exemplo da utilização da função zip() é apresentado na Figura 52. Para essa função é necessário apenas passar os arranjos como argumento de entrada para função zip() no laço.

Figura 52: Função zip() em Julia.

```
julia> x = [1, 2, 3, 4, 5, 6, 7];

julia> y = [1, 4, 9, 16, 25, 36, 49];

julia> z = [1, 8, 27, 64, 125, 216, 343];

julia> for (i, j, k) in zip(x, y, z)
           println([i j k])
       end
[1 1 1]
[2 4 8]
[3 9 27]
[4 16 64]
[5 25 125]
[6 36 216]
[7 49 343]
```

4.3 LAÇO WHILE

Por outro lado, se o número de iterações não é pré-definido, utilizar o laço while se torna uma solução viável, uma vez que o laço while executa um conjunto de instruções **até que** uma condição seja atendida. Em Julia, a sintaxe é apresentada como mostra a Figura 53.

Figura 53: Sintaxe do laço while em Julia.

```
while expressão
    instruções
end
```

A Figura 54 apresenta o mesmo problema implementado com o laço for (veja a Figura 48) utilizando o laço while. Primeiramente, deve-se iniciar o contador, no caso à variável i. Depois, a expressão de condição do laço while indica que enquanto a variável i for menor ou igual a 5 as instruções dentro do laço são executadas (no caso, são a instrução println e o incremento do contador i). Por fim, o laço é encerrado quando o contador i é igual a 6 (não satisfaz a expressão de controle do laço).

Figura 54: Exemplo simples usando o laço while em Julia.

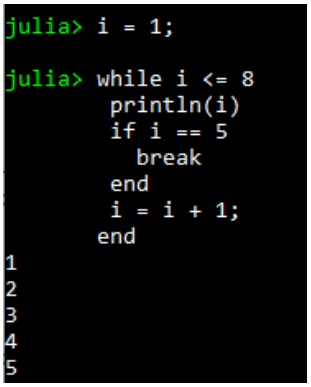
```
julia> i = 1;

julia> while i<=5
         println(i)
         i = i + 1;
       end
1
2
3
4
5
```

4.4 COMANDO BREAK DENTRO DO LAÇO

O comando **break** pode ser utilizado em Julia para sair de um laço imediatamente, seja um laço **for** ou **while**. Toda vez que o comando break é executado, o compilador imediatamente para de iterar mais valores e envia o "ponteiro" de execução para fora do laço. Um simples exemplo é apresentado em Figura 55, neste referido exemplo, o contador **i** tem inicialmente valor igual a 1. A lógica desse código é interpretada da seguinte forma: A expressão de parada no laço while indica que enquanto o contador **i** não for menor ou igual a 8 todas as instruções dentro do laço são executadas. Porém, é fornecida uma condição de parada dada pelo comando break em conjunto com o comando if, a condição é que a cada iteração do laço while, o contador é incrementado em 1 unidade (linha do código: i = i + 1). Dessa forma, quando o

contador é igual a 5, o programa entra no corpo do comando if, e consequentemente, é executado o comando break, sinalizando o encerramento do laço [1].

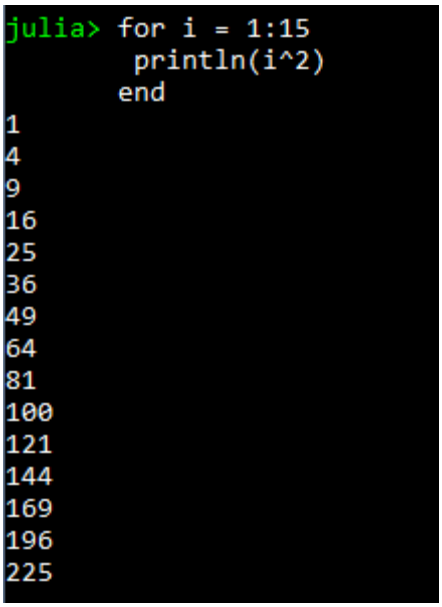
Figura 55: Exemplo simples usando o laço while e comando break.



4.5 EXEMPLOS DIVERSOS COM CÓDIGOS

Exemplo 4.1: Faça um programa que imprima o quadrado de todos os inteiros de 1 a 15.

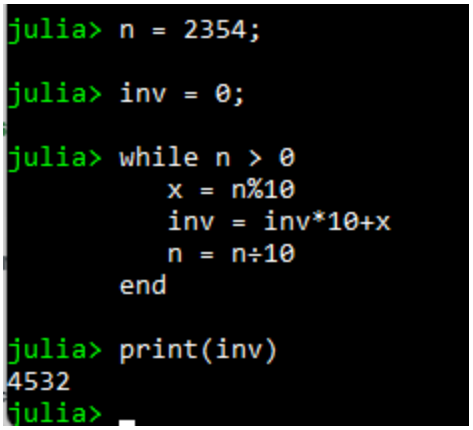
Figura 56: Resposta para o Exemplo 4.1.



Comentário: Para esse exemplo, o laço **for** foi implementado, pois a quantidade de iterações é conhecida (dada no enunciado).

Exemplo 4.2: Faça um programa que inverta a ordem dos algarismos de um número inteiro. Por exemplo, o número 2354 e a saída do programa do programa é 4532.

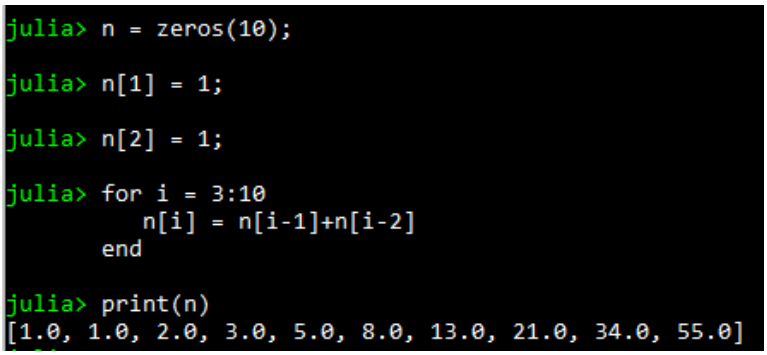
Figura 57: Resposta para o Exemplo 4.2.



Comentário: Para esse exemplo, as ferramentas matemáticas são utilizadas através da lógica de programação. Como visto anteriormente no capítulo sobre variáveis, do ponto de vista da linguagem de programação Julia, os símbolos **%** e ÷ equivalem ao resto da divisão da variável n por 10 e o valor inteiro da divisão da variável n por 10, respectivamente.

Exemplo 4.3: Faça um programa imprima os 10 primeiros números da sequência de Fibonacci.

Figura 58: Resposta para o Exemplo 4.3.



Comentário: Na sequência de Fibonacci os dois primeiros são 1. O terceiro termo é dado pela soma dos dois primeiros, ou seja, 1+1. Dessa forma, o quarto termo é dado pela soma do terceiro e o segundo termo, isto é, 1+2. Generalizando, dado a sequência de Fibonacci, $n = \{n_1, n_2, n_3, \cdots, n_{i-2}, n_{i-1}, n_i\}$, então $n_1 = 1$, $n_2 = 1$ e $n_i = n_{i-1} + n_{i-2}$, $\forall i \geq 3$.

4.6 EXERCÍCIOS PROPOSTOS

1. A população do país X é de 10.000.000 habitantes, essa população possui uma taxa de crescimento equivalente a 6% ao ano. Por outro lado, a população do país Y é de 212.000.00 habitantes sendo a taxa de crescimento de 4.6%. Faça um programa em Julia que calcule e escreva o número de anos necessários para que a população do país X ultrapasse ou iguale a população do país Y, mantidas as taxas de crescimento.

2. Faça um programa em Julia que solicite uma nota que esteja entre o intervalo de zero a dez. Depois mostre uma mensagem caso o valor seja inválido e continue pedindo até que o usuário informe um valor válido.

3. Dados dois números inteiros positivos, faça um programa em Julia que calcule o quociente e o resto da divisão inteira entre os dois, usando apenas somas e subtrações.

4. Crie um programa em Julia que retorna se um número é um palíndromo (ou seja, um número que apresenta a mesma sequência de unidades nos dois sentidos, exemplo, 121) ou não.

5. Faça um programa em Julia que leia os números no intervalo de 1 a 10, e que gere a tabuada do número escolhido.

6. Faça um programa em Julia que o usuário informe os números desejáveis e efetue a multiplicação entre as variáveis “a” e “b” utilizando adições sucessivas.

7. Faça um programa em Julia que calcule e mostre o produto dos inteiros ímpares de 1 a 31.

8. Felipe tem 1,40 metros e cada ano cresce 2 cm, enquanto Pedro tem 1,30 metros e cresce 3 cm por ano. Faça um programa em Julia que mostre quantos anos são necessários para que Pedro seja maior ou igual à altura de Felipe.

9. Faça um programa em Julia que calcule o valor total investido por um colecionador em sua coleção de CDs e o valor médio gasto em cada um deles. O usuário deverá informar a quantidade de CDs e o valor para em cada um.

10. Os números primos possuem várias aplicações dentro da computação, por exemplo, na criptografia. Um número primo é aquele que é divisível apenas por um e por ele mesmo. Faça um programa em Julia que peça um número inteiro e determine se ele é ou não um número primo.

4.7 REFERÊNCIAS

[1] https://docs.julialang.org/en/v1/manual/control-flow/#man-loops.

5. FUNÇÕES

A organização dos programas desenvolvidos na linguagem de programação em Julia é realizada através de funções, estas por sua vez são responsáveis pelo processamento de uma parte do algoritmo. Uma função pode ser definida como um conjunto de comandos agrupados em um determinado bloco, que recebe um "nome" e através dele pode ser "chamado" no algoritmo. Além disso, por executar uma tarefa em específico, a função pode ser "chamada" por diversas vezes no mesmo algoritmo, evitando a redundância de código no mesmo programa.

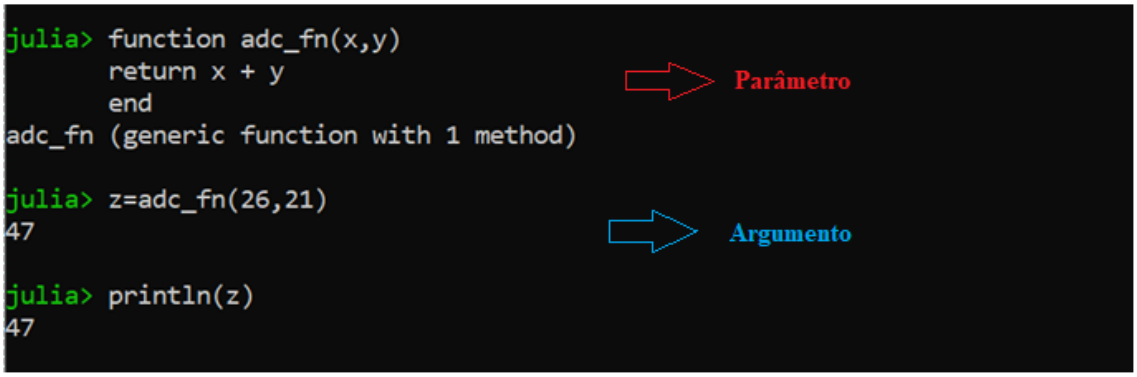
As principais funções utilizadas na linguagem de programação Júlia são:

- Função com expressão única;
- Função com várias expressões;
- Função sem argumento;
- Função com argumentos variáveis.

5.2 FUNÇÕES COM E SEM ARGUMENTO

Frequentemente os termos argumento e parâmetro são vistos como sinônimos, mas quando se trata de terminologia o significado entre eles diferem. Enquanto os argumentos são mais adequadamente considerados como referências atribuídas às variáveis, ou seja, são os valores reais atribuídos para a função. Por outro lado, parâmetro se destaca por ser a variável na declaração da função. Como exemplifica a Figura 59:

Figura 59: Diferença entre os termos parâmetro e argumento.



A função com argumento e sem argumento, de maneira prática, pode ser definida como:

- Função sem argumento: Parâmetro vazio;
- Função com argumento: Parâmetros definidos entre parênteses.

Implementado em Julia, a função sem argumento e com argumento, respectivamente nas Figuras 60 e 61.

Figura 60: Função sem argumento em Julia.

```
julia> function m()
       println("dentro da função")
       end
m (generic function with 1 method)
```

Figura 61: Função com argumento em Julia.

```
julia> function adc_fn(a,b)
       return a+b
       end
adc_fn (generic function with 1 method)

julia> s=adc_fn(47,16)
63

julia> println(s)
63
```

5.3 FUNÇÃO “VARARGS”

A função que contém números variáveis de argumentos, comumente é reconhecida como função “varargs”. Na maioria dos programas é conveniente escrever funções com um número arbitrário de argumentos, por exemplo, para criar uma função “varargs”, podem-se utilizar reticências no último argumento posicional, como demonstra a Figura 62.

Figura 62: Função “Varargs”.

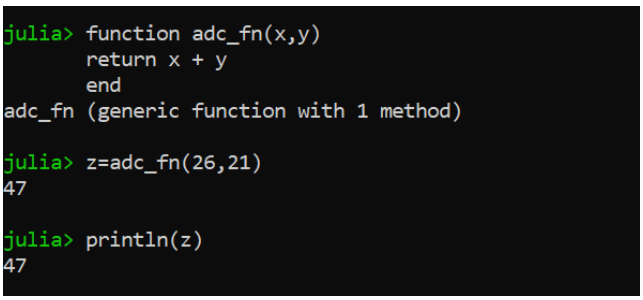
```
julia> bar(a,b,x...)=(a,b,x)
bar (generic function with 1 method)

julia> bar(2,3)
(2, 3, ())

julia> bar(4,5,6)
(4, 5, (6,))

julia> bar(7,8,9,10)
(7, 8, (9, 10))

julia> bar(2,3,4,5,6,7)
(2, 3, (4, 5, 6, 7))

julia> bar(1,2,3,4,5,6,7,8,9)
(1, 2, (3, 4, 5, 6, 7, 8, 9))
```

Conforme o programa da Figura 62 é possível observar que os termos definidos arbitrariamente como “a” e “b” estão vinculados aos dois primeiros valores do argumento, enquanto “x” é o termo variável da lista, podendo ser adicionada qualquer quantidade de números que o usuário desejar. Os termos adicionados são facilmente identificados dentro do segundo parêntese na função.

5.4 COMANDO RETURN

Para retornar um valor de uma função, a palavra-chave de retorno é usada. Isso retornará o valor calculado de acordo com as instruções definidas na função, para a instrução de onde a função está sendo chamada. Uma instrução de retorno também pode ser usada para calcular a operação da função apenas escrevendo toda a operação após a instrução de retorno. Isso resultará em menos linhas de código, por não escrever as instruções para calcular e, em seguida, retornar o valor calculado, conforme apresenta a Figura 63.

Figura 63: Implementação do comando return em Julia.



5.5 FUNÇÕES ANÔNIMAS

A linguagem de programação Julia também permite a criação de funções sem nome, estas são conhecidas com funções anônimas. A principal característica é o uso da palavra-chave “ans” que tem como objetivo passar os argumentos para a função anônima, como a Figura 64 representa. Outra maneira de criar funções anônimas em Julia é utilizando o símbolo “->”, como ilustra a Figura 65.

Figura 64: Primeira forma de implementar funções anônimas em Julia.

```
julia> function(x)
       x^2+2x-1
       end
#7 (generic function with 1 method)

julia> ans(5)
34
```

Figura 65: Segunda forma de implementar funções anônimas em Julia.

```
julia> x-> x^2+2x-1
#1 (generic function with 1 method)

julia> ans(5)
34
```

Uma das principais aplicações das funções anônimas é passá-las como argumento de outras funções. Por exemplo, a função map() aplica na função anônima cada valor de um vetor e retorna um novo vetor que contém os valores resultados, como ilustra a Figura 66.

Figura 66: Aplicação da Função map() em Julia.

```
julia> map(x-> x^2-1, [1, 0, -1, 2, 1.5])
5-element Vector{Float64}:
  0.0
 -1.0
  0.0
  3.0
  1.25

julia> map(y -> exp(y), [-1, 0, 1, 2, 3])
5-element Vector{Float64}:
  0.36787944117144233
  1.0
  2.718281828459045
  7.38905609893065
 20.085536923187668
```

Uma função anônima com mais de uma variável (mais de 1 argumento) obedece à sintaxe x_1, x_2, x_3, ..., x_n -> f(x_1, x_2, x_3, ..., x_n). Para compreender melhor o uso dessa função, o código representado pela Figura 67 é apresentado.

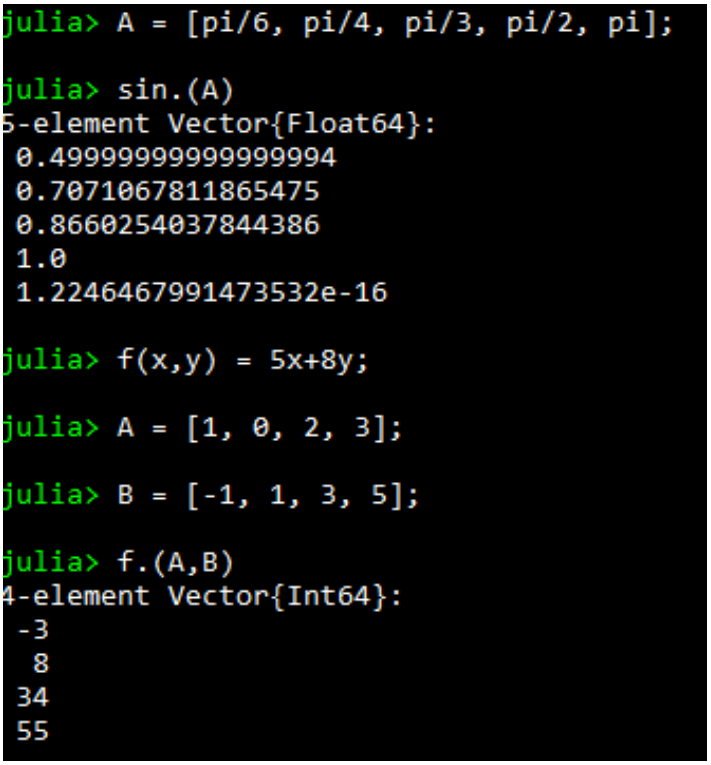
Figura 67: Função anônima com múltiplos argumentos.

```
julia> (x, y, z) -> x^2+y^2+z^2
#13 (generic function with 1 method)

julia> ans(2, 3, 4)
29
```

5.6 FUNÇÕES VETORIZADAS

Em algumas linguagens de programação, é comum ter versões "vetorizadas" de funções, que simplesmente aplicam uma determinada função f(x) a cada elemento de um vetor v (ou matriz) objetivando gerar um novo vetor (ou matriz) através da aplicação f(v). Esse tipo de sintaxe é conveniente para processamento de dados e geralmente aumenta o desempenho do código (por exemplo, se os laços forem lentos, a versão "vetorizada" pode aumentar o desempenho computacional do programa). Em Julia, as funções vetorizadas não são utilizadas para aumentar o desempenho devido à característica da própria linguagem, mas são utilizadas para tornar o programa mais organizado e conciso. Portanto, em Julia, qualquer função f pode ser aplicada elemento a qualquer vetor (ou outra coleção) com a sintaxe f.(A) (seria similar a map(f, A)). Por exemplo, a Figura 68 apresenta um simples exemplo de vetorização da função seno e f(x,y) = 5x+8y.

Figura 68: Vetorização de funções em Julia.



5.6 EXEMPLOS DIVERSOS

A Figura 69 apresenta um exemplo de função com argumento. A função em questão retorna o produto de dois números e foi nomeada de “m”. No exemplo da Figura 70 é apresentada uma função anônima. O uso de “ans” para execução de funções anônimas é fundamental. Na Figura 71 observamos o uso do return na função g(x,y). O uso do return em g(x,y) não é necessário. Para retornar o produto x*y simplesmente poderíamos inserir x*y como penúltima linha função (antes de end). Diferentemente do caso da Figura 71, o uso do return, apresentado na função da Figura 72, é necessário [2]. Observe que existem três possíveis pontos de retorno da função hypot(x,y). Isso significa que a referida função pode retornar os valores de três expressões diferentes, dependendo dos valores de x e y (observação: o último return poderia ser omitido e fica a cargo do leitor responder a esta observação). A Figura 73 apresenta uma ampliação do uso de funções do tipo Varags. Por fim, a Figura 74 apresenta um simples exemplo de função sem argumento, algo muito comum em Julia.

Figura 69: Função com argumento

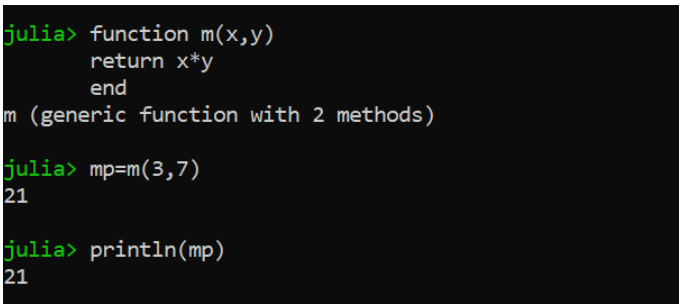


Figura 70: Função anônima.

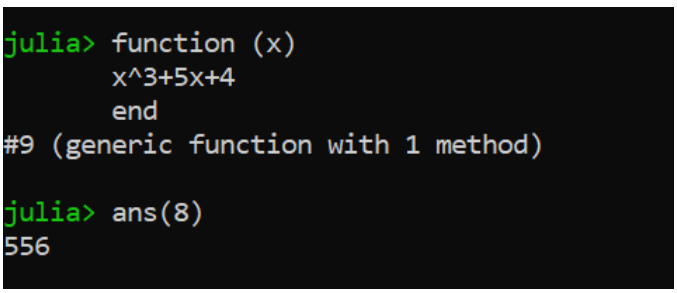


Figura 71: Uso do return- caso 1.

```
julia> f(x,y)=x+y
f (generic function with 1 method)

julia> function g(x,y)
       return x*y
       x+y
       end
g (generic function with 1 method)

julia> f(6,8)
14

julia> g(6,8)
48
```

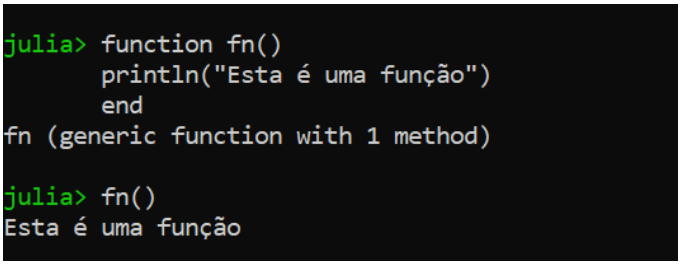
Figura 72: Uso do return- caso 2.

```
julia> function hypot(x,y)
       x=abs(x)
       y=abs(y)
       if x> y
       r=y/x
       return x*sqrt(1+r*r)
       end
       if y==0
       return zero(x)
       end
       r=x/y
       return y*sqrt(1+r*r)
       end
hypot (generic function with 1 method)

julia> hypot(5,6)
7.810249675906656
```

Figura 73: Utilizando vetores na função Varags.

```
julia> x=[8,9]
2-element Vector{Int64}:
 8
 9

julia> bar(1,2,x...)
(1, 2, (8, 9))

julia> x=[3,4,5,6,7]
5-element Vector{Int64}:
 3
 4
 5
 6
 7

julia> bar(x...)
(3, 4, (5, 6, 7))
```

Figura 74: Função sem argumento.



5.7 EXERCÍCIOS PROPOSTOS

1. Criar um algoritmo que possa entrar com cinco números e para cada um, imprimir o triplo, usando uma função que retorne valor.

2. Escreva uma função que retorne o maior número entre números de ponto flutuante.

3. Crie um programa que receba os dois lados menores de um triângulo retângulo e uma função que retorne o valor da hipotenusa.

4. Crie um jogo onde o computador sorteie um número de 1 até 20, e você tente adivinhar qual é.

5. Aos moldes do jogo par ou ímpar, crie o jogo da Pedra, Papel ou Tesoura, em Julia.

6. Desenvolver uma função que identifique e retorne o peso ideal de uma pessoa, a partir do fornecimento da altura e do sexo para essa função na forma de parâmetros. Para o fornecimento do sexo, assuma que sejam admitidos os seguintes valores (caracteres): sexo masculino “M” e sexo feminino “F”. Para cálculo do peso ideal, adote as seguintes fórmulas: para homem igual a [(72,7 × Altura) – 58] e para mulher igual a [(62,1 × Altura) – 44,7].

7. Criar uma função que receba um caractere como parâmetro e retorne 1, caso seja uma vogal e 0 em caso contrário.

8. A organização dos programas desenvolvidos na linguagem de programação em Julia é realizada através de funções. Por que usá-las?

9. Descrita no final do século 12 pelo matemático italiano Leonardo Fibonacci, a sequência de Fibonacci é uma sucessão de números que aparece codificada em muitos fenômenos da natureza, é infinita e começa com 0 e 1. Os números seguintes são sempre a soma dos dois números anteriores. Portanto: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34... Diante disso escreva uma função que calcule o seu n-ésimo número da sequência de Fibonacci.

10. Faça uma função em Julia que peça um número inteiro positivo 'n' para o usuário e imprima um quadrado de lado 'n' preenchido de *hashtags* (#). Por exemplo, para n=5, deve aparecer na tela:

```
#####
#####
#####
#####
#####
```

5.8 REFERÊNCIAS

[1] https://www.geeksforgeeks.org/functions-in-julia/

[2] https://julia-pt-br.readthedocs.io/pt_BR/latest/manual/functions.html

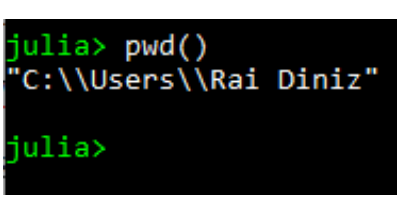
6. CARREGAMENTO DE CÓDIGOS

Nas seções anteriores, os códigos foram escritos e executados no REPL. A linguagem Julia oferece ao programador possibilidades no carregamento de códigos. Partindo deste pressuposto, é possível realizar o carregamento de scripts de duas outras formas, sendo a primeira escrever os códigos e utilizá-los posteriormente (salvando estes e carregando-os no REPL) e a segunda possibilidade efetuando o carregamento dos pacotes, objetivando assim a otimização dos scripts em tarefas mais complexas, funcionando como uma caixa fechada.

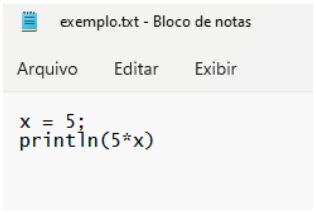
6.2 EXECUÇÃO DE SCRIPTS

Na linguagem Julia, os programas ou códigos (scripts) são salvos na extensão .jl. Para a execução dos scripts a função **include()** em Julia é essencial. No REPL, os caminhos incluídos são interpretados em relação ao diretório de trabalho atual (função **pwd()**) [1]. É possível criar e executar os scripts em um IDE (por exemplo, o Jupyter), mas por enquanto será utilizado o REPL do Julia para realizar esta tarefa. Primeiramente é necessário conhecer qual o diretório atual de trabalho (dado pela função pwd()- conforme a Figura 75(a) aponta). Posteriormente, em qualquer editor de texto (bloco de notas no nosso caso) deverá ser criado o seguinte script ilustrado na Figura 75(b). Após seguir esses passos, é possível salvá-lo no diretório de trabalho como exemplo.jl (ou qualquer outro nome que o usuário deseje). Por exemplo, ao abrir o REPL do Julia e digitar o código include ("exemplo.jl"). O resultado deve ser similar ao da Figura 76. A título de comparação, o mesmo código digitado no REPL é escrito e executado. Para os dois casos, o mesmo resultado é impresso na tela do REPL.

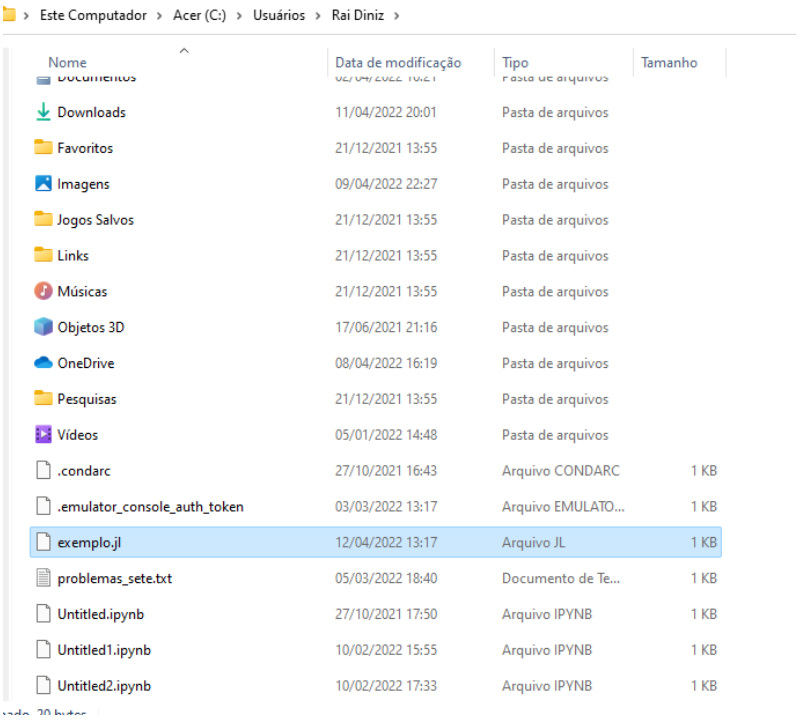
Figura 75: Criando script .jl.



(a)

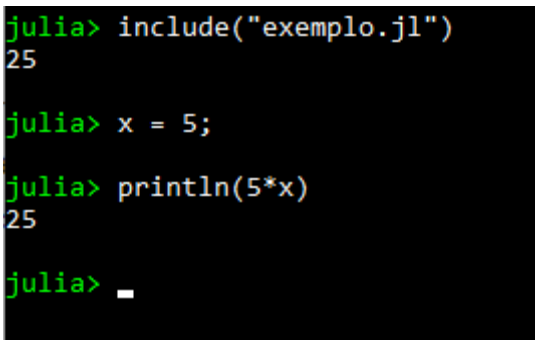


(b)



(c)

Figura 76: Executando script do arquivo .jl.



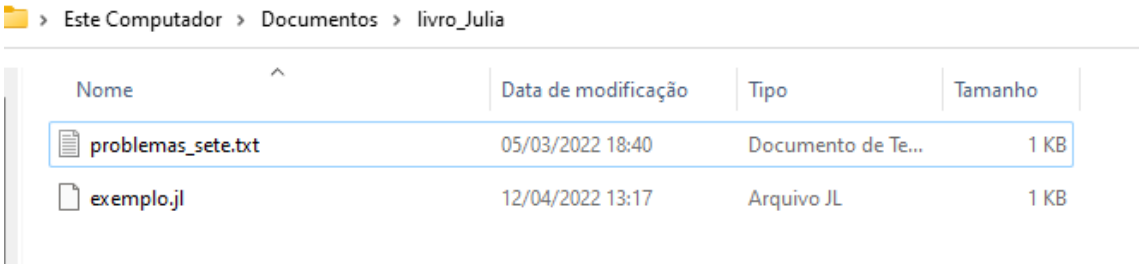
Caso os usuários desejem salvar os scripts em um diretório específico, há a possibilidade de mudar o diretório de trabalho. A função **cd()** pode ser utilizada neste caso. Por exemplo, o diretório atual de trabalho é (C:\Users\Rai Diniz), mas queremos mudar para (C:\Users\Rai Diniz\Documents\livro_Julia), para isso pode-se seguir as

linhas de códigos da Figura 77. Definindo este diretório onde ficarão salvos os scripts, é possível enviar todos os scripts para aquele diretório com o objetivo de ter mais organização, conforme apresenta Figura 78 (a) e (b).

Figura 77: Função pwd().

```
julia> cd("C:\\Users\\Rai Diniz\\Documents\\livro_Julia")

julia> pwd()
"C:\\Users\\Rai Diniz\\Documents\\livro_Julia"
```

Figura 78: Executando programas em diretório especificado.



(a)

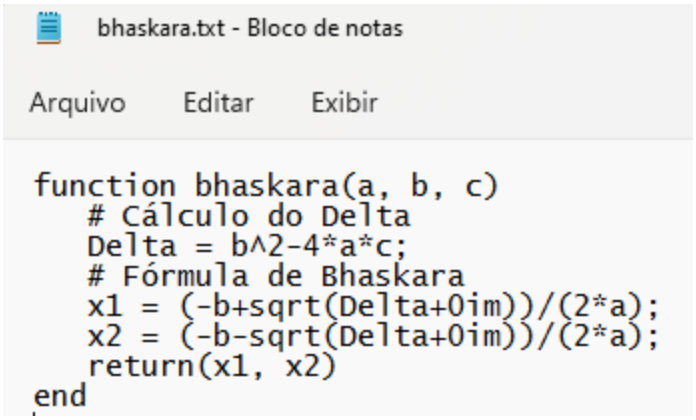
```
julia> cd("C:\\Users\\Rai Diniz\\Documents\\livro_Julia")

julia> pwd()
"C:\\Users\\Rai Diniz\\Documents\\livro_Julia"

julia> include("exemplo.jl")
25
```

(b)

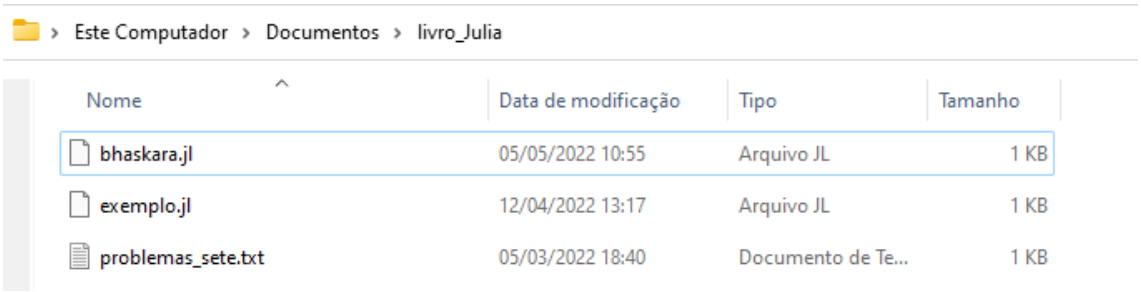
Algo muito comum em linguagem Julia é realizar a chamada de funções implementadas em .jl e e abri-las no REPL ou outro programa com a extensão .jl. As funções em Julia foram abordadas no Capítulo 5 deste livro. Portanto, será criado um exemplo de uma função que calcule as raízes de uma equação do segundo grau. A lógica de programação é descrita a seguir: os argumentos de entrada da função serão os coeficientes a, b e c (ou seja, ax^2+bx+c) e os argumentos de saída serão as raízes da função (x_1 e x_2). As raízes de uma equação do segundo grau podem ser calculadas através da Fórmula de Bhaskara. As Figuras 79 (a) e (b) ilustram a função bhaskara.jl e o diretório ao qual foi salvo a referida função, respectivamente.

A função bhaskara foi implementada em um simples editor de texto (no caso o Bloco de Notas do Windows) e salvo em .jl (extensão de scripts da Linguagem Julia). Observe no código da Figura 79 (a) que o argumento de entrada da função sqrt() é Delta+0im. O índice im em Julia indica que o argumento de saída da função sqrt() pode ser um número complexo caso o valor de Delta seja negativo [2]. Se utilizássemos simplesmente sqrt(Delta), exibiria uma mensagem de erro, como indica a Figura 80.

Figura 79: Função Bháskara.

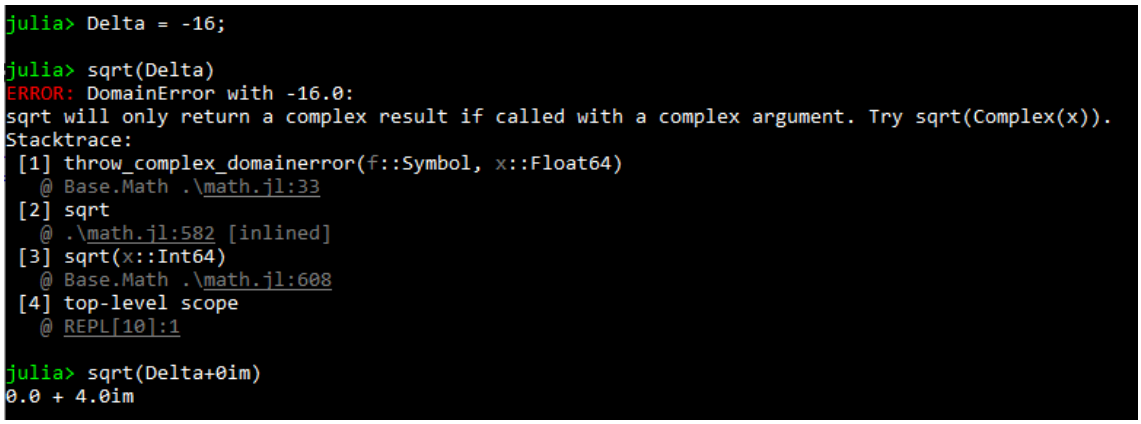


(a)



(b)

Figura 80: Função sqrt() com números negativos em Julia.



Para iniciar a chamada da função no REPL de um arquivo .jl externo, primeiramente deve-se verificar o diretório de uso do REPL. A função pwd() indica que o diretório de trabalho é (C:\Users\Rai Diniz), logo devemos mudar o diretório para (C:\Users\Rai Diniz\Documents\livro_Julia), pois a função bhaskara.jl está no referido diretório. Após mudar o diretório de utilização, pode-se utilizar a função bhaskara.jl como apresenta a Figura 81. Três casos foram utilizados: x^2-5x+6, $4x^2$-4x+1e x^2-6x+10.

Figura 81: Utilizando a função bhaskara.

```
julia> pwd()
"C:\\Users\\Rai Diniz"

julia> cd("C:\\Users\\Rai Diniz\\Documents\\livro_Julia")

julia> pwd()
"C:\\Users\\Rai Diniz\\Documents\\livro_Julia"

julia> include("bhaskara.jl")
bhaskara (generic function with 1 method)

julia> bhaskara(1, -5, 6)
(3.0 + 0.0im, 2.0 - 0.0im)

julia> bhaskara(4, -4, 1)
(0.5 + 0.0im, 0.5 - 0.0im)

julia> bhaskara(1, -6, 10)
(3.0 + 1.0im, 3.0 - 1.0im)
```

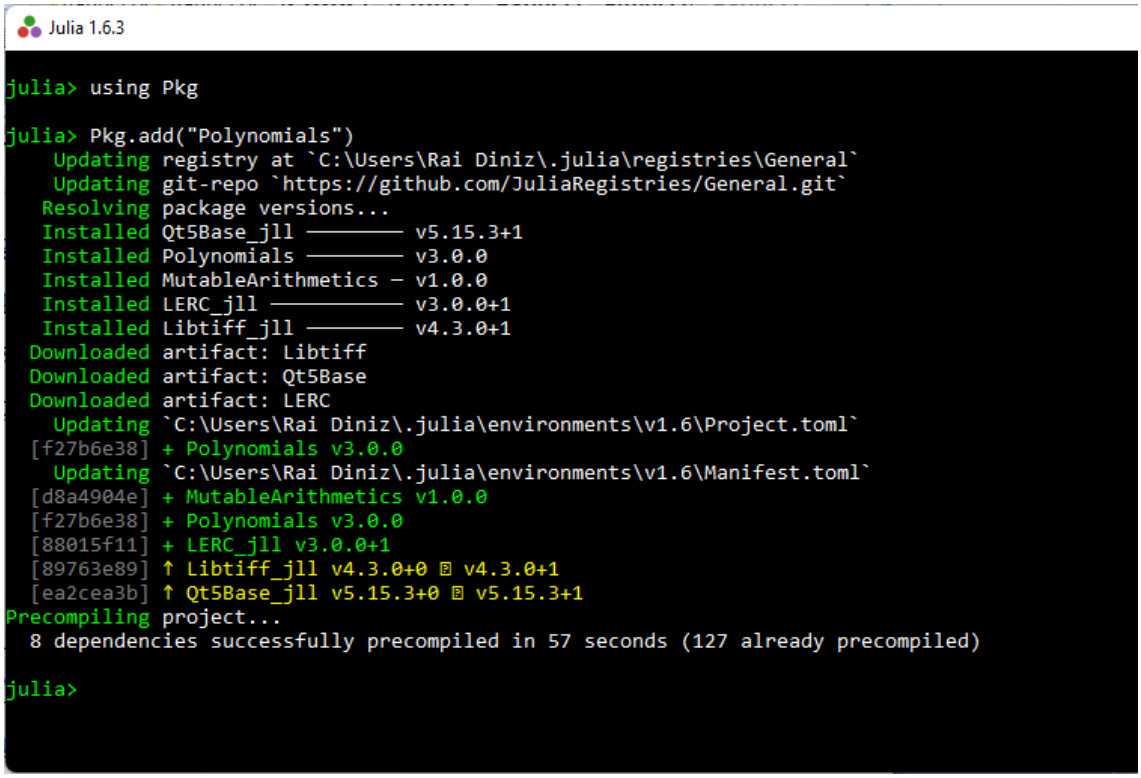
6.3 CARREGAMENTO DE PACOTES

Antes de definir pacotes, será apresentado o conceito de módulos. Em muitas linguagens de programação os módulos e pacotes são maneiras fáceis de organizar o código para facilitar o uso e acesso. Em geral, um módulo são arquivos que possuem funções, variáveis, etc. Por outro lado, os pacotes simplesmente são uma coleção de um ou mais módulos conectados hierarquicamente. Portanto, antes de desenvolver um pacote, o programador deverá se certificar de entender completamente como os módulos se comportam em Julia [3].

O Pkg é um gerenciador de pacotes embutido em Julia, responsável por lidar com operações como instalação, atualização e remoção de pacotes. A lista de pacotes em Julia registrados pode ser encontrada em [4]. Nos exemplos a seguir será

implementado alguns códigos para resolver as equações do segundo grau através do pacote Polynomials.jl [5]. Caso o programador deseje instalar um pacote em Julia basta digitar no REPL os seguintes comandos: using Pkg e Pkg.add("nome do pacote"). A Figura 82 indica o processo de instalação do pacote Polynomials em Julia.

Figura 82: Instalação do pacote Polynomials.



Após o pacote ter sido adicionado, deve-se consultar a documentação do Polynomials.jl. Na referida documentação, é possível observar que o citado pacote tem uma função denominada de roots cuja sintaxe é roots(Polynomial[c, b, a]) para solução de equação do segundo grau ($ax^2+bx+c = 0$). A saída da função são as raízes de uma equação polinomial. Executando a função roots do pacote Polynomial.jl para cada um dos exemplos da Figura 81 obtemos a solução exibida na Figura 83 (veja que é igual a solução obtida na Figura 81).

Figura 83: Utilizando o pacote Polynomials.

```
julia> using Polynomials

julia> roots(Polynomial([6, -5, 1]))
2-element Vector{Float64}:
 1.9999999999999998
 3.0000000000000004

julia> roots(Polynomial([1, -4, 4]))
2-element Vector{Float64}:
 0.5
 0.5

julia> roots(Polynomial([10, -6, 1]))
2-element Vector{ComplexF64}:
 3.0 - 1.0000000000000007im
 3.0 + 1.0000000000000007im
```

Para finalizar este capítulo, será resolvido um sistema linear, Ax = b, com o pacote IterativeSolvers.jl. No referido pacote há uma coleção de métodos iterativos para solução de sistemas lineares: Jacobi, Gauss-Seidel, SOR, Gradiente Conjugado, MINRES, GMRES, BiCGStab e etc.

O sistema linear da Equação (1), será resolvido utilizando o pacote IterativeSolvers.jl [6], adotando o método Gauss-Seidel. A solução exata do sistema é $x_1 = 1$ e $x_2 = -1$.

$$\begin{cases} 2x_1 + x_2 = 13 \\ x_1 + 4x_2 = -1 \end{cases} \tag{1}$$

Primeiramente é necessário realizar a instalação do pacote. Seguindo o exemplo de instalação do pacote Polynomials.jl será utilizado o gerenciador de pacote Pkg. O pacote foi instalado com sucesso, como ilustra a Figura 84.

Figura 84: Instalação do pacote IterativeSolvers.

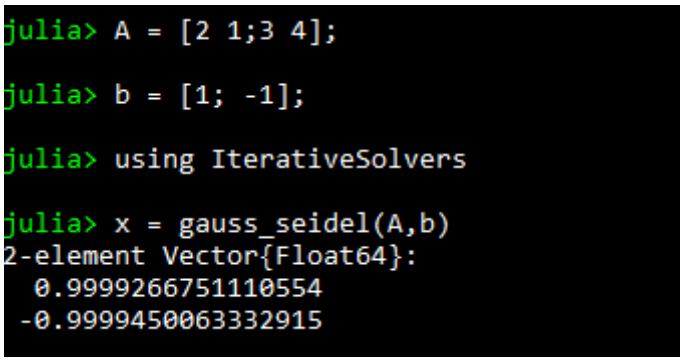
```
julia> using Pkg

julia> Pkg.add("IterativeSolvers")
    Updating registry at `C:\Users\Rai Diniz\.julia\registries\General`
    Updating git-repo `https://github.com/JuliaRegistries/General.git`
   Resolving package versions...
   Installed IterativeSolvers — v0.9.2
    Updating `C:\Users\Rai Diniz\.julia\environments\v1.6\Project.toml`
  [42fd0dbc] + IterativeSolvers v0.9.2
    Updating `C:\Users\Rai Diniz\.julia\environments\v1.6\Manifest.toml`
  [42fd0dbc] + IterativeSolvers v0.9.2
Precompiling project...
  1 dependency successfully precompiled in 9 seconds (135 already precompiled)
```

No referido sistema linear a matriz de coeficientes e o vetor de termos independentes são dados por (2):

$$A = \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} e\ b = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad (2)$$

Através da documentação do pacote IterativeSolvers [7] a sintaxe da função para resolver o sistema linear com o método de Gauss-Seidel é gauss_seidel(A, b). Logo, o código completo no REPL do Julia é apresentado em Figura 85. Como o método de Gauss-Seidel é do tipo iterativo, a solução é bem próxima da exata.

Figura 85: Utilização do pacote IterativeSolvers.



Por fim, para remover um pacote em Julia basta digitar o código Pkg.rm("nome do pacote") no REPL.

6.4 REFERÊNCIAS

[1] https://docs.julialang.org/en/v1/manual/code-loading/ https://docs.julialang.org/en/v1/manual/complex-and-rational-numbers/

[2] https://syl1.gitbook.io/julia-language-a-concise-tutorial/language-core/11-developing-julia-packages

[3] https://juliapackages.com/

[4] https://juliamath.github.io/Polynomials.jl/stable/

[5] https://juliapackages.com/p/iterativesolvers

[6] https://iterativesolvers.julialinearalgebra.org/dev/linear_systems/stationary/#Gauss-Seidel

7. ARQUIVOS

O processo de leitura e gravação de dados em um computador é semelhante ao modo como se lê e grava dados na vida real. Por exemplo, para acessar as informações em um livro, primeiramente ele deverá ser aberto, depois as palavras serão lidas, talvez escritas novas palavras, e por fim o livro deverá ser fechado. Similarmente, quando precisa ser lido o código de dados de um arquivo, é obrigatório fornecer um local de arquivo e, em seguida, o computador traz esses dados para a memória RAM e os analisa. Da mesma forma, quando seu código precisa gravar dados em um arquivo, o computador coloca novos dados no buffer de gravação na memória do sistema e os sincroniza com o arquivo no dispositivo de armazenamento [1]. Neste contexto, para programadores, é muito comum escrever código para ler e escrever arquivos, mas cada linguagem lida com essa tarefa de forma um pouco diferente. Em Julia, há funções especializadas para leitura e escrita de arquivos. Os detalhes serão vistos a seguir.

7.2 ABERTURA E LEITURA DE ARQUIVOS

A manipulação de arquivos em Julia é feita essencialmente usando as funções open() e read(). A função close() fecha o arquivo ou variável que contém a instância de um arquivo aberto. Há outras funções que iremos ver adiante como com objetivos específicos, por exemplo, readline() e readlines() [1]- [3].

A primeira etapa para utilização das funções de abertura, leitura e gravação de arquivos em Julia é a criação do próprio arquivo ao qual queremos utilizar. Inicialmente, iremos utilizar arquivos de texto que podem ser criados em um bloco de notas. Vamos criar um arquivo de texto (.txt) que pode ser aberto pelo bloco de notas. Simplesmente, poderíamos abrir o bloco de notas e digitar as informações que queremos ler. Entretanto, em Julia existem as funções pwd() e touch(), que podem ser utilizadas para criação do nosso arquivo de texto. Primeiramente, a função pwd() é utilizada para informar o diretório que estamos trabalhando e assim fornecer o caminho onde desejamos criar o arquivo. Já a função touch() em essência cria o arquivo de texto. Vamos a um exemplo apresentado na Figura 86. O arquivo criado, denominado de "problemas_sete", pela função touch() está no diretório "C:\\Users\\Rai Diniz" (veja a

Figura 87). Para mudar o diretório de trabalho, pode ser utilizado cd(caminho) e fornecer o caminho onde se deseja criar o arquivo.

Figura 86: Funções pwd() e touch().

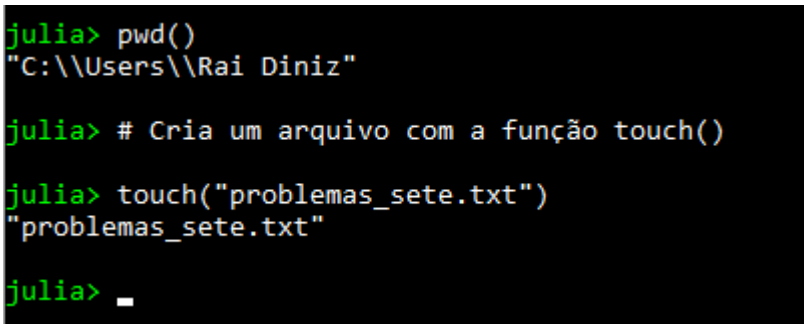
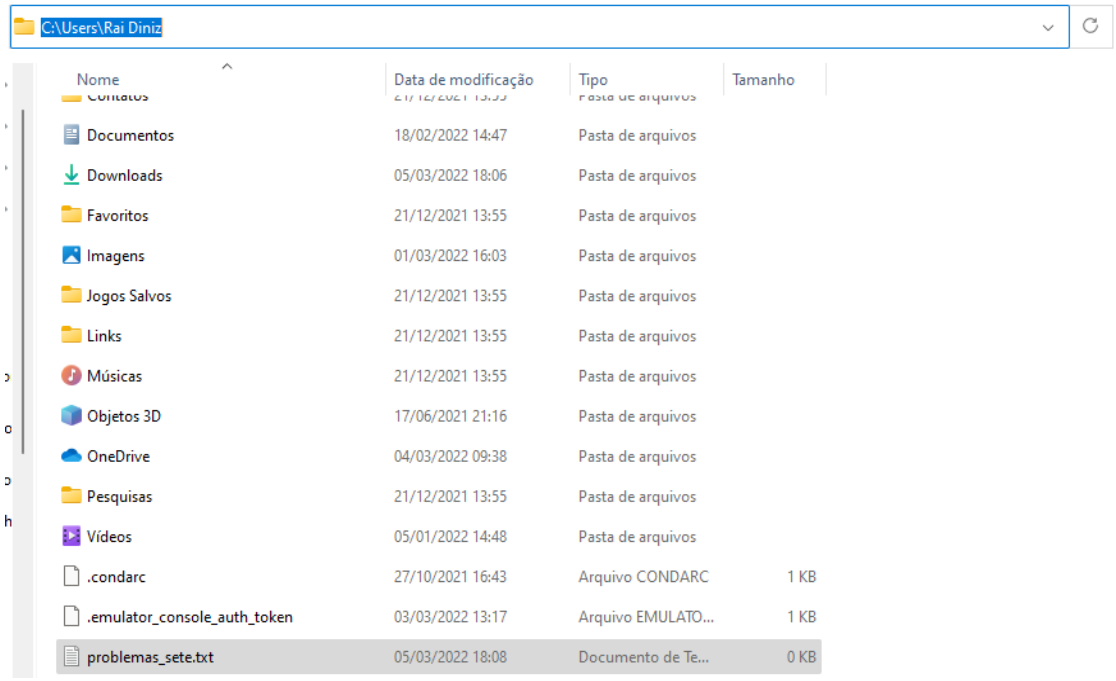
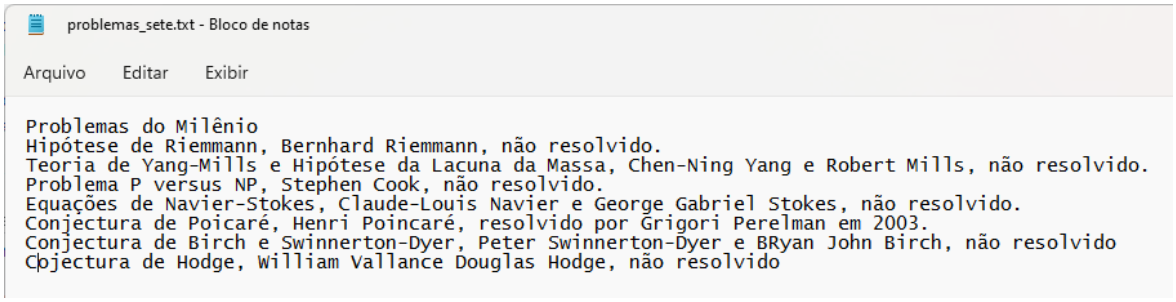


Figura 87: Arquivo criado no diretório de trabalho.



O arquivo deve ser preenchido com as informações que queremos ler pela Julia. No arquivo vamos colocar as informações sobre os sete problemas matemáticos estabelecidos pelo Instituto Clay de Matemática (por isso o nome do arquivo "problemas_sete"). O referido arquivo contém informações sobre o nome do problema, principais protagonistas do problema e se foram resolvidos ou não. Para mais informações sobre os sete problemas citados, temos a referência [4].

Figura 88: Arquivo problemas_sete.txt preenchido.



Em geral, há duas maneiras de utilizar a função open(). Método 1 (Figura 89): abra um arquivo usando open() e atribua-o a uma variável que é a instância do arquivo aberto, então use essa variável para outras operações a serem executadas no arquivo aberto. Seguidamente, feche o arquivo usando close().

Figura 89: Função open()- método 1.

```
julia> arquivo = open("problemas_sete.txt", "r")
IOStream(<file problemas_sete.txt>)

julia> # Faça alguma operação

julia> close(arquivo) # fechar o arquivo

julia>
```

No método 2 (Figura 90) é utilizado o bloco **do**. Em geral, esta é a maneira mais comum e prática de se trabalhar com arquivos em Julia. O arquivo aberto será fechado automaticamente quando o final do bloco **do** for atingido (com o comando end).

Figura 90: Função open()- método 2.

```
julia> open("problemas_sete.txt") do arquivo
       # Faça alguma operação
       end

julia>
```

A leitura de um arquivo pode ser feita de várias maneiras com o uso de funções pré-definidas em Julia. Os arquivos podem ser lidos linha por linha, na forma de strings

ou o arquivo inteiro de uma vez [3]. No primeiro caso (linha por linha) é utilizado a o função readline() (ler linha por linha), enquanto que no segundo caso é empregada a função read() (ler todas as linhas).

A Figura 91 apresenta a leitura do arquivo linha por linha. A função open() é utilizada para abertura do arquivo. Seguidamente, tem-se o laço while. Neste laço aparece a função eof(). Esta função retorna *false* até que o final do arquivo seja atingido. A função readline() ler linha por linha e armazena o conteúdo da linha na variável s. Por fim, a função println() apresenta o conteúdo da variável s na tela e pula para próxima linha na tela do REPL do Julia continuando o laço while até que seja falso o retorno da função eof (o símbolo ! em Julia significa negação).

Figura 91: Lendo linha por linha de um arquivo em Julia.

```
julia> open("problemas_sete.txt") do arquivo
       while ! eof(arquivo)
         s = readline(arquivo)
         println(s)
       end
       end
Problemas do Milênio
Hipótese de Riemmann, Bernhard Riemmann, não resolvido.
Teoria de Yang-Mills e Hipótese da Lacuna da Massa, Chen-Ning Yang e Robert Mills, não resolvido.
Problema P versus NP, Stephen Cook, não resolvido.
Equações de Navier-Stokes, Claude-Louis Navier e George Gabriel Stokes, não resolvido.
Conjectura de Poicaré, Henri Poincaré, resolvido por Grigori Perelman em 2003.
Conjectura de Birch e Swinnerton-Dyer, Peter Swinnerton-Dyer e BRyan John Birch, não resolvido
Cojectura de Hodge, William Vallance Douglas Hodge, não resolvido
```

Caso queiramos que seja lido arquivo inteiro de uma vez, podemos utilizar a função read() e para apresentar no REPL do Julia a função print(), conforme a Figura 92.

Figura 92: Lendo um arquivo completo em Julia.

```
julia> arquivo = open("problemas_sete.txt", "r")
IOStream(<file problemas_sete.txt>)

julia> s = read(arquivo, String);

julia> print(s)
Problemas do Milênio
Hipótese de Riemmann, Bernhard Riemmann, não resolvido.
Teoria de Yang-Mills e Hipótese da Lacuna da Massa, Chen-Ning Yang e Robert Mills, não resolvido.
Problema P versus NP, Stephen Cook, não resolvido.
Equações de Navier-Stokes, Claude-Louis Navier e George Gabriel Stokes, não resolvido.
Conjectura de Poicaré, Henri Poincaré, resolvido por Grigori Perelman em 2003.
Conjectura de Birch e Swinnerton-Dyer, Peter Swinnerton-Dyer e BRyan John Birch, não resolvido
Cojectura de Hodge, William Vallance Douglas Hodge, não resolvido
julia> close(arquivo)
```

7.3 TRABALHANDO COM ARQUIVOS EXCEL

A linguagem Julia fornece um pacote para manipulação de planilhas do software Excel [5] (também há outros pacotes em Julia para manipulação de planilhas do Excel, mas neste livro iremos focar no XLSX). O XLSX.jl é um pacote para ler e gravar arquivos de planilhas do Excel [6]. Para utilização, o pacote deve ser instalado no REPL do Julia. A instalação de pacotes foi abordada no capítulo 6 e a Figura 93 apresenta o processo de instalação do pacote XLSX no REPL do Julia.

Figura 93: Instalação do pacote XLSX.

```
julia> using Pkg

julia> Pkg.add("XLSX")
    Updating registry at `C:\Users\Rai Diniz\.julia\registries\General`
    Updating git-repo `https://github.com/JuliaRegistries/General.git`
   Resolving package versions...
   Installed ZipFile ─ v0.9.4
   Installed EzXML ─── v1.1.0
   Installed XLSX ──── v0.7.10
    Updating `C:\Users\Rai Diniz\.julia\environments\v1.6\Project.toml`
  [fdbf4ff8] + XLSX v0.7.10
    Updating `C:\Users\Rai Diniz\.julia\environments\v1.6\Manifest.toml`
  [8f5d6c58] + EzXML v1.1.0
  [fdbf4ff8] + XLSX v0.7.10
  [a5390f91] + ZipFile v0.9.4
Precompiling project...
  3 dependencies successfully precompiled in 8 seconds (136 already precompiled)

julia>
```

Para praticar, vamos ler o arquivo com extensão .xlsx. O arquivo contém o nome das regiões do Brasil e a quantidade de estados. O referido arquivo é apresentado na Figura 94.

Figura 94: Arquivo estado.xlsx.

A	B
Região	Quantidade de Estados
Norte	7
Nordeste	9
Sul	3
Sudeste	4
Centro-Oeste	3

A primeira etapa para a leitura de um arquivo do Excel em Julia é observar o diretório ao qual está o arquivo xlsx e direcionar o diretório de trabalho da Julia para

onde o arquivo xlsx encontra-se no seu computador (veja a função cd()). Após esta etapa, podemos ler o arquivo inteiro ou linha por linha. O pacote XLSX fornece a função readxlsx() para ler todo o conteúdo de um arquivo. A função readxlsx() retorna a dimensão de todas as planilhas do arquivo Excel (o arquivo do exemplo tinha 3 planilhas, mas somente a planilha 1 possuía os dados da Figura 94). As informações das planilhas podem ser armazenadas em uma variável e assim imprimir no REPL do Julia conforme a Figura 95. Observe que os resultados mostram também as linhas A7 e B7 através do termo "missing".

Figura 95: Lendo dados de um arquivo xlsx em Julia.

```
julia> pwd()
"C:\\Users\\Rai Diniz"

julia> cd("C:\\Users\\Rai Diniz\\Documents\\livro_Julia")

julia> using XLSX

julia> dados = XLSX.readxlsx("estados.xlsx")
XLSXFile("estados.xlsx") containing 3 Worksheets
            sheetname size          range
-------------------------------------------------
               Plan1 7x2           A1:B7
               Plan2 1x1           A1:A1
               Plan3 1x1           A1:A1
```

(a)

```
julia> P = dados["Plan1"]
7×2 XLSX.Worksheet: ["Plan1"](A1:B7)

julia> P[:]
7×2 Matrix{Any}:
 "Região"          "Quantidade de Estados"
 "Norte"         7
 "Nordeste"      9
 "Sul"           3
 "Sudeste"       4
 "Centro-Oeste"  3
 missing          missing
```

(b)

Podemos alterar os dados do arquivo xlsx. Neste caso devemos utilizar a função openxlsx() no modo "rw", ou seja, no modo leitura e gravação. Com a ajuda de um laço podemos percorrer as células do arquivo xlsx e substituir o novo valor a ser alterado. Por exemplo, vamos alterar os números da quantidade de estados no arquivo estados.xlsx para o seu correspondente por extenso, ou seja, a palavra e não o valor numérico. Sendo assim, as células ficarão assim: B2 = "Sete", B3 = "Nove", B4 =

"Três", B5 = "Quatro" e B6 = "Três". O código completo é apresentado na Figura 96(a) e o arquivo alterado na Figura 96(b).

Figura 96: Alteração dos dados de um arquivo xlsx em Julia.

```
julia> XLSX.openxlsx("estados.xlsx", mode = "rw") do dados
           Plan = dados[1]
           Plan["B2"] = "Sete"
           Plan["B3"] = "Nove"
           Plan["B4"] = "Três"
           Plan["B5"] = "Quatro"
           Plan["B6"] = "Três"
       end
```

(a)

Região	Quantidade de Estados
Norte	Sete
Nordeste	Nove
Sul	Três
Sudeste	Quatro
Centro-Oeste	Três

(b)

Podemos inserir dados na planilha. Vamos inserir os valores numéricos na coluna "C" do arquivo estados.xlsx. O código utilizado é apresentado na Figura 97(a) e o resultado obtido na Figura 97(b).

Figura 97: Inserção de dados em um arquivo xlsx.

```
julia> dados = XLSX.readxlsx("estados.xlsx")
XLSXFile("estados.xlsx") containing 3 Worksheets
            sheetname size          range
-------------------------------------------------
               Plan1 7x2            A1:B7
               Plan2 1x1            A1:A1
               Plan3 1x1            A1:A1


julia> XLSX.openxlsx("estados.xlsx", mode = "rw") do dados
           Plan = dados[1]
           Plan["C2"] = 7;
           Plan["C3"] = 9;
           Plan["C4"] = 3;
           Plan["C5"] = 4;
           Plan["C6"] = 3;
       end
```

(a)

A	B	C
Região	Quantidade de Estados	
Norte	Sete	7
Nordeste	Nove	9
Sul	Três	3
Sudeste	Quatro	4
Centro-Oeste	Três	3

(b)

7.4 REFERÊNCIAS

[1] https://opensource.com/article/21/7/programming-read-write
https://docs.julialang.org/en/v1/base/io-network/

[2] https://www.geeksforgeeks.org/opening-and-reading-a-file-in-julia/

[3] https://acervolima.com/tratamento-de-arquivos-em-julia/

[4] Keith Devlin, “Os problemas do milênio: sete granes enigmas matemáticos do nosso tempo”, 2. Ed. São Paulo: Editora Record, 2008.

[5] https://www.geeksforgeeks.org/working-with-excel-files-in-julia/

[6] https://felipenoris.github.io/XLSX.jl/stable/

8. GRÁFICOS

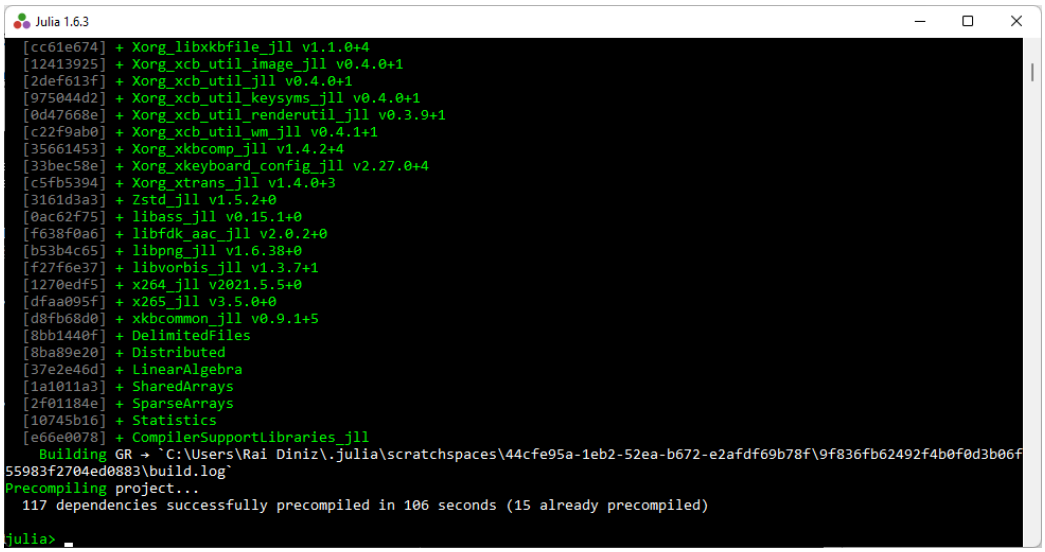
O livro não ficaria completo sem um capítulo sobre gráficos. Os gráficos são uma ferramenta importante em diversas áreas da ciência, pois facilita a visualização dos resultados e de dados. Neste capítulo serão apresentados alguns exemplos simples com pacotes gráficos disponíveis para linguagem Julia.

8.2 PACOTE PLOTS.jl

Julia possui um pacote denominado "Plots" que possui várias bibliotecas de gráficos denominadas de "backend", como exemplo, PyPlot, GR, UnicodePlots, PlotlyJS, dentre outras. Para instalar o pacote Plots.jl abra o REPL da Julia e digite na seguinte ordem: "usingPkg", Pkg.add("Plots"). As Figuras 98 e 99 apresentam o processo de instalação do referido pacote gráfico [1]. O processo de instalação de pacotes foi abordado no Capítulo 6.

Figura 98: Instalação do Pacote Plots.

Figura 99: Instalação com sucesso do Pacote Plots.



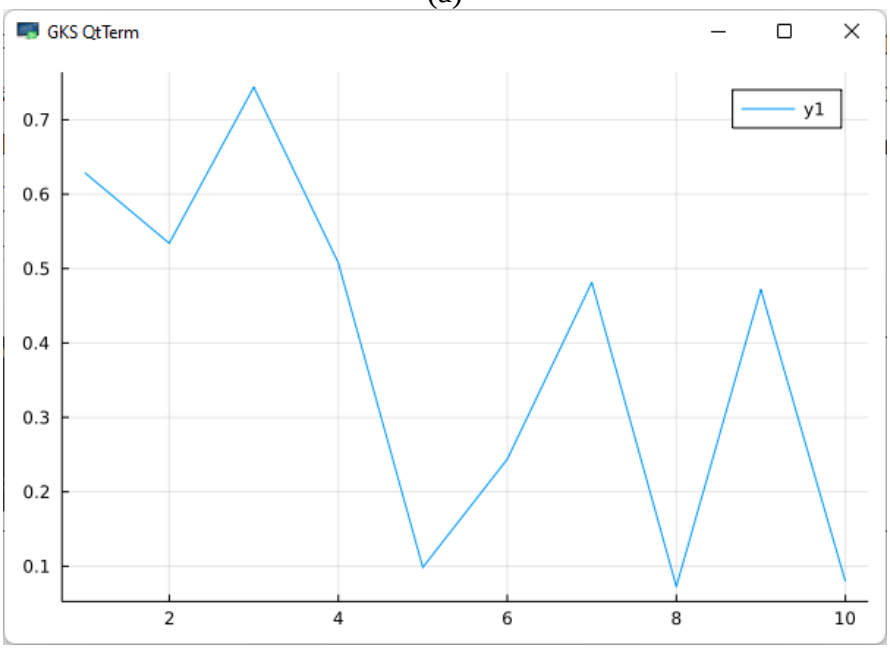
Inicialmente, quando começarmos a utilizar o pacote Plot.jl, estamos usando o backend padrão, o GR (não há necessidade de instalar este backend, pois ele é o padrão do pacote Plots.jl). Entretanto, podemos utilizar outro backend e isso depende do usuário. Para instalar outros backend basta utilizar o padrão da linguagem Julia (Pkg.add("BackendPackage")) [2].

Um simples exemplo é um gráfico de linhas. As Figuras 100(a) e (b) apresentam o código no REPL e o resultado fornecido pelo pacote Plots do Julia, respectivamente. No referido código temos 10 pontos (vetor x) e 10 pontos escolhidos aleatoriamente entre 0 e 1 armazenados no vetor y. A linha do código, plot(x,y), traça os pontos do vetor x e do vetor y em um gráfico 2D.

Figura 100: Usando o pacote Plots e função plot().

```
julia> using Plots

julia> x = 1:10;

julia> y = rand(10);

julia> plot(x,y)
```

(a)



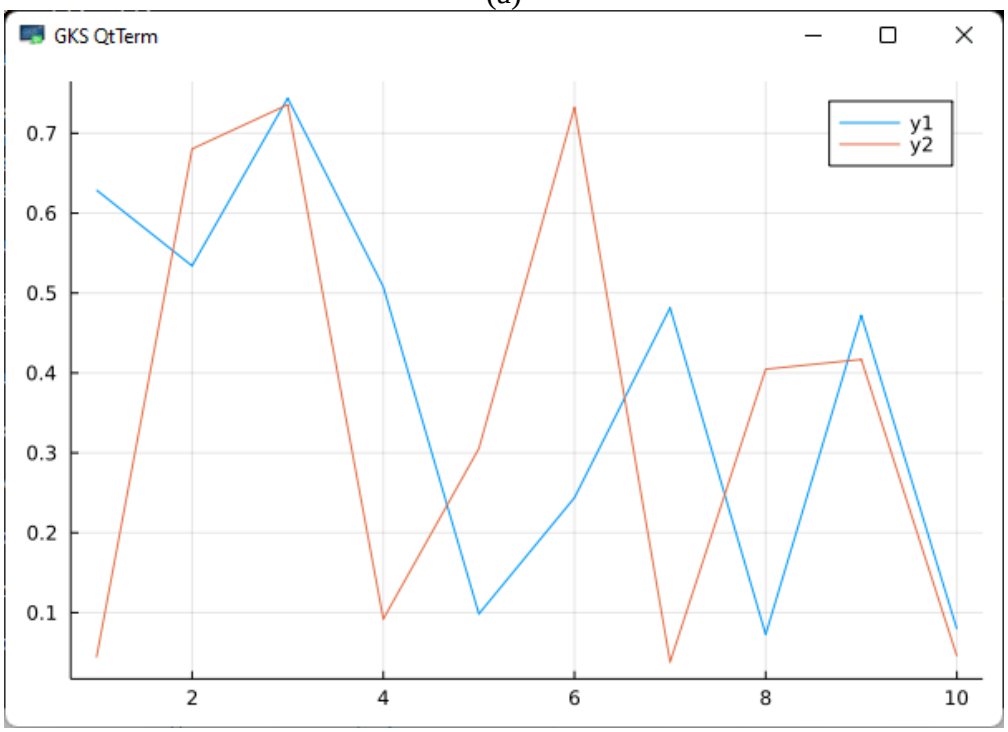
(b)

Além disso, podemos adicionar mais linhas alterando o comando plot. Isso é feito utilizando plot!(). Vamos adicionar outra linha ao nosso gráfico anterior. O novo código e gráfico é apresentado nas Figuras 101(a) e (b).

Figura 101: Função plot() com dois gráficos.

```
julia> using Plots

julia> x = 1:10;

julia> y = rand(10);

julia> plot(x,y)

julia> z = rand(10);

julia> plot!(x,z)
```

(a)



(b)

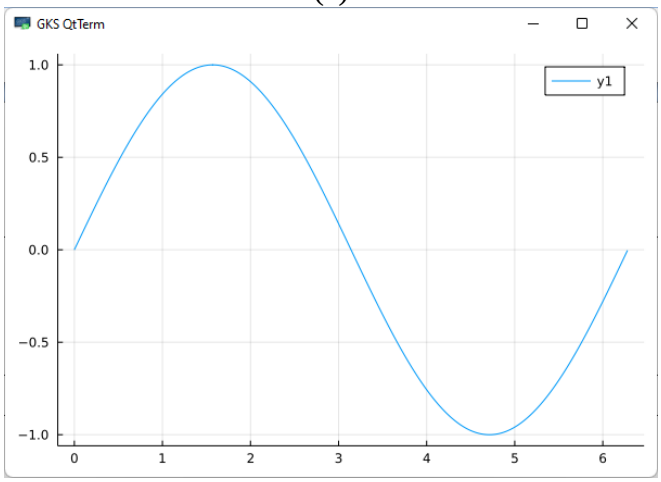
Há a possibilidade de traçar o gráfico de uma senóide diretamente na função plot (veja o exemplo das Figuras 102(a) e (b)). Também podemos salvar o gráfico em formato de imagem. No caso, a senóide foi salva em png como apresenta o código da Figura 102(a). A imagem senoide.png estará no diretório de trabalho do REPL.

Para traçar o gráfico de uma função qualquer, a sintaxe da função plot é plot(nome da função, limite inferior: passo: limite superior) ou plot(nome da função, limite inferior, limite superior). As Figuras 103(a) e (b) apresenta o código e o gráfico de uma função quadrática (ou do segundo grau), respectivamente. O mesmo gráfico poderia ser traçado usando plot(f, -3, 5) [3].

Figura 102: Traçando uma senóide com a função plot().

```
julia> using Plots

julia> plot(sin, 0:0.01:2*pi)

julia> savefig("senoide.png")

julia>
```

(a)

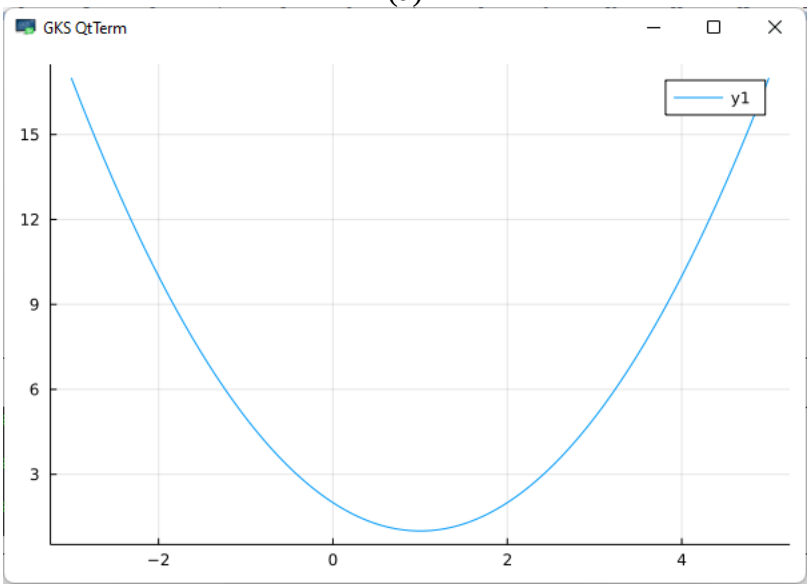


(b)

Figura 103: Traçando uma parábola com a função plot().

```
julia> using Plots

julia> f(x) = x^2-2x+2;

julia> plot(f, -3:0.01:5)
```

(a)



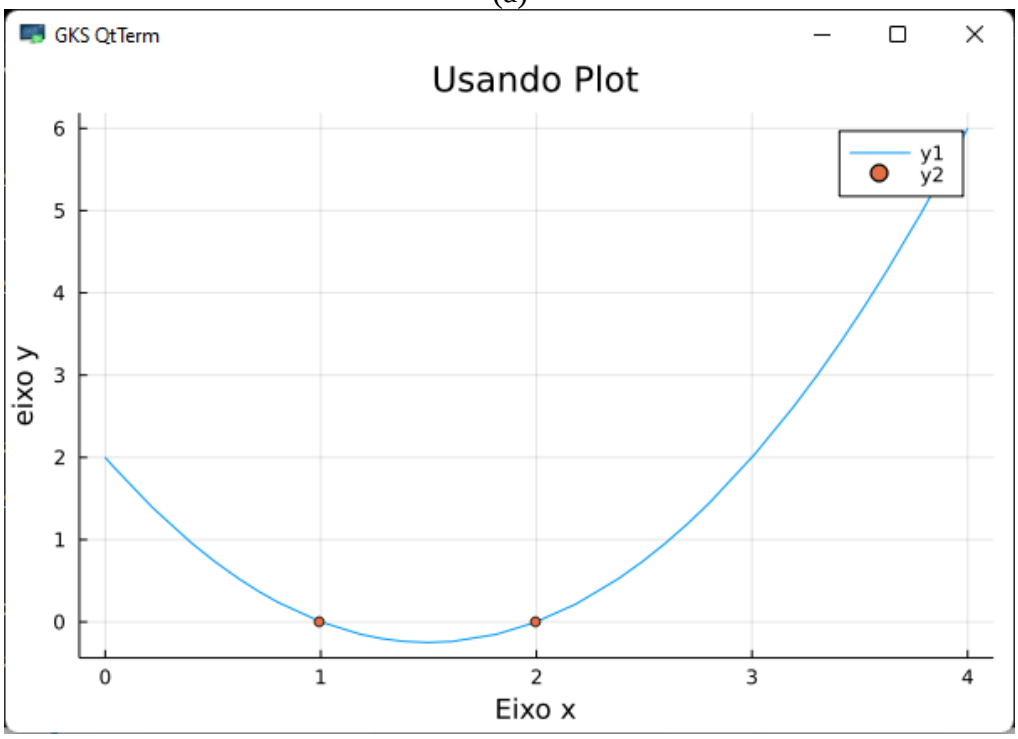
(b)

Pode-se adicionar pontos no gráfico com o comando scatter!. Simplesmente, colocam-se os valores das variáveis dentro de um colchete. Por exemplo, o polinômio x^2-3x+2 tem raízes 2 e 1, podemos colocar estes pontos do gráfico através do código da Figura 104(a). Os nomes dos eixos podem ser inseridos atribuindo xlabel = "nome do eixo x" e ylabel = "nome do eixo y". Em geral, podemos colocar atributos na função plot utilizando a sintaxe: plot!(atributo = "nome para o atributo"). Por exemplo, o título do gráfico pode ser inserido plot!(title = "Título do Gráfico", como apresenta o exemplo das Figuras 105 (a) e (b). A lista completa de atributos é encontrada na referência [4].

Figura 104: Função plot() e scatter.

```
julia> f(x) = x^2-3x+2;

julia> raizes = [1, 2];

julia> plot(f, 0, 4)

julia> scatter!(raizes, [0, 0])
```

(a)

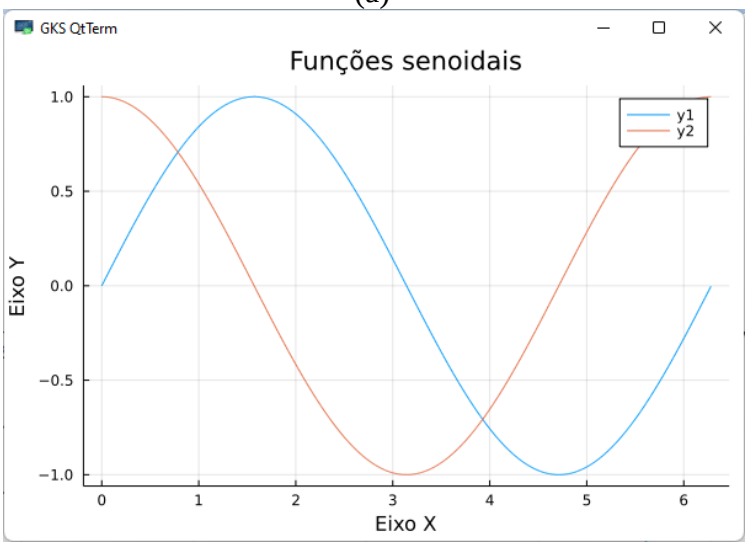


(b)

Figura 105: Usando atributos em Plots.

```
julia> using Plots

julia> x = 0:0.01:2*pi;

julia> y = sin.(x);

julia> z = cos.(x);

julia> plot(x,y)

julia> plot!(x,z)

julia> plot!(title = "Funções senoidais")

julia> plot!(ylabel = "Eixo Y")

julia> plot!(xlabel = "Eixo X")
```

(a)



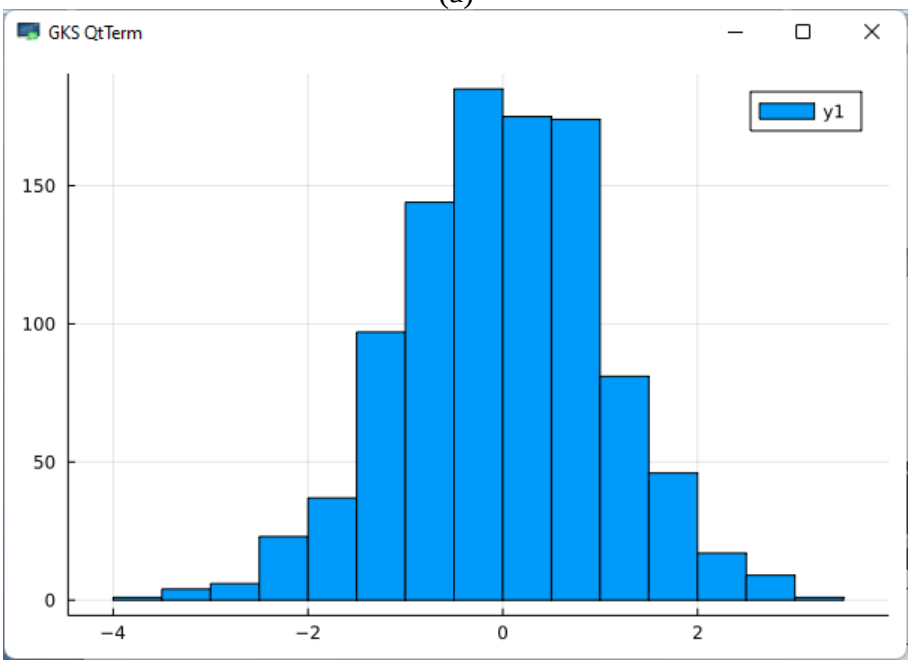
(b)

No GR do pacote Plots também é possível traçar histogramas. Para isso, vamos usar um simples exemplo. Em um vetor iremos armazenar 1000 números aleatórios normalmente distribuídos com média 0 e desvio padrão 1 (função randn). Utilizando a função histogram, iremos apresentar o vetor r em um histograma (Veja as Figuras 106 (a) e (b)).

Figura 106: Função histogram().

```
julia> r = randn(1000);

julia> histogram(r)
```

(a)



(b)

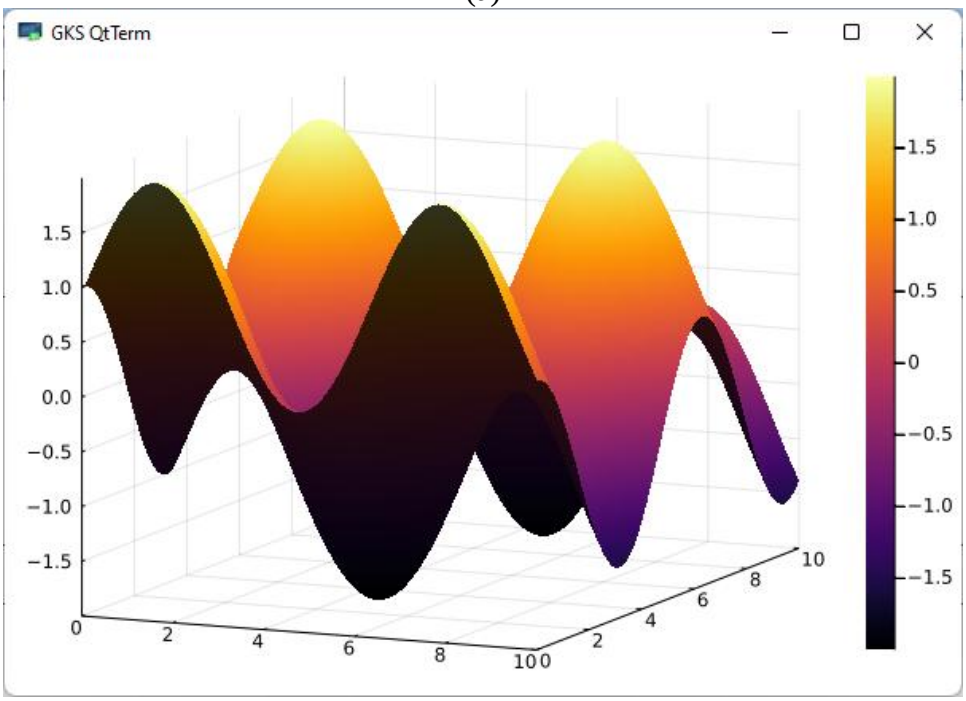
8.3 GRÁFICOS 3D

Os gráficos em três dimensões (3D) podem ser traçados no backend GR. A documentação completa do GR em Julia (também em Python e C) pode ser encontrada em [5]. O backend GR possui a função surface() para traçar gráficos 3D. A sintaxe é surface(px, py, pz, option:int) onde px é um vetor de pontos do eixo X, py é um vetor de pontos do eixo Y, pz é uma matriz adequadamente dimensionada igual a length(px)*length(py) do eixo Z e option são as opções de exibição do gráfico. Os pontos pz podem ser calculados através da definição de uma função. Por exemplo, vamos traçar o gráfico da função f(x,y) = sen(x)+cos(y). Primeiramente devemos digitar using Plots e depois f(x,y) = sin(x)+cos(y) no REPL do Julia, conforme a Figura 107(a). Seguidamente, vamos digitar surface(0:0.1:10, 0.0.1:10, f) e quando apertar a tecla "enter" vai aparecer o gráfico da Figura 107(b).

Figura 107: Gráfico 3D com a função surface().

```
julia> using Plots

julia> f(x,y) = sin(x)+cos(y)
f (generic function with 1 method)

julia> surface(0:0.1:10, 0:0.1:10, f)
```

(a)



(b)

O backed GR passou por várias melhorias e inclusão de várias rotinas e módulos. Para utilizar essas novas funcionalidades é necessário adicionar o framework GR no REPL (veja Figura 108). Um módulo interessante do GR, denominado de JLGR, tem muitas similaridades com funções gráficas residentes do MatLab.

Figura 108: Instalação da nova versão do GR.

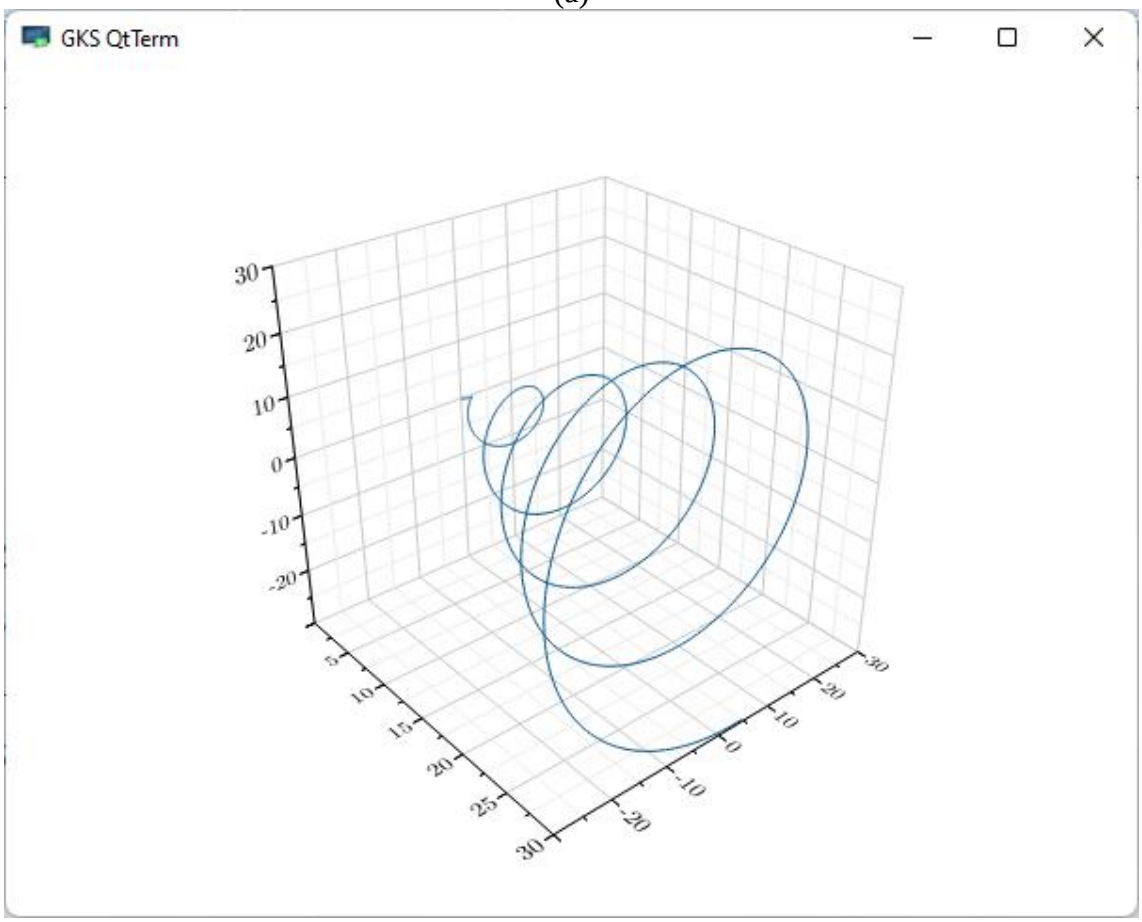
```
julia> using Pkg

julia> Pkg.add("GR_jll")
    Updating registry at `C:\Users\Rai Diniz\.julia\registries\General`
    Updating git-repo `https://github.com/JuliaRegistries/General.git`
   Resolving package versions...
   Installed GR_jll ─ v0.64.3+0
  Downloaded artifact: GR
    Updating `C:\Users\Rai Diniz\.julia\environments\v1.6\Project.toml`
  [d2c73de3] + GR_jll v0.64.3+0
    Updating `C:\Users\Rai Diniz\.julia\environments\v1.6\Manifest.toml`
  [d2c73de3] ↑ GR_jll v0.64.0+0 ▮ v0.64.3+0
Precompiling project...
  ▮ GR
  ▮ Plots
  3 dependencies successfully precompiled in 87 seconds (136 already precompiled)
  2 dependencies precompiled but different versions are currently loaded. Restart julia to access the new versions
```

Com a ajuda do JLGR vamos poder traçar gráficos 3D utilizando a função plot3(). O código e um gráfico em formato de espiral 3D são apresentados nas Figuras 109(a) e 109(b), respectivamente. A função LinRange (sintaxe: LinRange(lb, ub, len)) cria uma coleção de elementos numéricos espaçados linearmente entre o início (lb) e o fim (ub). O tamanho do espaçamento é controlado pelo parâmetro "len".

Figura 109: Função plot3().

```
julia> using GR

julia> x = LinRange(0, 30, 1000);

julia> y = cos.(x) .* x;

julia> z = sin.(x) .* x;

julia> plot3(x, y, z)
```

(a)



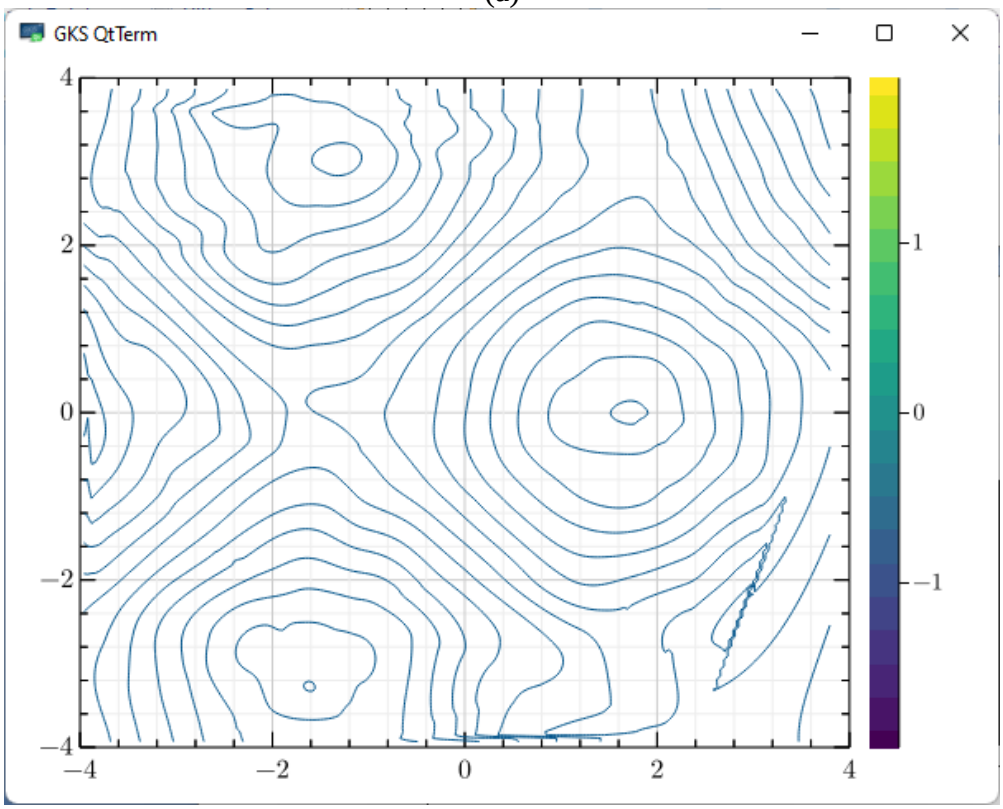
(b)

No JLGR existe a função contour(), cuja finalidade é traçar gráfico de contornos (curvas de níveis). Vamos a um exemplo apresentado nas Figuras 110(a) e 110(b).

Figura 110: Função contour().

```
julia> x = 8 .* rand(100) .- 4;

julia> y = 8 .* rand(100) .- 4;

julia> z = sin.(x) .+ cos.(y);

julia> contour(x, y, z)
```

(a)



(b)

8.4 OUTROS GRÁFICOS

No GR() (no JLGR) existe a opção de plotar (traçar) outros gráficos. A seguir iremos apresentar os gráficos e seus respectivos códigos.

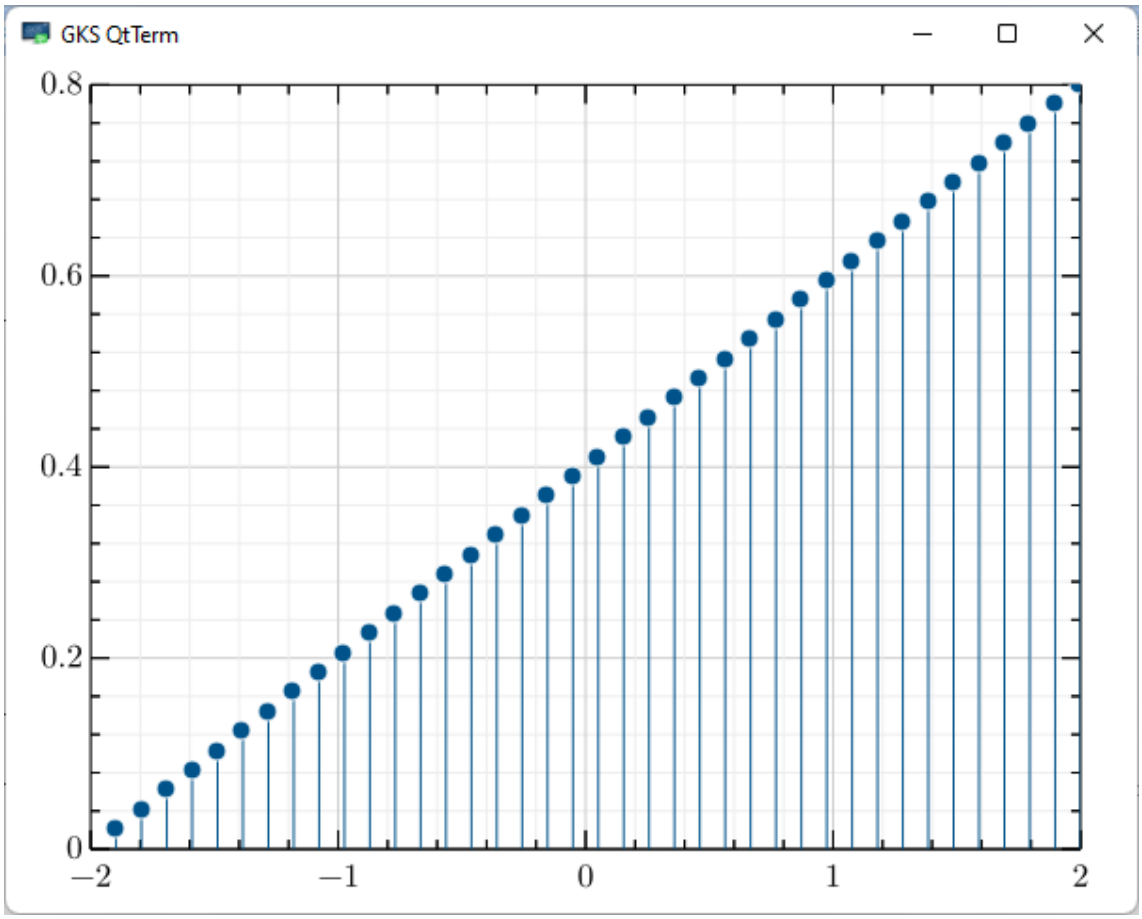
Função STEM()

O comando stem() é utilizado para traçar gráficos em tempo discreto. Vamos plotar o gráfico discreto da reta y = 0.2*x+0.4 limitado em -2≤x≤2. As Figuras 111(a) e 111(b) apresenta o código e o gráfico utilizando a função stem(), respectivamente.

Figura 111: Função stem().

```
julia> using GR

julia> x= LinRange(-2, 2, 40);

julia> y = 0.2 .* x .+ 0.4;

julia> stem(x,y)
```

(a)



(b)

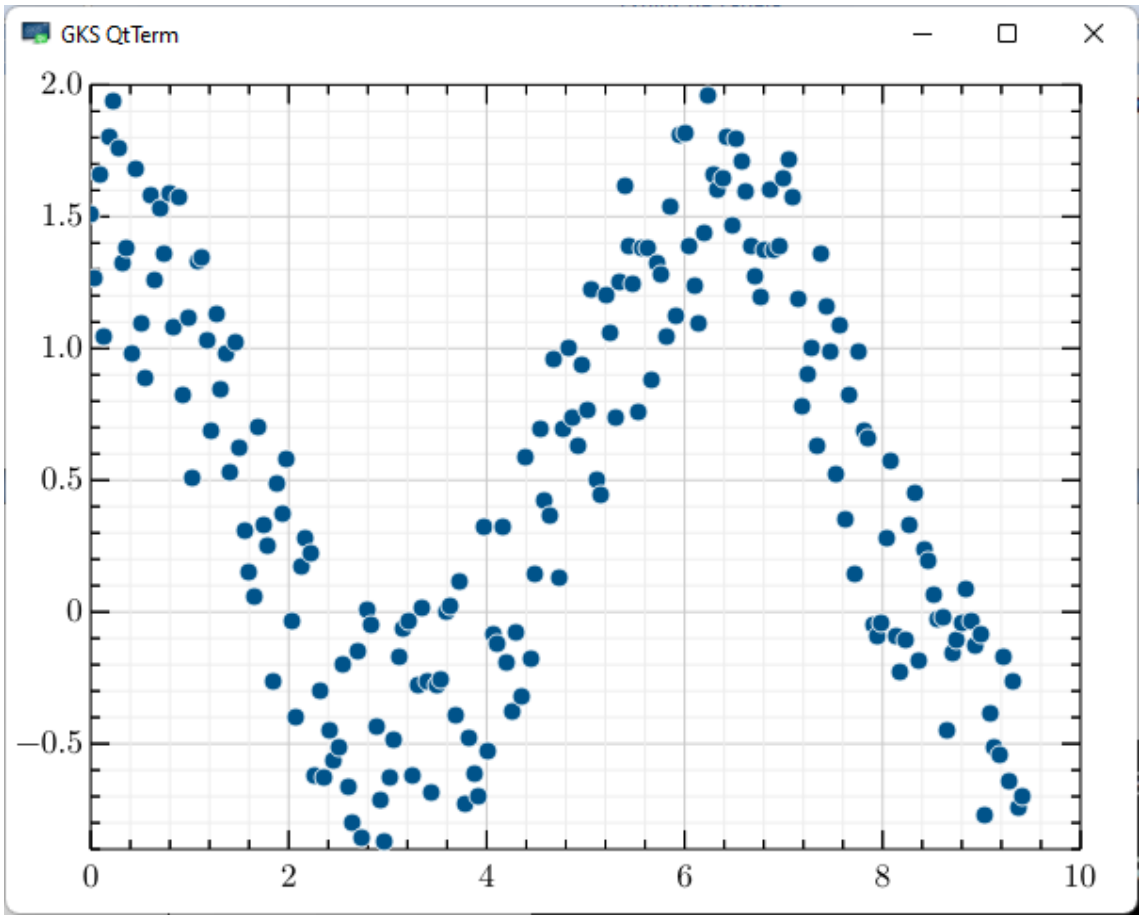
Função SCATTER()

O gráfico de dispersão (scatter) pode ser obtido em Julia através da função scatter(x,y, outros argumentos). Os outros argumentos nos dados de entrada da função scatter referem-se à mudança de cor e tamanho dos marcadores que podem ser alterados a depender do usuário. Um exemplo de utilização da função scatter com código e gráfico é mostrado nas Figuras 112(a) e 112(b), respectivamente.

Figura 112: Função scatter().

```
julia> x = LinRange(0, 3*pi, 200);

julia> y = cos.(x) + rand(200);

julia> scatter(x, y)
```

(a)



(b)

8.5 REFERÊNCIAS

[1] https://julialang.github.io/PackageCompiler.jl/dev/

[2] https://docs.juliaplots.org/latest/tutorial/

[3] http://mth229.github.io/graphing.html

[4] https://docs.juliaplots.org/latest/generated/attributes_plot/

[5] https://gr-framework.org/index.html

9. INTERFACE GRÁFICA DO USUÁRIO (GUI)

9.1 INTRODUÇÃO

As interfaces gráficas do usuário (GUI- *Graphic User Interface*) são muito comuns no uso de softwares em geral e os programadores devem estar aptos a trabalhar na criação de GUI, pois o uso das referidas interfaces torna a manipulação mais fácil além de aumentar a produtividade do usuário final [1]. O uso de GUI em Julia ainda é recente e, portanto, não é um ponto forte da referida linguagem de programação. Neste contexto, ao contrário das Linguagens C# e Java, por exemplo, a noção de GUI em Julia ainda precisa amadurecer bastante. Há alguns pacotes que existem em Julia com o objetivo de criação de GUI, mas estes pacotes ainda precisam de muitas melhorias para se chegar a um nível de competitividade comparado com outras linguagens de programação.

9.2 PACOTES

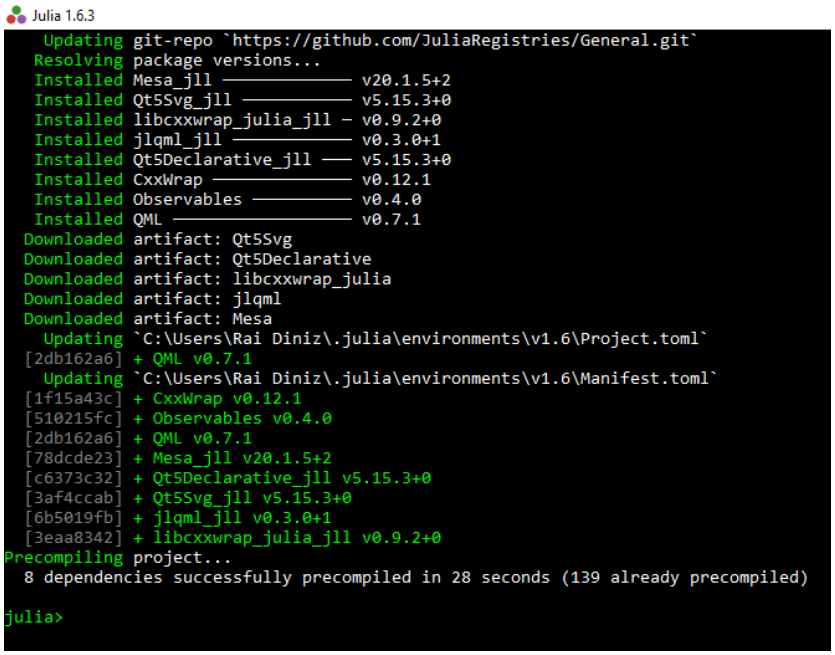
A Linguagem Julia possui alguns pacotes que podem ser úteis na criação de GUI. Existe o *Genie* que podemos criar aplicação web (é comparável ao Django em Python). O referido pacote é funcional com alguns exemplos interessantes, mas a documentação não é completa. O JGUI é um simples pacote para criação de GUI. O uso é fácil e limitado. O *Makie* é um ecossistema de visualização de dados para Julia, possui alto desempenho e extensibilidade. O QML é um pacote muito utilizado e possui uma vasta documentação na internet. Por fim, existe o pacote Gtk.jl ao qual fornece uma ligação com a biblioteca Gtk (https://www.gtk.org/). A biblioteca Gtk contém ferramentas para criação de interfaces gráficas e elementos de controle gráfico (denominados de *widgets*). Uma lista de pacotes para GUIs em Julia pode ser acessada em [2].

9.3 QML

O QML (Qt Meta Linguagem ou *Qt Modelling Language*) é uma linguagem declarativa para criação de interfaces gráficas e é parte do toolkit Qt. O Qt é um toolkit para desenvolvimento multiplataforma de sistemas e criado há mais de duas décadas. As vantagens de utilização do QML são baixa curva de aprendizagem, alta produtividade, expressividade na construção de interfaces gráficas para tablets e smartphones, alto desempenho em função do suporte à execução em GPUs e facilidade de integração com linguagens de programação como, por exemplo, JavaScript e C++ [3].

A instalação do QML em Julia é similar aos demais pacotes, conforme Capítulo 6. Utilizando o comando Pkg.add("QML") é possível instalar o QML, segundo a Figura 113.

Figura 113: Instalação do QML em Julia.



Para usar o QML em Julia, é interessante instalar o pacote Observable (ver Figura 114). Com o referido pacote é possível que algumas aplicações sejam atualizadas interativamente. Também, os pacotes Qt5QuickControls_jll e Qt5QuickControls2_jll são úteis nas aplicações com QML em Julia como apresentam as Figuras 115 e 116, respectivamente.

Figura 114: Instalação do pacote Observable em Julia.

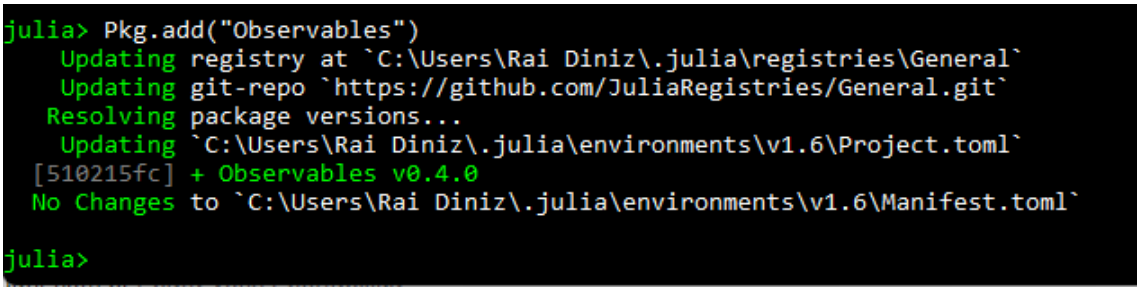


Figura 115: Instalação do pacote Qt5QuickControls_jll em Julia.

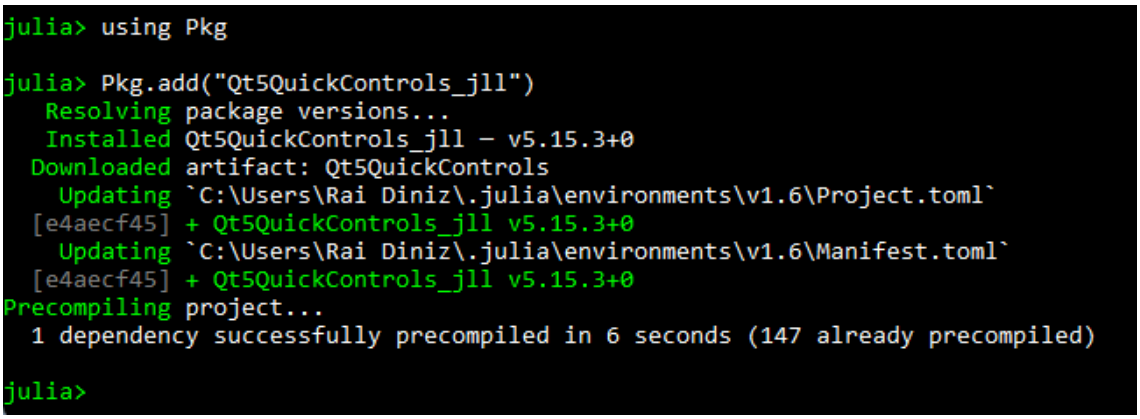
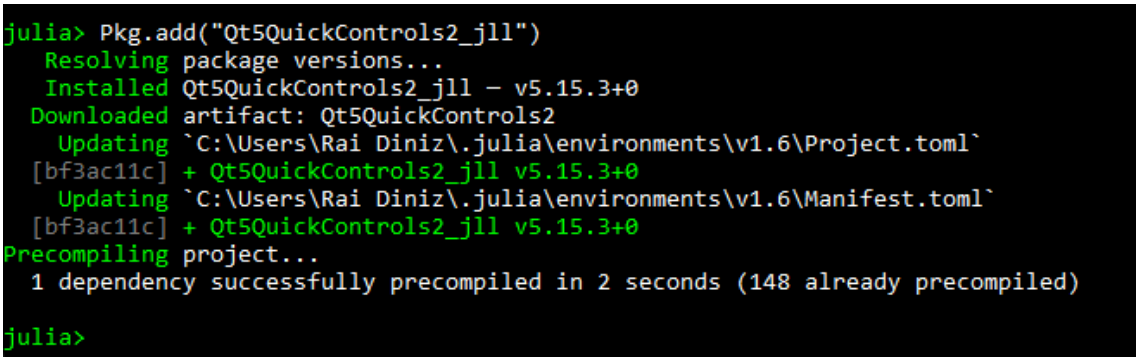


Figura 116: Instalação do pacote Qt5QuickControls2_jll em Julia.



Após a instalação dos pacotes necessários para implementação e execução dos códigos, devemos implementar o código em Julia e em QML. Vamos iniciar com um simples exemplo: implementar uma interface gráfica com uma janela e um botão. Primeiro devemos ter o código em QML. O referido código, apresentado na Figura 117, será carregado no REPL do Julia utilizando a função include (ver código da Figura 118). O resultado é apresentado na Figura 119. A função exec() executa a aplicação, mas podemos usar a função exec_async() que faz uma execução assíncrona (ou seja, o REPL do Julia "passa" para a próxima linha após fechar a GUI com exec()).

Figura 117: Código em QML do simples exemplo de uma janela e um botão.

Teste_Button.qml - Bloco de notas

Arquivo Editar Exibir

```
import QtQuick 2.15
import QtQuick.Window 2.15
import QtQuick.Controls 2.15

Window {
    width: 320
    height: 240
    visible: true
    title: qsTr("Teste")

    Button {
        id: testButton
        anchors.centerIn: parent
        text: qsTr("Clicar Aqui")
    }
}
```

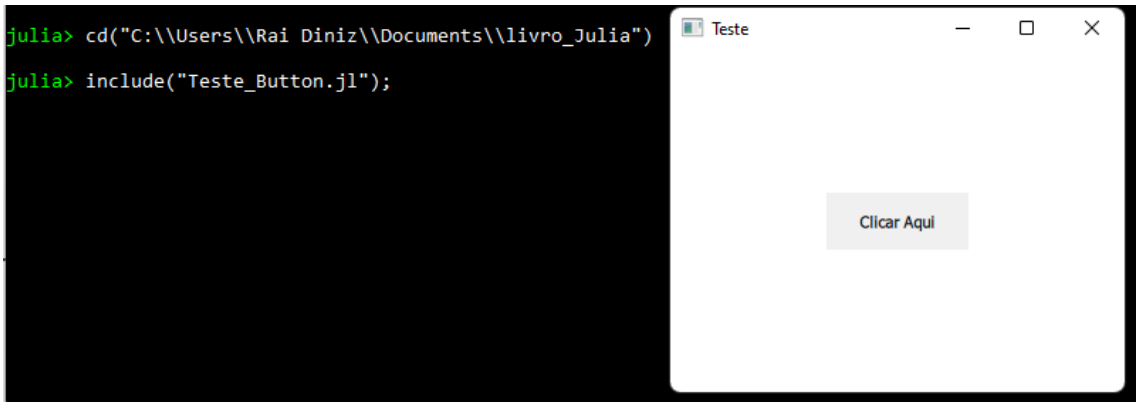
Figura 118: Código em Julia do simples exemplo de uma janela e um botão.

Teste_Button.jl - Bloco de notas

Arquivo Editar Exibir

```
using QML
using Qt5QuickControls2_jll


loadqml("Teste_Button.qml")

# Comentário
# exec_async()

exec()
```

Figura 119: Resultado da execução no REPL do simples exemplo de uma janela e um botão.



O próximo exemplo foi retirado de [4] (mas, com algumas alterações, pois erros estavam sendo detectados devido à incompatibilidade de versões do Julia e QML). É um exemplo de uma barra deslizante. Os códigos em Julia e QML do exemplo da barra deslizante são mostrados nas Figuras 120 e 121, respectivamente. Os dois arquivos slider.jl e slider.qml foram salvos no diretório C:\Users\Rai Diniz\Documents\livro_Julia.

Figura 120: Código Júlia da barra deslizante.

slider.jl - Bloco de notas

Arquivo Editar Exibir

```
using QML
using Observables
using Qt5QuickControls2_jll

slidervalue = Observable(0.0)

loadqml(
  "slider.qml",
  variables = JuliaPropertyMap(
    "slidervalue" => slidervalue
  )
)

exec_async()
```

Figura 121: Código QML da barra deslizante.

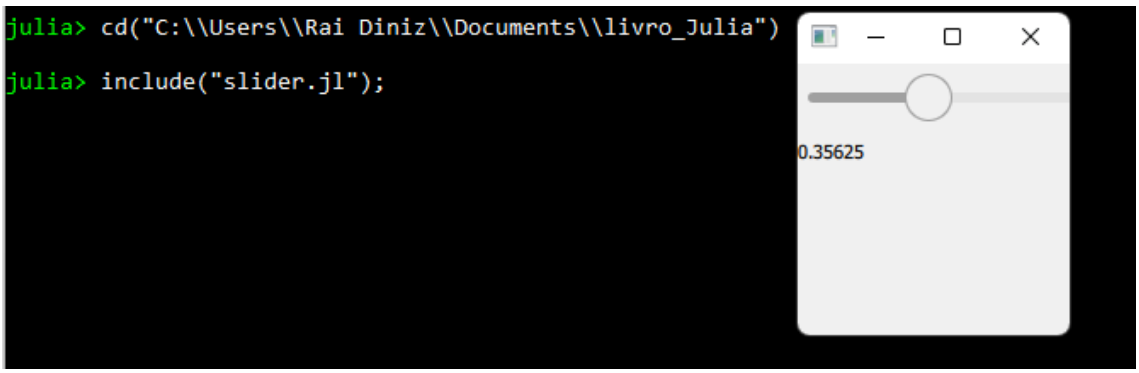
slider.qml - Bloco de notas

Arquivo Editar Exibir

```
import QtQuick 2.6
import QtQuick.Controls 2.3
import QtQuick.Layouts 1.0

ApplicationWindow {
  visible: true
  onClosing: Qt.quit()

  ColumnLayout {
    Slider {
      value: variables.slidervalue
      onValueChanged: {
        variables.slidervalue = value
      }
    }

    Text {
      text: variables.slidervalue
    }
  }
}
```

Ao executar o arquivo slider.jl temos o resultado da Figura 122. Uma janela é aberta e podemos deslocar o botão e alterar a variável slidervalue.

Figura 122: GUI da barra deslizante com QML e Julia.

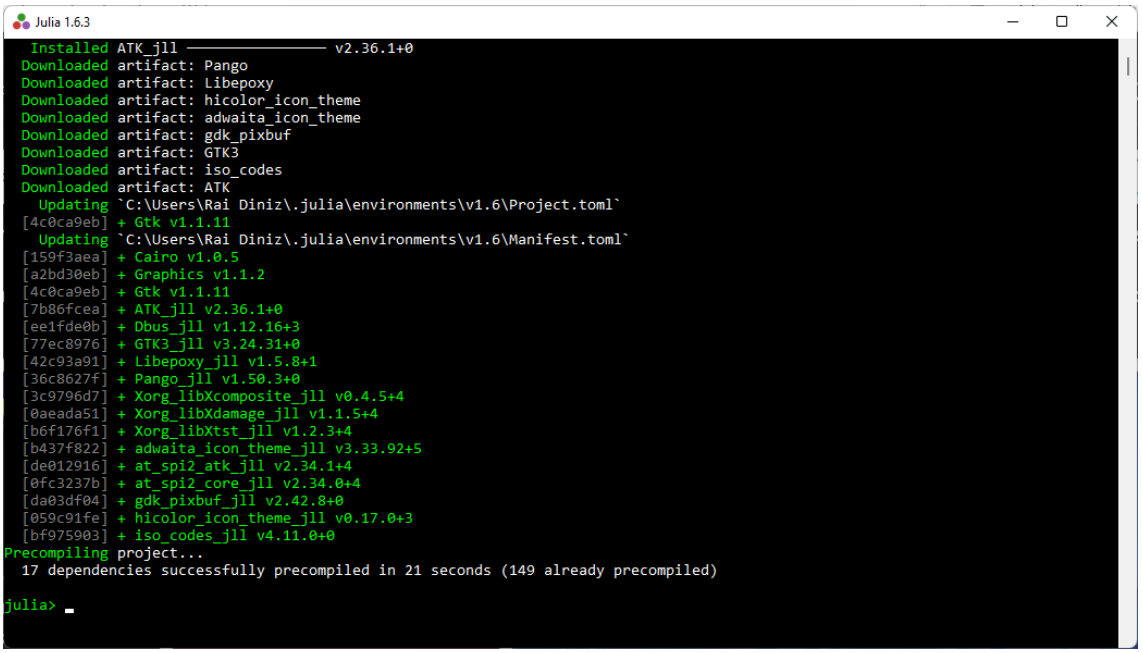


Uma desvantagem da utilização do QML é que o usuário deve ter o código do programa em QML, ou seja, o usuário deve ter conhecimento da "linguagem" QML. Entretanto, existe um ambiente visual denominado de Qt Creator (dentro do Qt Quick) para desenvolver interfaces gráficas com QML. Há boas referências na internet sobre a referida linguagem, como por exemplo, veja a referência [5].

9.4 QTK

A instalação do pacote Qtk.jl é direta e usando o comando Pkg. A Figura 123 apresenta a tela do RELP do Julia após o término da instalação do pacote Qtk.jl.

Figura 123: Instalação do QTK em Julia.



Como primeiro exemplo, vamos construir uma janela vazia de tamanho 400x200 pixels e adicionar um botão. Inicialmente, o pacote Gtk é carregado utilizando o comando using do Julia [6]. Seguidamente, uma janela é criada usando o comando GtkWindow do pacote Gtk. O referido comando tem como argumento de entrada o título da janela, a largura da janela e a altura da janela. Depois, um botão é criado usando o comando GtkButton. Para inserir o botão na janela chamamos o comando push!. Finalmente, o comando showall renderizará todo o aplicativo na tela. Observe que caso não coloque ponto e vírgula no final de cada comando, o REPL mostrar a sintaxe completa de cada comando, conforme a Figura 124. Para evitar o aparecimento de textos após cada comando, podemos utilizar o ponto e vírgula, conforme a Figura 125.

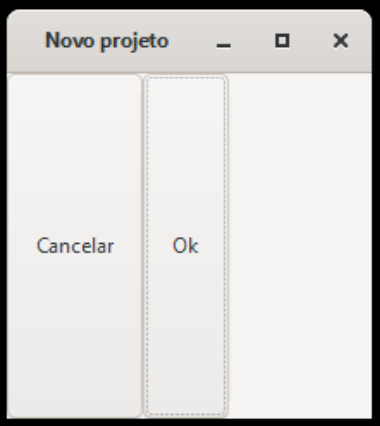
Figura 124: Usando o QTK- caso sem ponto-e-vírgula.

Figura 125: Usando o QTK- caso com ponto-e-vírgula.

Vamos para mais um exemplo com o Gtk.jl. Agora, iremos colocar dois botões na janela. Os botões serão denominados de “cancelar” e “ok”. Clicar nos referidos botões não vai acontecer nada, pois devemos alterar o código da Figura 126. O Gtk é baseado em objetos. O comando ou objeto GtkWindow é base para inserção de outros objetos. O objeto GtkBox(:,h) organiza os widgets filhos em uma área retangular. A área retangular de um GtkBox é organizada em uma única linha ou em uma única coluna, dependendo da orientação (h para horizontal e v para vertical). O comando push!, irá inserir a área retangular horizontal criada pelo GtkBox na janela criada pelo GtkWindow. O objeto Gtkbutton cria os botões cancelar e ok. Por fim, o botões são inseridos na área retangular horizontal através do comando push!. A lista de objetos e comandos completa do Gtk pode ser encontra em [7].

Figura 126: Usando o QTK- caso com dois botões.

```
julia> using Gtk

julia> win = GtkWindow("Novo projeto");

julia> hbox = GtkBox(:h);

julia> push!(win, hbox);

julia> cancelar = GtkButton("Cancelar");

julia> ok = GtkButton("Ok");

julia> push!(hbox, cancelar);

julia> push!(hbox, ok);

julia> showall(win)
```



9.4 REFERÊNCIAS

[1] https://www.devmedia.com.br/tkinter-interfaces-graficas-em-python/33956
https://juliapackages.com/c/gui

[2] https://juliapackages.com/c/gui

[3] https://qmob.solutions/portfolio-qmob-tela.pdf

[4] https://github.com/barche/juliacon2020-qml

[5] https://mimecar.gitbook.io/qt-course/en/chapter-04-qml/chapter-04-s01

[6] https://juliagraphics.github.io/Gtk.jl/latest/manual/gettingStarted/

[7] https://devdocs.io/gtk~3.20/

10. JULIA E JUPYTER

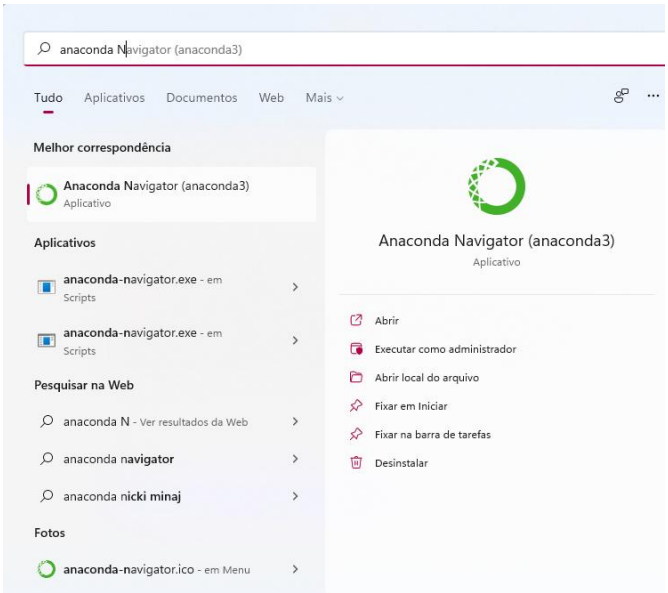
Jupyter Notebook, anteriormente denominado de IPython Notebook, é uma aplicação web que fornece um ambiente interativo voltado à ciência de dados e cálculos científicos que podem ser utilizados com uma variedade de linguagens de programação. É um software baseado na web e é de uso gratuito para todos. Os denominados notebook Jupyter são documentos nos quais os programadores podem implementar ao mesmo tempo seus códigos e textos explicativos [1].

O processo de instalação do Jupyter foi apresentado no Capítulo 1 na Seção 1.2 deste livro. Ademais, a instalação do Julia no Jupyter também foi apresentada na referida seção.

10.2 PROGRAMANDO EM JULIA COM O JUPYTER

O Jupyter pode ser inicializado de várias maneiras dependendo do sistema operacional que o programador está utilizando. Neste livro, o Jupyter e o Julia foram instalados no sistema operacional Windows. A primeira forma de inicializar o Jupyter é indo para a barra de pesquisa do Windows e digitar “Anaconda Navigator”. Este comando, de acordo com a Figura 127, acessa o aplicativo Anaconda Navigator.

Figura 127: Execução do aplicativo Anaconda Navigator.



Após a execução do aplicativo Anaconda Navigator você pode acessar o ambiente de programação Jupyter. A Figura 128 apresenta a janela Home do aplicativo Anaconda Navigator e o acesso ao Jupyter é feito clicando em “Lauch”. Após um tempo de espera, o Jupyter será inicializado no seu navegador padrão (no caso deste livro o navegador padrão é Microsoft Edge). Como curiosidade, existe ainda o JupyterLab, um ambiente de programação similar ao Jupyter Notebook, porém mais robusto. Possivelmente, no futuro o JupyterLab irá substituir o Jupyter Notebook. Como o objetivo deste livro é apresentar os fundamentos básicos da linguagem Julia, foi utilizado o Jupyter Notebook, pois será utilizado somente o navegador de arquivos e o ambiente de programação. Outra forma de acessar o Jupyter Notebook é através do CMD do Windows. Basta digitar “jupyter notebook” no prompt de comando e apertar a tecla “enter”. Por fim, uma terceira forma de acesso, é simplesmente digitar na barra de pesquisa “jupyter notebook” e executar, como apresenta a Figura 129 [2].

Figura 128: Acesso ao Jupyter Notebook pelo Anaconda Navigator.

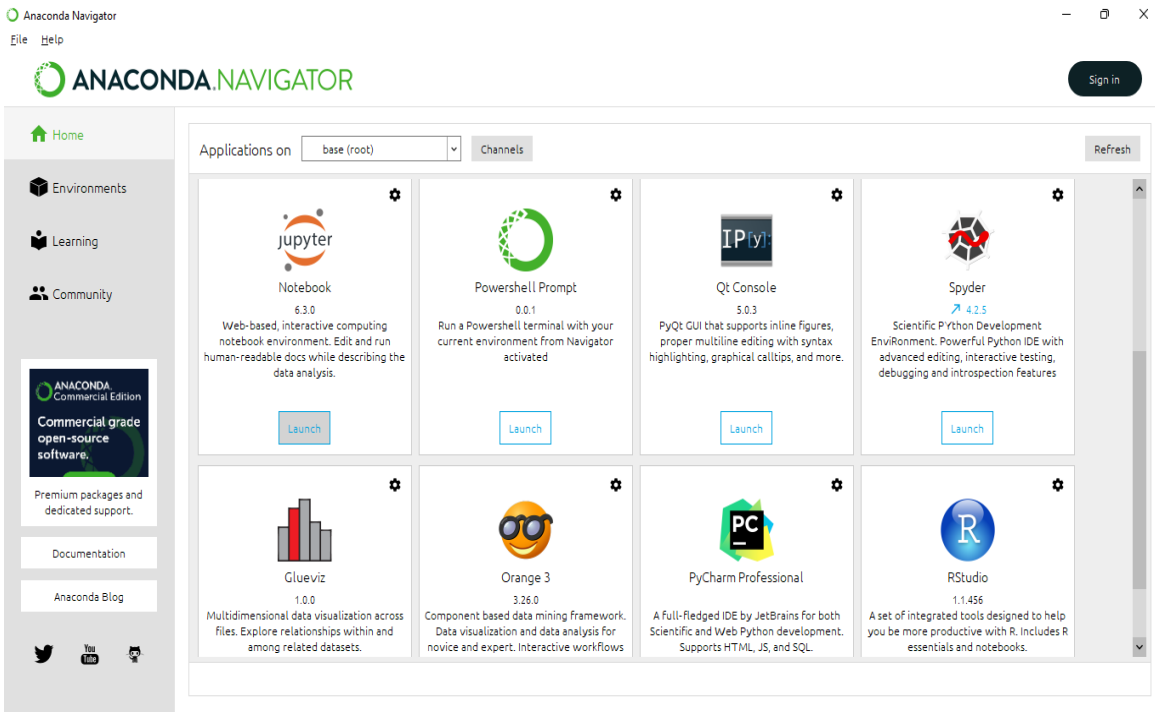
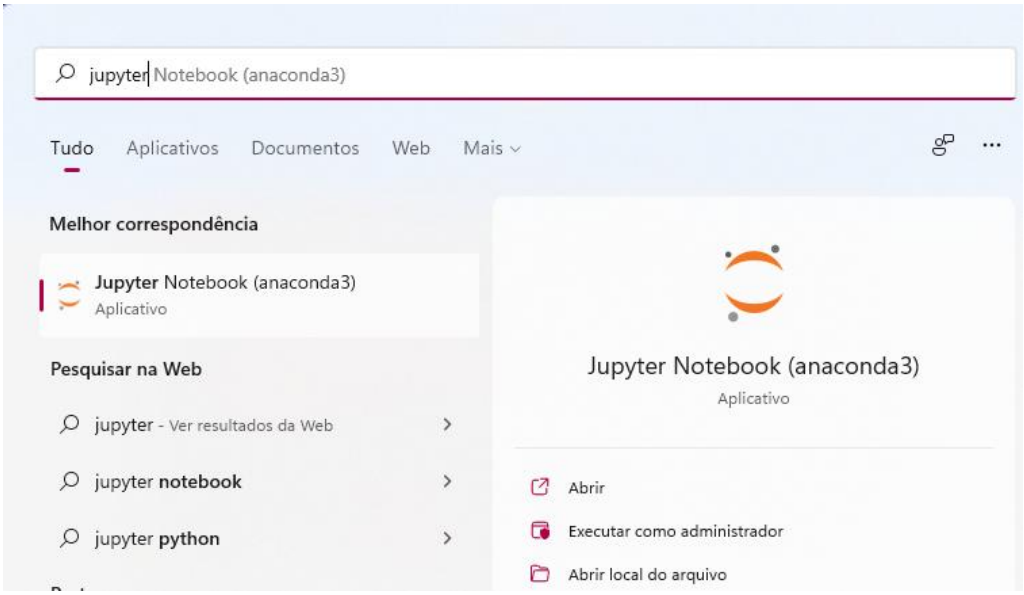


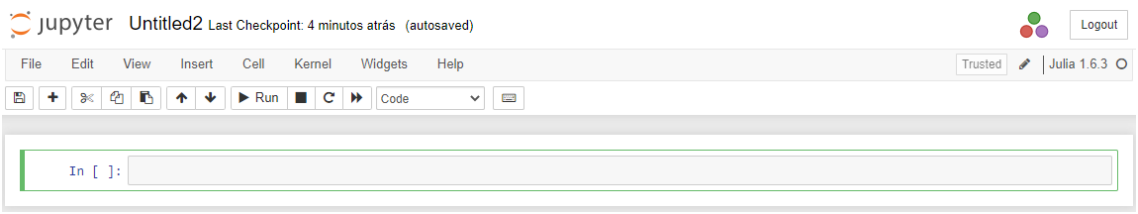
Figura 129: Acesso direto ao Jupyter Notebook.



Após a execução do Jupyter Notebook, será aberta uma janela denominada de Home Page. Nesta janela vamos acessar o Julia para criar nossos programas. Todo este processo é mostrado nas Figuras 130 e 131.

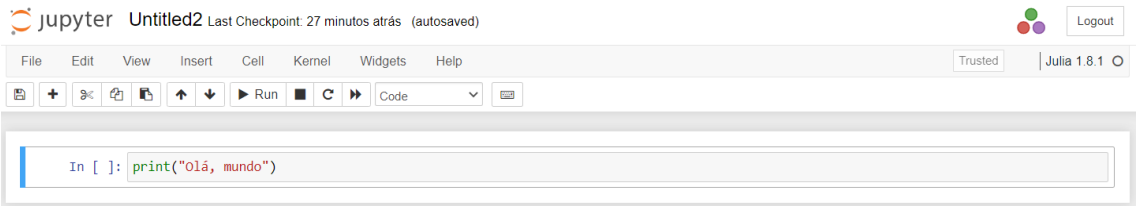
Figura 130: Home Page do Jupyter.

Figura 131: Célula de implementação do Jupyter na linguagem Julia.



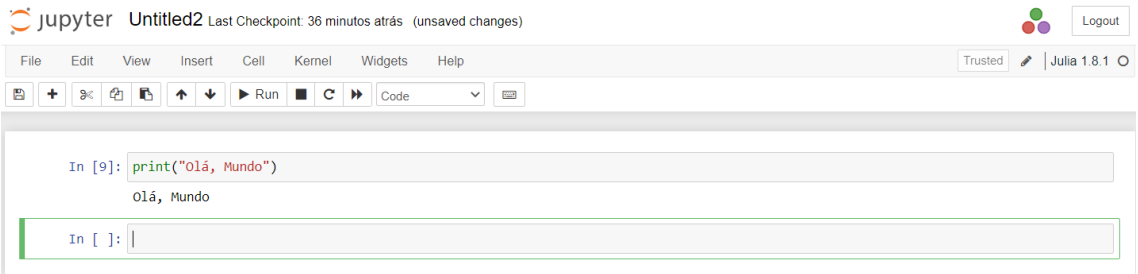
Seguindo o costume, a primeira implementação será a implementação do programa “Olá, Mundo” na tela do Jupyter. Para isto, basta digitar o comando print(), descrevendo os caracteres que deseja mostrar no parênteses e entre aspas, como apresenta a Figura 132 .

Figura 132: Implementação do código em Julia.



Para que o programa seja executado, é preciso selecionar a célula ou grupos de células (caso haja mais de uma célula, a célula selecionada apresentará uma barra azul na lateral esquerda) e selecionar o botão ▸Run, apresentado na barra superior. Também é possível executar o programa através das teclas de atalho Shift + Enter do teclado. O comando “A” do teclado adiciona uma nova célula abaixo, enquanto o comando “D+D” apaga a célula selecionada. A execução do programa e seus resultados aparecem logo abaixo da célula onde foi inserida a programação (ver Figura 133).

Figura 133: Resultado da primeira implementação.



As células de implementação possuem ordem de hierarquização, sendo assim, cada célula abaixo seguirá o código ou implementação que foi proposto na célula anterior, podendo ter seus valores afetados por este programa também.

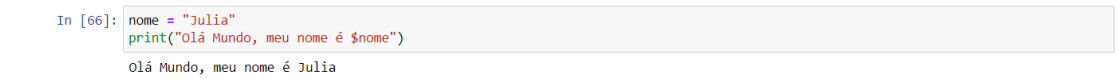
A linguagem Julia possui similaridade com a linguagem Python, como pode ser visto na Figura 133 que apresenta o programa “Olá, Mundo”. No Jupyter Notebook não há necessidade de caracteres como ponto-e-vírgula ao final de cada programa ou utilização de end ao final da programação, devido ao sistema de células já funcionarem como o comando final para cada programa.

O comando printstyle adiciona características às strings, como coloração. Para dar essa característica é preciso implementar o código printstyled(“Texto”, color =:cor desejada), vale ressaltar que as cores devem ser ditas em língua inglesa, como apresenta a Figura 134.

Figura 134: Implementação de texto com cores.

Em Julia o caractere $ é usado para inserir uma variável de qualquer tipo em uma string (texto). É necessário ter a variável e adicionar a ela um valor, como mostra a Figura 135.

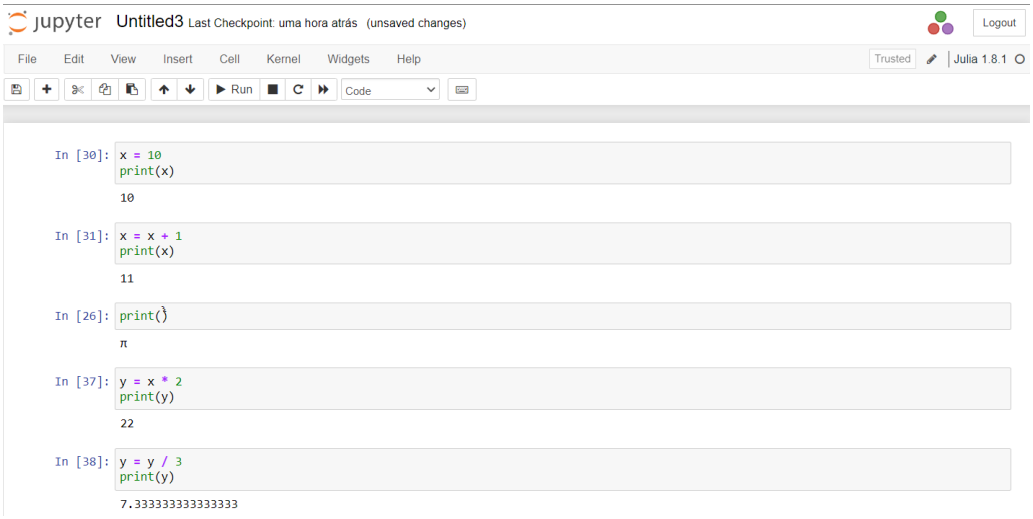
Figura 135: Implementação com uso do $.



Vetores são uma forma de agrupar uma determinada quantidade de valores em uma mesma variável, onde os valores seguem uma sequência, geralmente a de inserção. Estes valores podem ser exibidos, substituídos, alterados e realizarem operações. Os vetores são um conjunto de dados dispostos em uma única dimensão, enquanto as matrizes são dispostas em duas.

A implementação com o uso de variáveis é fundamental em toda linguagem de programação, permitindo o uso de funções algébricas e resolução de problemas. Em Julia também é aceito o uso de variáveis, onde podem ser atribuídos valores para uma unidade de armazenamento definida de maneira rápida e eficaz. A Figura 136 apresenta a inserção de uma variável, atribuindo-lhe um valor numérico e em seguida apresentando esse valor através da variável. Existem diferentes tipos de denotação de valores na linguagem Julia, sendo todas elas derivadas de um tipo maior denominado *Any*, tais tipos são: Float64, Int, Complex, AbstractFloat, Integer, Real, Number entre outros.

Figura 136: Resultado da implementação com variável e operações.

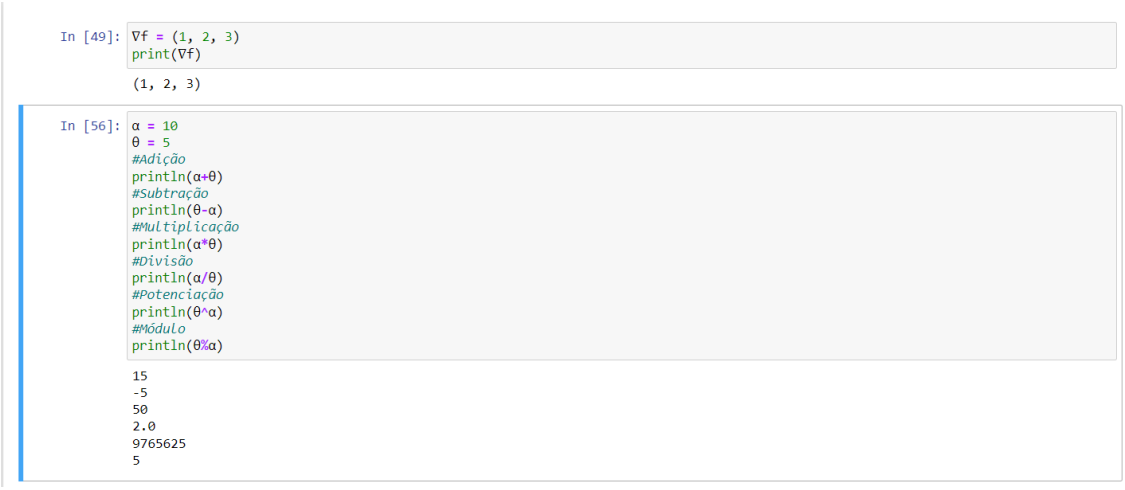


A linguagem Julia aceita caracteres como +, -, * e / as operações matemáticas de soma, subtração, multiplicação e divisão, respectivamente. Alguns exemplos são apresentados na Figura 137 programados no Jupyter. A linguagem Julia aceita a inserção de letras gregas para variáveis, que são caracteres especiais, assim, é possível inserir símbolos como ($\nabla, \div, \Omega$) entre outros, para isto, basta digitar a tecla "\" (barra invertida) precedida do nome do símbolo a ser adicionado. Ao digitar a tecla "\" e as letras iniciais do símbolo desejado é possível pressionara a tela Tab para que o Jupyter possa reconhecer de qual letra grega se trata, em seguida, quando o nome do símbolo estiver totalmente descrito, pressione Tab novamente e a tela irá imprimir o símbolo correspondente. Alguns exemplos de símbolos são descritos na Tabela 1.

Tabela 1: Exemplos de símbolos inseridos na linguagem Julia.

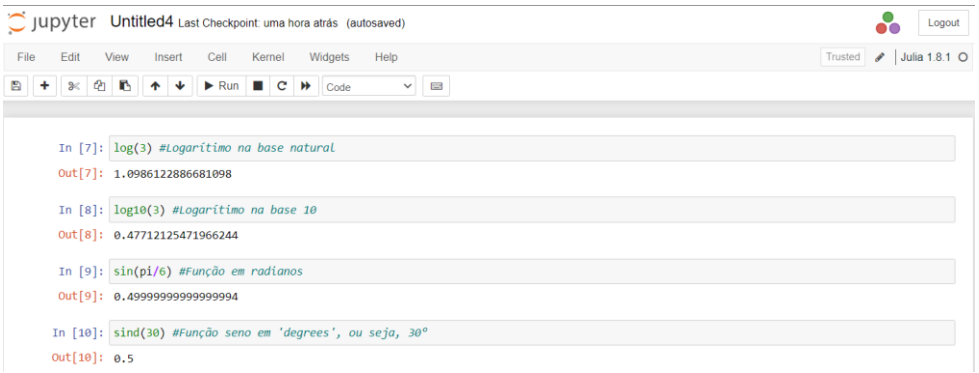
Comando	Símbolo inserido
\div	÷
\nabla	∇
\pi	π
\Omega	Ω
\sqrt	√

Figura 137: Operações matemáticas no Jupyter.



A linguagem Julia é, em sua origem, matemática, por isto apresenta diversos recursos que auxiliam na resolução de problemas através de formulações e cálculos matemáticos. Deste modo, esta linguagem Julia entende diversas operações matemáticas e também as compila automaticamente, como as funções seno, cosseno e tangente, logarítmica entre outras, como apresentado na Figura 138.

Figura 138: Formulações matemáticas no Julia.



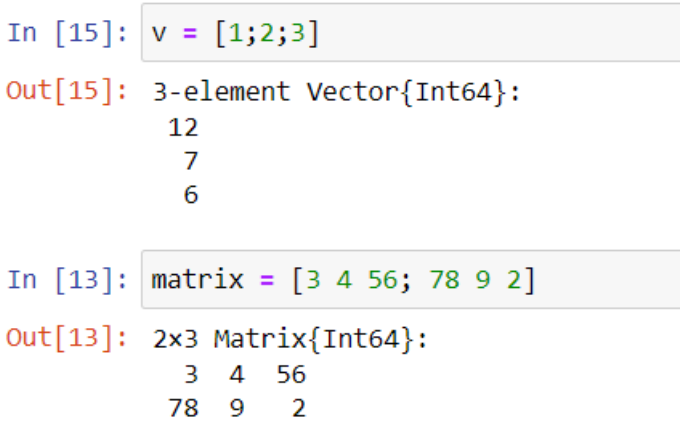
10.3 MATRIZES E VETORES

As aplicações de álgebra linear em cada linguagem são indicativos da aplicabilidade da mesma na resolução de problemas e formulações matemáticas. Linguagens como C e Fortran necessitam de pacotes adicionais externos, denominadas *librarys*, para que seja possível implementar tais funcionalidades. A linguagem Python apresenta o NumPy, tratando-se de uma biblioteca que suporta o processamento de

grandes arranjos multidimensionais, como matrizes, adicionado a um grande número de operações matemáticas para serem utilizadas sobre elas.

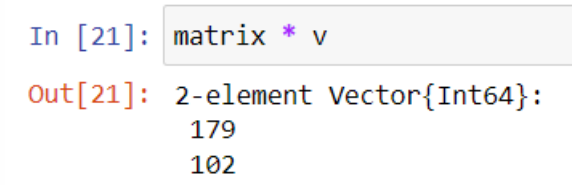
A linguagem Julia apresenta esse pacote de operações sobre geometria analítica nativamente, ou seja, é possível implementar operações lógicas deste tipo sem a necessidade de importar bibliotecas ou pacotes adicionais. É possível criar vetores e matrizes e usá-los diretamente pela linguagem Julia, como apresentado na Figura 139.

Figura 139: Criação de vetores e matrizes.



A linguagem Julia compreende formulações matemáticas utilizando tais vetores e matrizes sem necessidades de funcionalidades ou comandos especiais. Operações como a soma, subtração e multiplicação dos componentes podem ser realizadas como apresentada na Figura 140.

Figura 140: Operação de multiplicação entre um vetor e uma matriz.



Similarmente, as funções matriciais *transpose* e *adjoint* podem ser aplicadas diretamente a partir de uma matriz já criada. A função *adjoint* pode ser aplicada através de uma aspa simples (‘) ou por meio da nomenclatura da função. A Figura 141 apresenta os dois meios de aplicar a função *adjoint*.

Figura 141: Função adjoint.

```
In [22]: matrix'
Out[22]: 3×2 adjoint(::Matrix{Int64}) with eltype Int64:
           3  78
           4   9
          56   2

In [24]: adjoint(matrix)
Out[24]: 3×2 adjoint(::Matrix{Int64}) with eltype Int64:
           3  78
           4   9
          56   2
```

Funções como *transpose* e *one* (cria uma matriz preenchida com 1 em todos os espaços definidos na chamada) também são possíveis de serem utilizadas, sem que haja necessidade de adicionar pacotes externos.

10.4 FUNÇÕES EM JUPYTER/JULIA

As funções podem ser apresentadas de maneira cursiva utilizando os comandos nativos do Jupyter Notebook. Ao selecionar uma célula, os comandos de 1 a 6 alteram o tipo de apresentação do texto inserido, o que pode ser usado para apresentar funções, como mostra a Figura 142.

Figura 142: Modelo de apresentação, utilizando o comando “1” nativa do Jypiter.

```
# $$ f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} $$
```

A representação gráfica da função inserida na Figura 142 será de:

Figura 143: Representação gráfica do comando inserido na Figura 142.

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Partindo para a real finalidade das funções, há diferentes formas de descrevê-las na linguagem Julia, uma delas é puramente matemática, bastando apenas descrever uma função como **f(x),** por exemplo. Para este caso é necessário descrever os parâmetros da

função (valores) e acioná-las, atribuindo a função um valor real que será utilizado no lugar da variável descrita. A Figura 144 apresenta como pode ser formar tal processo. Nota-se que não há necessidade de especificar qual tipo de valor deve ser devolvido pela função, onde é entendido automaticamente que se trata de um valor *inteiro* ou *float*.

Figura 144: Uso de função em Julia.

```
In [11]: f(x) = sqrt(x^2+1), 2x+1
         f(4)

Out[11]: (4.123105625617661, 9)
```

Outra forma de realizar funções é explicitamente descrever uma função, através de um comando *function*, que possui parâmetros e retorna determinado valor. O caso mais clássico é da função Bhaskara (resolução de equações do segundo grau), como mostra a Figura 145.

Figura 145: Função Bharkara através de uma function.

```
In [13]: function bhaskara(a, b, c)
             Δ = b^2 - 4 * a * c
             x1 = (-b + sqrt(Δ))/2*a
             x2 = (-b - sqrt(Δ))/2*a
             return x1, x2
         end

Out[13]: bhaskara (generic function with 1 method)

In [23]: bhaskara(2.0,-10.0,4.0)

Out[23]: (18.246211251235323, 1.7537887487646788)
```

Nota-se que não foram definidos domínios para equações onde o Delta (Δ) é negativo, ou seja, no conjunto dos números imaginários. Para este caso, a função descrita apenas calcula da maneira como foi inserida. Este tipo de chamada da função é bastante útil visualmente e para organização em um código extenso, uma vez que a função pode ser chamada quantas vezes forem necessárias durante todo o código.

Uma terceira forma de se utilizar funções no Jypiter é chamando uma função anônima, denotada por $x \rightarrow (função)$, este tipo de função só pode ser utilizado se for

atribuído um nome para a mesma, deste modo, é interessante que seja utilizada dentro de outra função, gerando maior praticidade no código aplicado.

10.5 LAÇOS EM JUPYTER/JULIA

Laços são amplamente utilizados em qualquer linguagem de programação, caracterizadas por um *loop* em determinada quantidade de vezes ou infinita até que algum parâmetro ou objetivo seja alcançado. Um ótimo exemplo é pedir que uma função conte $x = x + 1$, sendo parâmetro de entrada $x = 0$, até que $x =< 0$. Como este objetivo é inalcançável, a função entrará em *loop* eterno.

Uma das maneiras de se realizar um laço em Julia é através do comando ***for*** (para em português*)* que realiza determinada função em razão de um objetivo imposto, como exemplificado na Figura 146.

Figura 146: Uso do laço for.

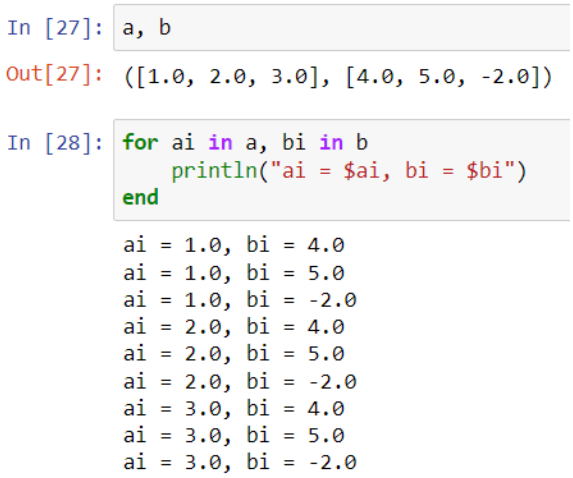
```
In [25]: for x in 1:5
             println(x)
         end

1
2
3
4
5
```

O uso do for pode ser relacionado para percorrer índices e apresentar seus valores, a exemplo, sejam definidos dois vetores a e b, onde:

$$a = [1.0, 2.0, 3.0] \; e \; b = [4.0, 5.0, -2.0]$$

pode-se usar o valor de um vetor para percorrer todos os outros valores do segundo vetor, ou seja, enquanto a = 1.0, b será igual a 4.0, 5.0 e -2.0, repetindo o laço 3 vezes nesse processo. A Figura 147 representa o modelo de construção desse tipo de laço.

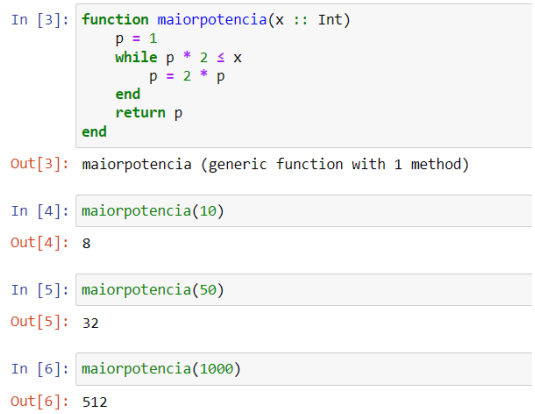
Figura 147: Laço for combinado.



Observa-se que foi utilizada também a concatenação de tipo, com o uso do **$** atribui o valor de uma variável ao texto impresso. Outro comando de laço é o **while**, que significa “enquanto”. Este comando funciona de maneira análoga ao ***for***, entretanto, sem a necessidade de índices que percorram o laço, ou seja, quando se sabe onde a função deverá encerrar é utilizado o for e quando não se sabe, utiliza-se o while.

Um ótimo exemplo de uma função com o *while* é tentar descobrir qual é a maior potência de 2 dentro de um determinado valor. Assim, caso queira saber a maior potência de 2 dentro do intervalo compreendido entre 1 e 20, o programa deverá retornar 16, uma vez que $2^4 = 16$ e posteriormente $2^5 = 32$, sendo este último valor maior que 20. A Figura 148 representa o programa mencionado como exemplo para o *while*.

Figura 148: Laço while para encontrar a maior potência de 2.



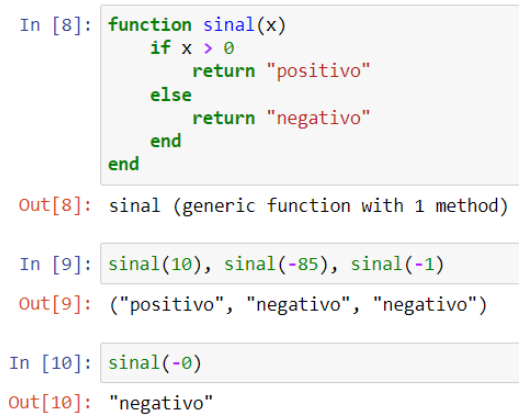
Neste caso da Fig. não está definido nenhum intervalo para números negativos, ou seja, ao inserir o comando “maiorpotencia(-10)” o programa retornará o valor de 1, uma vez que não foi estabelecido nenhuma condição para este caso.

10.6 CONDICIONAL EM JUPYTER/JULIA

Condicional em programação é uma forma de aplicar condições a um comando, que realiza diferentes ações caso a condição seja verdadeira ou falsa. Assim como em outras linguagens é utilizado os comandos ***if/elseif/else*** para designar uma condição em Julia. As condicionais auxiliam na tomada de processos durante a execução do programa e em sua fluidez a partir de um *input*.

Um exemplo da aplicação do *if* é criar uma função que retorne se o valor descrito é positivo ou negativo, como mostrado na Figura 149.

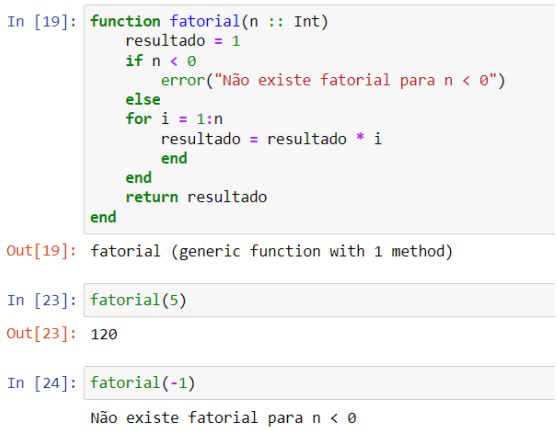
Figura 149: Condicional para descobrir o sinal de um número.



Nota-se que para o caso do número 0 não há nenhuma condicional que faça referência a esse valor ser positivo ou negativo, como para os números inteiros, seria necessária uma condição em que caso o $x = 0$ o programa não retornar nenhuma das opções positivo ou negativo.

O comando *if* é comumente utilizado para os casos específicos em funções, onde a função matematicamente não pode retornar um valor determinado, mesmo que o programa consiga chegar a um valor devido seus parâmetros de entrada. Este é o caso da função que retorna o fatorial de um número, como apresentado na Figura 150.

Figura 150: Condicional para função Fatorial.



Como não existe fatorial de um número menor ou igual a zero, é necessário explicitar uma condição para estes casos, sendo assim, o comando if é utilizado para retornar um erro sempre que a chamada da função se der com algum número negativo.

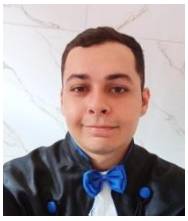
10.7 REFERÊNCIAS

[1] https://www.youtube.com/watch?v=szP9IRbwO9o

[2] https://www.youtube.com/watch?v=dPb4acFiaYs

[3] https://www.youtube.com/watch?v=Gmm5voUQaHw&t=2624s

SOBRE OS AUTORES

	Possui Graduação (2011), Mestrado (2014) e Doutorado (2018) em Engenharia Elétrica pela Universidade Federal do Maranhão (UFMA). Trabalhou de setembro de 2017 a janeiro de 2018 na Universidade de Dalhousie orientado pelo professor M. E. El-Hawary com Inteligência Computacional aplicada a Sistemas de Energia Elétrica (SEE). Atualmente é Professor Adjunto da Universidade Federal do Maranhão- Centro de Ciências de Balsas. Suas atividades incluem o desenvolvimento de metodologias e software de análise para a operação e planejamento de SEE, Controladores FACTS (*Flexible AC Transmission Systems*) e PSS (*Power System Stabilizers*), aplicações de técnicas de inteligência artificial no ajuste coordenado de controladores com vista a estabilidade angular em SEE e desenvolvimento de novas técnicas de inteligência artificial.
	Discente do Curso de Engenharia Elétrica da Universidade Federal do Maranhão- Centro de Ciências de Balsas.
	Discente do Curso Bacharelado Interdisciplinar em Ciência e Tecnologia (BICT) da Universidade Federal do Maranhão- Centro de Ciências de Balsas.
	Discente do Curso de Engenharia Civil da Universidade Federal do Maranhão- Centro de Ciências de Balsas. Atuação na área de Engenharia Civil, com ênfase em Construção Civil. Produção de pesquisas e conhecimento em áreas adversas, entre elas Programação e Análise Cromatográfica