

# Linguagem Julia para programação linear inteira-mista

Prof. Angelo Aliano Filho

UTFPR – Universidade Tecnológica Federal do Paraná

Primeiro semestre de 2021

# Sumário

- 1 Instalação e introdução
- 2 JuMP – *Julia for Mathematical Programming*
- 3 Construindo e implementando modelos
  - Variáveis
  - Expressões
  - Função objetivo
  - Restrições
  - Pós-otimização
- 4 Exemplos de implementações

# Sumário

- 1 Instalação e introdução
- 2 JuMP – *Julia for Mathematical Programming*
- 3 Construindo e implementando modelos
- 4 Exemplos de implementações

# Instalação e introdução

## Porque usarmos linguagem Julia?

- A computação científica tem tradicionalmente exigido o mais alto desempenho possível.
- A linguagem de programação Julia é uma linguagem dinâmica flexível, apropriada para computação científica e numérica, com um desempenho comparável às linguagens tradicionais.
- Julia apresenta tipagem opcional, despacho múltiplo, e bom desempenho, alcançado usando inferência de tipo e compilação just-in-time (JIT)
- Julia proporciona facilidade e expressividade para programação numérica de alto nível, da mesma forma que linguagens como R, MATLAB, e Python, mas também suporta programação geral.

# Instalação e introdução

## Porque usarmos linguagem Julia?

As justificativas mais significativas são:

- *Open source*
- Não há necessidade de vetorizar o código para o desempenho; o código desvetorizado é rápido.
- O *core* da linguagem impõe muito pouco; a base Julia e a biblioteca padrão são escritas na própria Julia, incluindo operações primitivas como a aritmética inteira.
- Chamar diretamente as funções C (não são necessários invólucros ou APIs especiais).
- Bom desempenho, aproximando-se das linguagens estaticamente compiladas como a C.

# Instalação e introdução

**Tabela:** Crescimento da linguagem Julia - algumas métricas

	Total cumulativo desde 2018	Total cumulativo desde 2019	Crescimento
Total de download	1,8 milhão	3,2 milhões	+78%
Total de pacotes	1.688	2.462	+46%
Número de novos artigos com citação	93	253	+172%
Fórum (questões)	8.620	16.363	+90%
Número de novos iniciantes	9,626	19,472	+102%
Citações do artigo: “A fresh Approach to numerical Computing (2017)”	613	1.048	+71%

Outras referências, ver em: [1, 2, 3, 4]

# Instalação e introdução

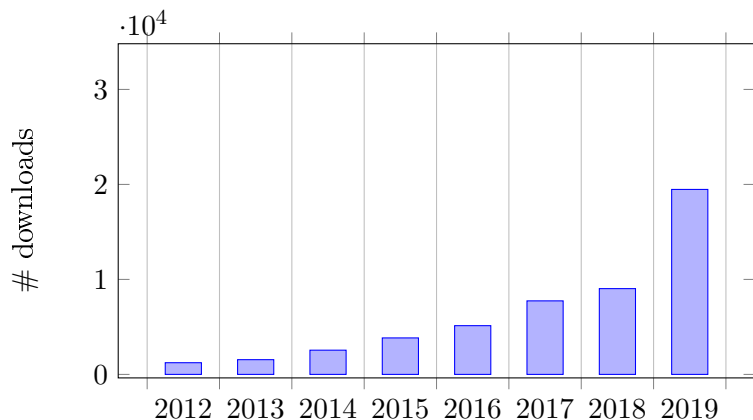


Figura: Crescimento do uso da linguagem Julia

# Instalação e introdução

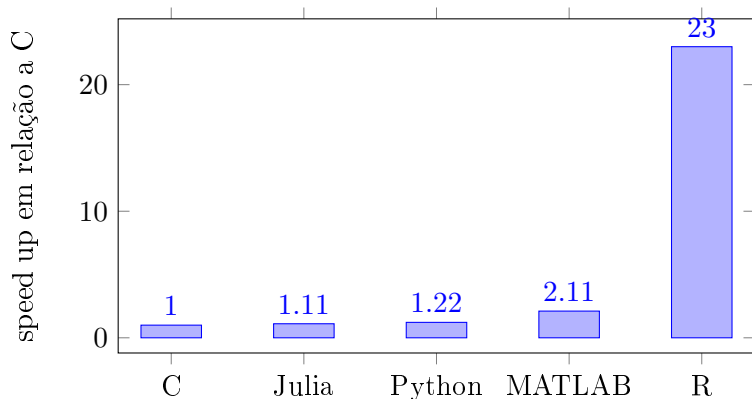


Figura: Velocidade da linguagem Julia em comparação com as demais



# Instalação e introdução

## Instalação do *Julia Professional*

**Método 1:** diretamente do site:

- Acesse <https://juliacomputing.com/products/juliapro>
- Escolher o sistema operacional. O acesso dará a versão mais atual da linguagem (v1.5.0-1).
- Fazer um cadastro com e-mail da UTFPR
- Descarregar o executável e instalar logo em seguida.

# Instalação e introdução

The screenshot shows the JuliaPro website. The header includes the Julia Computing logo and navigation links: PRODUCTS, INDUSTRIES, CASE STUDIES, EVENTS, TRAINING, MEDIA, RESOURCES, and BLOG. The main banner features the JP Julia PRO logo, the text 'Seamless Development', the version 'Version: 1.5.0-1', and links for 'All Links', 'FAQs', and 'Documentation'. A 'FREE DOWNLOAD' button is prominently displayed. Below the banner, a paragraph states that JuliaPro is free to download and is the fastest on-ramp to Julia for various professionals. Further down, there are two sections: 'Current stable release (v1.5.0-1)' and 'Release with long-term support (v1.0.5-2)'. Each section contains a table of operating systems (Windows, Mac, Linux) with download links. To the right, there are links for 'OUR OTHER ENTERPRISE PRODUCTS' and 'COMPARE FEATURES'.

Julia Computing

PRODUCTS INDUSTRIES CASE STUDIES EVENTS TRAINING MEDIA RESOURCES BLOG

JP Julia PRO

Seamless Development

Version: 1.5.0-1

All Links | FAQs | Documentation

FREE DOWNLOAD

JuliaPro is free to download and is the fastest on-ramp to Julia for individual researchers, engineers, scientists, quants, traders, economists, students and others. Julia developers can build better software quicker and easier while benefiting from Julia's unparalleled high performance.

JuliaPro is lightweight and easy to install. Use any package from 2600+ open source packages or from a curated list of 250+ JuliaPro packages. Curated packages are tested, documented and supported by Julia Computing. See below for details on curated packages.

Current stable release (v1.5.0-1)

Operating system	v1.5.0-1*	Quick-start Guide
Windows		
Mac**		
Linux		
Linux (SPG)		

Release with long-term support (v1.0.5-2)

Operating system	v1.0.5-2*	Quick-start Guide
Windows		
Mac**		
Linux		
Linux (SPG)		

OUR OTHER ENTERPRISE PRODUCTS

Julia SURE<sup>™</sup>

Julia TEAM<sup>™</sup>

Julia RUN<sup>™</sup>

COMPARE FEATURES

Figura: Página de *download* do JuliaPro

# Instalação e introdução



Figura: Cadastro no site oficial do JuliaPro

# Instalação e introdução

## Instalação do *Julia Professional*

**Método 2:** acesso a versões mais antigas e instalação em dois passos:

- Acessar <https://julialang.org/downloads/oldreleases/>
- Escolher a versão e baixar o executável.
- Em minha página pessoal há também (versão 1.5.0): acesse <http://paginapessoal.utfpr.edu.br/angeloaliano> em seguida na pasta “Linguagem de programação Julia”.
- Deveremos em seguida fazer o download do editor ATOM.

# Instalação e introdução

**Older Unmaintained Releases**

Old releases are available should you need to use them to run Julia code written for those releases. Note that these are not actively developed nor maintained anymore.

**v1.4.2 (May 23, 2020)**

<a href="#">Windows [help]</a>	<a href="#">64-bit</a>	<a href="#">32-bit</a>
<a href="#">macOS [help]</a>	<a href="#">64-bit</a>	
<a href="#">Generic Linux on x86 [help]</a>	<a href="#">64-bit (GPG)</a>	<a href="#">32-bit (GPG)</a>
<a href="#">Generic Linux on ARM [help]</a>	<a href="#">64-bit (AArch64) (GPG)</a>	
<a href="#">Generic FreeBSD on x86 [help]</a>	<a href="#">64-bit (GPG)</a>	
<a href="#">Source</a>	<a href="#">Tarball (GPG)</a>	<a href="#">Tarball with dependencies (GPG)</a>
		<a href="#">GitHub</a>

**v1.3.1 (Dec 30, 2019)**

<a href="#">Windows (.exe) [help]</a>	<a href="#">32-bit</a>	<a href="#">64-bit</a>
<a href="#">macOS (.dmg) [help]</a>		<a href="#">64-bit</a>
<a href="#">Generic Linux Binaries for x86 [help]</a>	<a href="#">32-bit (GPG)</a>	<a href="#">64-bit (GPG)</a>
<a href="#">Generic Linux Binaries for ARM [help]</a>	<a href="#">32-bit (AArch64 hard float) (GPG)</a>	<a href="#">64-bit (AArch64) (GPG)</a>
<a href="#">Generic FreeBSD Binaries for x86 [help]</a>		<a href="#">64-bit (GPG)</a>
<a href="#">Source</a>	<a href="#">Tarball (GPG)</a>	<a href="#">Tarball with dependencies (GPG)</a>
		<a href="#">GitHub</a>

Figura: Página do software Julia em versões antigas

# Instalação e introdução

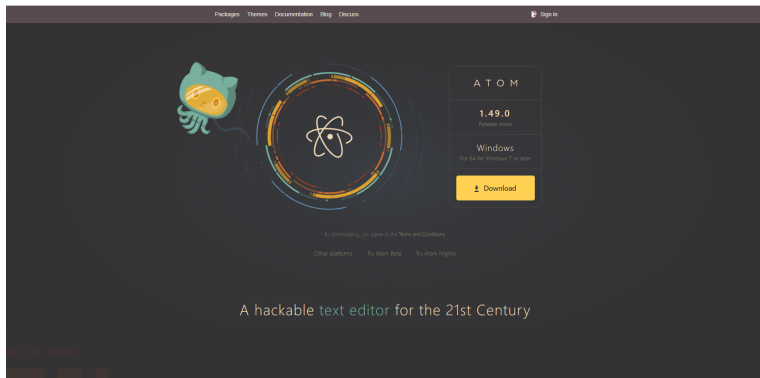


Figura: Página de download do ATOM

# Instalação e introdução



Figura: Arquivos para instalação da linguagem Julia

# Instalação e introdução

## Instalação do *Julia Professional*

**Método 2:** acesso a versões mais antigas e instalação em dois passos:

- Acessar <https://atom.io/>
- Faça o download do editor ATOM
- Em seguida, rode o executável baixado
- Precisamos, agora, configurar o ATOM para a linguagem Julia e escrever os códigos para compilação



# Instalação e introdução

## Instalação do *Julia Professional*

Configuração do ambiente ATOM:

- Abra o ATOM.
- Vá em *settings* –» *install* para instalar os pacotes para o ATOM
- Na caixinha “install Packages” digite **uber-juno** e dê um enter para instalar o pacote “uber-juno” clicando em *install*. Este pacote serve para configurar o ATOM para a linguagem Julia

# Instalação e introdução

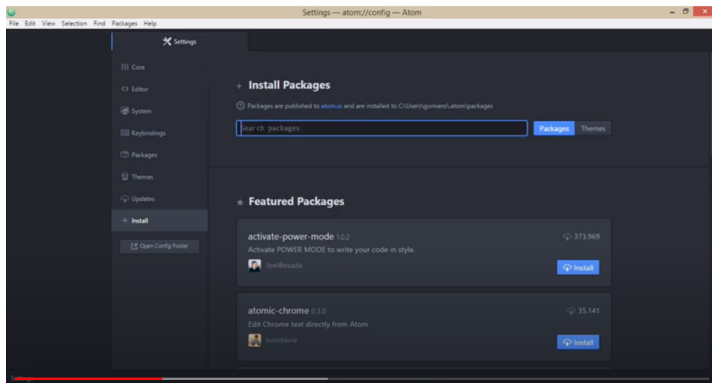


Figura: Configuração ATOM - passo 1

# Instalação e introdução

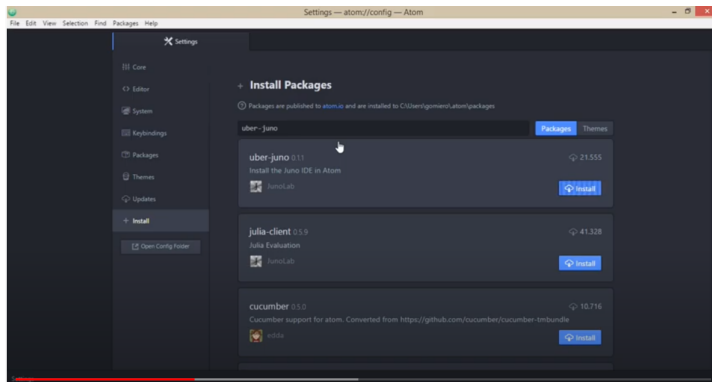


Figura: Configuração ATOM - passo 2

# Instalação e introdução

## Instalação do *Julia Professional*

Configuração do ambiente ATOM:

- Depois disso, vá em *settings* → *packages* → *julia-client*
- Clique em *settings*
- Insira o caminho de onde está instalado o executável  
“...\\Julia-1.5.0\\bin\\julia.exe”

# Instalação e introdução

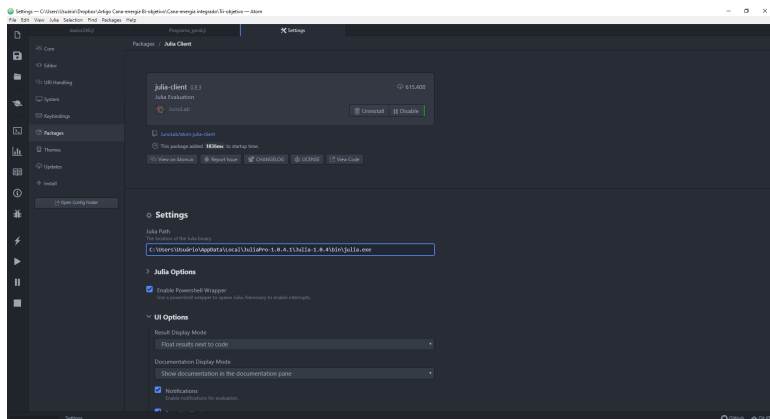


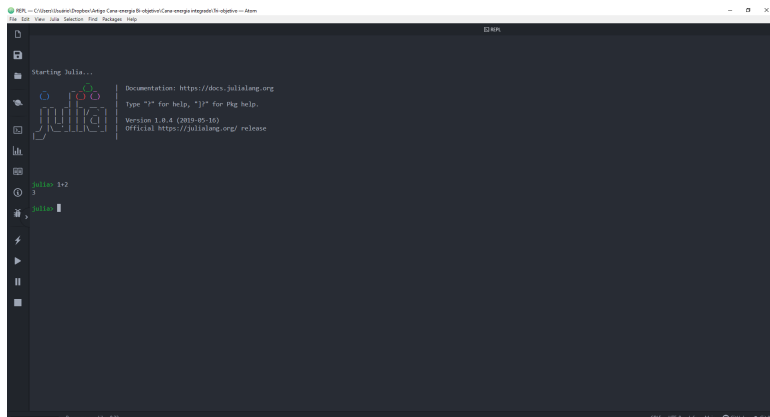
Figura: Configuração ATOM - passo 3

# Instalação e introdução

## Primeiros passos

- A forma mais fácil de aprender e experimentar com Julia é iniciar uma sessão interativa “REPL” na linha de comando.
- Alternativamente, você pode criar um arquivo e salvá-lo no formato “.jl” e compilar cada linha selecionando-a e dando “ctrl+enter” ou pressionar o pequeno triângulo.
- É neste arquivo .jl é que podemos criar nossos primeiros programas.

# Instalação e introdução



The screenshot shows a terminal window titled "REPL" with a menu bar (File, Edit, View, Julia, Selection, Find, Packages, Help). The terminal content is as follows:


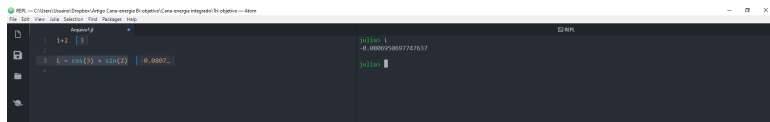
```
Starting Julia...  
 Documentation: https://docs.julialang.org  
Type "?" for help, "}" for Pkg help.  
Version 1.0.4 (2019-05-16)  
Official https://julialang.org/ release  
  
julia> 1+2  
3  
julia> 
```

Figura: Primeiros passos na linguagem Julia - linha de comando

# Instalação e introdução



The screenshot shows a Julia REPL window with the following content:

```
1 1+2 3
2
3 L = cos(3) + sin(2) -0.9997...
4
```

On the right side of the window, the output is displayed:

```
julia> L
-0.0006558697747637
julia>
```

Figura: Primeiros passos na linguagem Julia - via arquivo .jl



# Operações matemáticas

## Primeiros passos

- Para sair da sessão interactiva, digite `ctrl+d` ou digite `exit()`
- Se uma expressão é introduzida numa sessão interativa com um ponto-e-vírgula, o seu valor não é mostrado. Caso contrário é.
- A variável `ans` está ligada ao valor da última expressão avaliada, quer seja mostrada ou não
- Para avaliar expressões escritas num arquivo `fonte.jl`, escreva `include("file.jl")`.
- Para limpar o prompt de comando, só digitar `ctrl+l`
- Todo comentário é inserido ao colocarmos `#`

# Sumário

- 1 Instalação e introdução
- 2 JuMP – *Julia for Mathematical Programming*
- 3 Construindo e implementando modelos
- 4 Exemplos de implementações

# JuMP – *Julia for Mathematical Programming*

- JuMP é uma linguagem de modelagem para otimização matemática incorporada em Júlia.
- Suporta uma série de solvers de código aberto e comerciais para uma variedade de classes de problemas, incluindo programação linear inteira-mista, programação cônica de segunda ordem, programação semidefinida e programação não linear.

# JuMP – *Julia for Mathematical Programming*

Vantagens do uso do pacote JuMP ([5]):

- Facilidade de utilização
  - ▶ Sintaxe que imita as expressões matemáticas naturais.
  - ▶ Documentação completa em <https://jump.dev/JuMP.jl/v0.21.5/>
- Velocidade
  - ▶ JuMP pode criar problemas a velocidades semelhantes às das linguagens tais como AMPL.
  - ▶ JuMP comunica com a maioria dos solvers, evitando a necessidade de escrever arquivos intermediários.
- Independência do solver
  - ▶ JuMP utiliza uma interface genérica independente do solver facilitando a mudança entre vários pacotes de software de código aberto e de otimização comercial
  - ▶ Os solvers atualmente apoiados incluem: Artelys Knitro, Bonmin, Cbc, Clp, Couenne, CPLEX, ECOS, FICO Xpress, GLPK, Gurobi, Ipopt, MOSEK, NLOpt, e SCS.

## Instalação

- Primeiro, precisamos instalar o pacote JuMP rodando:

```
import Pkg  
Pkg.add("JuMP")  
e depois using JuMP
```

# JuMP – Julia for Mathematical Programming

**Tabela:** Lista de solvers suportados e tipos de problemas resolvidos com JuMP

Solver	Pacote Julia	Licença	Suporte
Artelys Knitro	KNITRO.jl	Comm.	LP, MILP, SOCP, MISOCP, NLP, MINLP
Cbc	Cbc.jl	EPL	MILP
CDCS	CDCS.jl	GPL	LP, SOCP, SDP
CDD	CDDLib.jl	GPL	LP
Clp	Clp.jl	EPL	LP
COSMO	COSMO.jl	Apache	LP, QP, SOCP, SDP
CPLEX	CPLEX.jl	Comm.	LP, MILP, SOCP, MISOCP
CSDP	CSDP.jl	EPL	LP, SDP
ECOS	ECOS.jl	GPL	LP, SOCP
FICO Xpress	Xpress.jl	Comm.	LP, MILP, SOCP, MISOCP
GLPK	GLPK.jl	GPL	LP, MILP
Gurobi	Gurobi.jl	Comm.	LP, MILP, SOCP, MISOCP
Ipopt	Ipopt.jl	EPL	LP, QP, NLP
MOSEK	MosekTools.jl	Comm.	LP, MILP, SOCP, MISOCP, SDP
OSQP	OSQP.jl	Apache	LP, QP
ProxSDP	ProxSDP.jl	MIT	LP, SOCP, SDP
SCIP	SCIP.jl	ZIB	MILP, MINLP
SCS	SCS.jl	MIT	LP, SOCP, SDP
SDPA	SDPA.jl, SDPAFamily.jl	GPL	LP, SDP
SDPNAL	SDPNAL.jl	CC BY-SA	LP, SDP
SDPT3	SDPT3.jl	GPL	LP, SOCP, SDP
SeDuMi	SeDuMi.jl	GPL	LP, SOCP, SDP
Tulip	Tulip.jl	MPL-2	LP

# JuMP – *Julia for Mathematical Programming*

Legenda:

- LP = Linear programming
- QP = Quadratic programming
- SOCP = Second-order conic programming (including problems with convex quadratic constraints and/or objective)
- MILP = Mixed-integer linear programming
- NLP = Nonlinear programming
- MINLP = Mixed-integer nonlinear programming
- SDP = Semidefinite programming
- MISDP = Mixed-integer semidefinite programming

## Instalação do CPLEX

- Requer uma instalação funcional do CPLEX com uma licença (gratuita para docentes e assistentes de ensino graduados).
- Acessar: <https://www.ibm.com/support/pages/downloading-ibm-ilog-cplex-optimization-studio-v1290>
- Feito o download e a instalação, rodar:

```
import Pkg
Pkg.add("CPLEX")
using JuMP
using CPLEX
modelo = Model(CPLEX.Optimizer)
para a resolução de MILP com o CPLEX
```



## Definição de parâmetros

- Usamos a macro `set_optimizer_attributes` para definir opções específicas do solver.
- Um exemplo seria:

```
set_optimizer_attribute(modelo,"CPX_PARAM_TILIM",60)  
set_optimizer_attribute("CPX_PARAM_EPGAP",0.01)
```

define o CPLEX para rodar 60 segundos ou até que o Gap seja de 1%.

Outras opções de parâmetros para o CPLEX, ver em

[https://www.ibm.com/support/knowledgecenter/SSSA5P\\_12.8.0/ilog.odms.cplex.help/CPLEX/Parameters](https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/CPLEX/Parameters)

# Sumário

- 1 Instalação e introdução
- 2 JuMP – *Julia for Mathematical Programming*
- 3 Construindo e implementando modelos
  - Variáveis
  - Expressões
  - Função objetivo
  - Restrições
  - Pós-otimização
- 4 Exemplos de implementações

# Sumário

- 1 Instalação e introdução
- 2 JuMP – *Julia for Mathematical Programming*
- 3 Construindo e implementando modelos
  - Variáveis
  - Expressões
  - Função objetivo
  - Restrições
  - Pós-otimização
- 4 Exemplos de implementações

# Construindo e implementando modelos

Variáveis em JuMP são instâncias da estrutura da macro `@variable()` que contém uma referência a uma variável de um modelo de otimização.

- Para adicionar limitantes para uma variável  $x$  em um modelo, pode ser feito da seguinte maneira:

- ▶ Livre:

```
julia> @variable(modelo, x_livre)  
x_livre
```

- ▶ Limitada inferiormente:

```
julia> @variable(modelo, x_inf >= 0)  
x_inf
```

# Construindo e implementando modelos

Variáveis em JuMP são instâncias da estrutura da macro `@variable()` que contém uma referência a uma variável de um modelo de otimização.

- Para adicionar limitantes para uma variável  $x$  em um modelo, pode ser feito da seguinte maneira:

- ▶ Limitada superiormente:

```
julia> @variable(modelo, x_sup <= 1)
x_sup
```

- ▶ Intervalar:

```
julia> @variable(modelo, 2 <= x_intervalo <= 3)
x_intervalo
```

- ▶ Fixa:

```
julia> @variable(modelo, x_fixa == 4)
x_fixa
```

# Construindo e implementando modelos

## Observação

Ao criar uma variável com apenas um limite inferior ou um limite superior, e o valor do limite não é literal, o nome da variável deve aparecer no lado esquerdo. Colocar o nome no lado direito resultará num erro. Por exemplo:

```
@variable(modelo, 1 <= x) # funciona
a = 1
@variable(modelo, a <= x) # não funciona
@variable(modelo, x >= a) # funciona
```

# Construindo e implementando modelos

- Blocos de múltiplas variáveis podem ser criados com o ambiente `begin...end`. Veja o exemplo:

```
@variables(modelo,  
begin  
    x >= 0  
    y <= 2  
end)
```

ao invés de declarar cada restrição ou bloco separadamente.

## Construindo e implementando modelos

Ao invés de usar `<=` ou `>=` para definir os bounds, podemos usar as palavras-chave `lower_bound` e `upper_bound`. Por exemplo:

```
julia> @variable(modelo, x, lower_bound=1, upper_bound=2)
x
julia> lower_bound(x)
1.0
```

Outra opção seria usar `set_lower_bound` e `set_upper_bound`, modificando variáveis já existentes. Exemplo:

```
julia> @variable(modelo, x >= 1)
x
julia> lower_bound(x)
1.0
julia> set_lower_bound(x, 2)
julia> lower_bound(x)
2.0
```



# Construindo e implementando modelos

Para criar coleções de variáveis, como por exemplo,  $x_{ij}$  usamos *contadores*.

- Usando arrays para criar variáveis multidimensionais:

```
julia> @variable(modelo, x[1:2, 1:2])  
2x2 Array{VariableRef,2}:  
 x[1,1]  x[1,2]  
 x[2,1]  x[2,2]
```

- As variáveis poderão depender dos índices. Veja o exemplo:

```
julia> @variable(modelo, x[i=1:2, j=1:2] >= 2i + j)  
2x2 Array{VariableRef,2}:  
 x[1,1]  x[1,2]  
 x[2,1]  x[2,2]
```

```
julia> lower_bound.(x)  
2x2 Array{Float64,2}:  
 3.0  4.0  
 5.0  6.0
```

# Construindo e implementando modelos

- Exemplo: queremos criar  $0 \leq x_{ijk}$  com  $i \in \{2, 3, \dots, 10\}$ ,  $j \in \{10, \dots, 20, 100\}$  e  $k \in \{5, 6\}$ , fazemos:  

```
julia> @variable(modelo, x[i=2:10, j=10:10:100, k=5:6] >= 0)
```
- Exemplo: queremos criar  $1 \leq x_{ij} \leq 2$  com  $i \in I$  e  $j \in J_i$  onde  $I = \{1, \dots, 5\}$  e  $J_i = \{J | 1 \leq i + j \leq 10\}$ .  

```
julia> @variable(modelo, 1<= x[i=1:5, j=1:10-i] <=2)
```
- Exemplo: queremos criar  $x_{ijk} \leq 3$  com  $i \in I = \{1, \dots, 3\}$ ,  $J \in J = \{1, \dots, 4\}$  e  $k \in \{a_i, \dots, b_i\}$  onde  $a = [4, 5, 6]$  e  $b = [10, 20, 30]$ :  

```
a = [4,5,6]; b=[10,20,30]  
julia> @variable(modelo, x[i=1:3, j=1:4, k=a[i]:b[i]] <=3)
```

# Construindo e implementando modelos

- Exemplo: queremos criar  $0 \leq x_{ijk}$  com  $i \in \{2, 3, \dots, 10\}$ ,  $j \in \{10, \dots, 20, 100\}$  e  $k \in \{5, 6\}$ , fazemos:  

```
julia> @variable(modelo, x[i=2:10, j=10:10:100, k=5:6] >= 0)
```
- Exemplo: queremos criar  $1 \leq x_{ij} \leq 2$  com  $i \in I$  e  $j \in J_i$  onde  $I = \{1, \dots, 5\}$  e  $J_i = \{J | 1 \leq i + j \leq 10\}$ .  

```
julia> @variable(modelo, 1<= x[i=1:5, j=1:10-i] <=2)
```
- Exemplo: queremos criar  $x_{ijk} \leq 3$  com  $i \in I = \{1, \dots, 3\}$ ,  $J \in J = \{1, \dots, 4\}$  e  $k \in \{a_i, \dots, b_i\}$  onde  $a = [4, 5, 6]$  e  $b = [10, 20, 30]$ :  

```
a = [4,5,6]; b=[10,20,30]  
julia> @variable(modelo, x[i=1:3, j=1:4, k=a[i]:b[i]] <=3)
```

# Construindo e implementando modelos

- Exemplo: queremos criar  $0 \leq x_{ijk}$  com  $i \in \{2, 3, \dots, 10\}$ ,  $j \in \{10, \dots, 20, 100\}$  e  $k \in \{5, 6\}$ , fazemos:  

```
julia> @variable(modelo, x[i=2:10, j=10:10:100, k=5:6] >= 0)
```
- Exemplo: queremos criar  $1 \leq x_{ij} \leq 2$  com  $i \in I$  e  $j \in J_i$  onde  $I = \{1, \dots, 5\}$  e  $J_i = \{J | 1 \leq i + j \leq 10\}$ .  

```
julia> @variable(modelo, 1<= x[i=1:5, j=1:10-i] <=2)
```
- Exemplo: queremos criar  $x_{ijk} \leq 3$  com  $i \in I = \{1, \dots, 3\}$ ,  $J \in J = \{1, \dots, 4\}$  e  $k \in \{a_i, \dots, b_i\}$  onde  $a = [4, 5, 6]$  e  $b = [10, 20, 30]$ :  

```
a = [4,5,6]; b=[10,20,30]  
julia> @variable(modelo, x[i=1:3, j=1:4, k=a[i]:b[i]] <=3)
```

# Construindo e implementando modelos

- Variáveis binárias: basta usar o argumento adicional Bin

```
julia> @variable(modelo, x, Bin)
```

ou com

```
julia> @variable(modelo, x, binary=true)
```

- Variáveis binárias: basta usar o argumento adicional Int

```
@variable(modelo, x, Int)
```

ou com

```
julia> @variable(modelo, x, integer=true)
```

# Construindo e implementando modelos

- É possível *listar todas as variáveis* de um modelo. Isto pode ser útil para:
  - ▶ Relaxar todas as variáveis no modelo
  - ▶ Atribuir valores iniciais para as variáveis

Isso pode ser feito com o comando `JuMP.all_variables(modelo)`.

- Também é possível contar todas as variáveis do modelo com `num_variables(modelo)`

# Construindo e implementando modelos

- A integralidade das variáveis é relaxada com `relax_integrality(modelo)`. Veja o exemplo:

```
julia> modelo = Model(CPLEX.Optimizer);  
julia> @variable(modelo, x, Bin);  
julia> @variable(modelo, 1 <= y <= 10, Int);  
julia> @objective(modelo, Min, x + y);  
julia> undo_relax = relax_integrality(modelo);  
julia> print(model)  
Min x + y  
Subject to  
  x >= 0.0  
  y >= 1.0  
  x <= 1.0  
  y <= 10.0
```

# Construindo e implementando modelos

- É possível de fornecer uma solução primal para um modelo, usando o comando `set_start_value`
- Assim, `@variable(model, y, start = 1)` inicia a variável  $y$  em 1.
- Fornecendo o comando `start_value(y)` temos 1.0
- Outra maneira seria `set_start_value(y, 2)` donde segue que `start_value(y)` resulta em 2

Isto pode ser muito útil quando temos uma solução factível para um determinado modelo.



# Sumário

- 1 Instalação e introdução
- 2 JuMP – *Julia for Mathematical Programming*
- 3 Construindo e implementando modelos
  - Variáveis
  - Expressões
  - Função objetivo
  - Restrições
  - Pós-otimização
- 4 Exemplos de implementações

# Construindo e implementando modelos

- Há três tipos de expressões: afins, quadrática e não-linear. Podem ser inseridas nas restrições ou na função-objetivo
- Expressão afim: pode ser construída com a macro `@expression`. Veja o exemplo:

```
modelo = Model()
@variable(modelo, x)
@variable(modelo, y)
ex = @expression(modelo, 2x + y - 1)
@objective(model, Min, 2 * ex - 1)
objective_function(model)
#output
4 x + 2 y - 3
```

# Construindo e implementando modelos

- Outro exemplo: múltiplas expressões criadas como a seguir.

```
modelo = Model()
@variable(modelo, x[i = 1:3])
@expression(modelo, expr[i = 1:3], i*sum(x[j] for j in i:3))
expr
```

```
3-element Array{GenericAffExpr{Float64,VariableRef},1}:
 x[1] + x[2] + x[3]
 2 x[2] + 2 x[3]
 3 x[3]
```

- 
- 
-

# Construindo e implementando modelos

- Expressões quadráticas podem ser criadas através de `@expression`, como a seguir:

```
modelo = Model()
@variable(modelo, x)
@variable(modelo, y)
ex = @expression(modelo, x^2 + 2 * x * y + y^2 + x + y - 1)
```

```
# output
```

```
x^2 + 2 y*x + y^2 + x + y - 1
```

# Construindo e implementando modelos

- Expressões não-lineares são criadas com as macros `@NLexpression`, `@NLobjective`, `@NLconstraint`.
- Demais detalhes, ver em <https://jump.dev/JuMP.jl/v0.21.5/nlp/#Nonlinear-Modeling-1> para obtenção de modelos não-lineares.

# Sumário

- 1 Instalação e introdução
- 2 JuMP – *Julia for Mathematical Programming*
- 3 Construindo e implementando modelos
  - Variáveis
  - Expressões
  - Função objetivo
  - Restrições
  - Pós-otimização
- 4 Exemplos de implementações

# Construindo e implementando modelos

- As funções-objetivo lineares e quadráticas são escrita através da macro `@objective`
- Após a otimização, para verificar o valor da função-objetivo, use `objective_value` e `objective_bound`
- Para o valor ótimo do objetivo dual, use `dual_objective_value`

# Construindo e implementando modelos

- A macro `@objective` possui a seguinte estrutura:

```
@objective(nome_do_modelo, Min/Max, funcao)
```

- Exemplo: se queremos maximizar  $2x - 1$  usamos:

```
@objective(modelo, Max, 2x - 1)
```

- Exemplo: se queremos minimizar  $\sum_{i=1}^5 \sum_{j=i}^{10} c_{ij}x_{ij}$  fazemos:

```
c = rand(5,10)
```

```
@variable(modelo, 1<= x[i=1:5, j=i:10] <=2)
```

```
@objective(modelo, Min, sum(c[i,j]*x[i,j] for i in 1:5  
                             for j in i:10) )
```



# Sumário

- 1 Instalação e introdução
- 2 JuMP – *Julia for Mathematical Programming*
- 3 Construindo e implementando modelos
  - Variáveis
  - Expressões
  - Função objetivo
  - Restrições
  - Pós-otimização
- 4 Exemplos de implementações

# Construindo e implementando modelos

- São escritas usando a macro `@constraint`. Veja o exemplo:

```
julia> @constraint(modelo, res, 2x <= 1)
con : 2 x <= 1.0
```

insere a restrição **res**  $2x \leq 1$  no **modelo**. Esta restrição pode ser acessada no seu modelo dando o comando `modelo[:res]`

- Pode ser possível criar restrições do tipo  $\geq$ ,  $=$  ou canalizadas:

```
julia> @constraint(modelo, 2x >= 1)
2 x >= 1.0
```

```
julia> @constraint(modelo, 2x == 1)
2 x = 1.0
```

```
julia> @constraint(modelo, 1 <= 2x <= 3)
2 x in [1.0, 3.0]
```

```
julia> @constraint(modelo, 2x + 1 <= 4x + 4)
```

# Construindo e implementando modelos

- São escritas usando a macro `@constraint`. Veja o exemplo:

```
julia> @constraint(modelo, res, 2x <= 1)
con : 2 x <= 1.0
```

insere a restrição  $\text{res } 2x \leq 1$  no `modelo`. Esta restrição pode ser acessada no seu modelo dando o comando `modelo[:res]`

- Pode ser possível criar restrições do tipo  $\geq$ ,  $=$  ou canalizadas:

```
julia> @constraint(modelo, 2x >= 1)
2 x >= 1.0
```

```
julia> @constraint(modelo, 2x == 1)
2 x = 1.0
```

```
julia> @constraint(modelo, 1 <= 2x <= 3)
2 x in [1.0, 3.0]
```

```
julia> @constraint(modelo, 2x + 1 <= 4x + 4)
```

# Construindo e implementando modelos

- Há uma maneira de definir múltiplas restrições usando a macro `@constraint`. Veja o exemplo:

```
@constraints(modelo,  
begin  
    2x <= 1  
    x >= -1  
end)
```

```
julia> print(modelo)  
Feasibility  
Subject to  
  x >= -1.0  
  2 x <= 1.0
```

## Construindo e implementando modelos

- Há porém uma maneira mais eficiente ao usarmos contadores, permitindo construir grupos de restrições eficientemente. Veja o exemplo:

```
julia> @constraint(modelo, res[i = 1:3], i * x <= i + 1)
```

```
con[1] : x <= 2.0
```

```
con[2] : 2 x <= 3.0
```

```
con[3] : 3 x <= 4.0
```

- Se quisermos ver a primeira restrição, usamos

```
julia> res[1]
```

```
res[1] : x <= 2.0
```

- Todavia, restrições anônimas podem ser criadas também:

```
@constraint(model, [i = 1:2], i * x <= i + 1)
```

```
x <= 2.0
```

```
2 x <= 3.0
```

# Construindo e implementando modelos

- Conjunto de restrições multi-dimensionais também podem ser eficientemente criados. Veja o exemplo:

```
julia> @constraint(modelo, res[i = 1:2, j = 2:3], i*x <= j+1)
```

```
res[1,2] : x <= 3.0    res[1,3] : x <= 4.0  
res[2,2] : 2 x <= 3.0  res[2,3] : 2 x <= 4.0
```

- Condições podem ser adicionadas, como por exemplo,  $i \neq j$ . Veja:

```
julia> @constraint(model, con[i=1:2, j=1:2; i!=j], i*x <= j+1)
```

```
res[1,2] : x <= 3.0  
res[2,1] : 2 x <= 2.0
```

# Construindo e implementando modelos

- Conjunto de restrições multi-dimensionais também podem ser eficientemente criados. Veja o exemplo:

```
julia> @constraint(modelo, res[i = 1:2, j = 2:3], i*x <= j+1)
```

```
res[1,2] : x <= 3.0    res[1,3] : x <= 4.0
```

```
res[2,2] : 2 x <= 3.0  res[2,3] : 2 x <= 4.0
```

- Condições podem ser adicionadas, como por exemplo,  $i \neq j$ . Veja:

```
julia> @constraint(model, con[i=1:2, j=1:2; i!=j], i*x <= j+1)
```

```
res[1,2] : x <= 3.0
```

```
res[2,1] : 2 x <= 2.0
```

# Construindo e implementando modelos

Restrições na forma vetorizadas podem também ser escritas. Veja o exemplo passo-a-passo:

- Definimos a variável  $x$ : `@variable(model, x[i=1:2])`
- Definimos a matriz  $A$ : `A = [1 2; 3 4]`
- Definimos o vetor do lado direito  $b$ : `b = [5, 6]`
- Escrevendo a restrição  $Ax = b$  temos:  
`@constraint(model, con, A * x .== b)` fornece como resultado:  
$$\begin{aligned} x[1] + 2 x[2] &== 5.0 \\ 3 x[1] + 4 x[2] &== 6.0 \end{aligned}$$

**Observação:** os operadores de comparação `.==`, `.>=` e `.<=` devem vir acompanhados de um ponto.



# Sumário

- 1 Instalação e introdução
- 2 JuMP – *Julia for Mathematical Programming*
- 3 Construindo e implementando modelos
  - Variáveis
  - Expressões
  - Função objetivo
  - Restrições
  - Pós-otimização
- 4 Exemplos de implementações

# Construindo e implementando modelos

Assim que o modelo for construído, ele pode ser resolvido dando o comando `optimize(modelo)!`. Algumas questões queremos saber ao final da otimização, como:

- Porque a otimização parou?
- Temos uma solução para nosso problema?
- A solução é ótima?
- Temos uma solução dual?

# Construindo e implementando modelos

- O comando `termination_status(modelo)` diz porque a otimização parou por exemplo:
  - ▶ solução ótima global encontrada (`OPTIMAL`)
  - ▶ problema infactível ou ilimitado (`INFEASIBLE_OR_UNBOUNDED`)
  - ▶ determinação de ótimo local (`LOCALLY_SOLVED`)
  - ▶ tempo máximo de CPU atingido, máximo número de nós, etc (`TIME_LIMIT`, `NODE_LIMIT`)
- Adicionalmente, podemos receber informações adicionais sobre o porquê do solver parar com o comando `raw_status`
- Também é possível saber se, após a otimização, alguma solução factível foi determinada com o comando `has_values(modelo)`. Caso o modelo não tenha uma solução primal este comando retorna `false`

# Construindo e implementando modelos

- O comando `termination_status(modelo)` diz porque a otimização parou por exemplo:
  - ▶ solução ótima global encontrada (`OPTIMAL`)
  - ▶ problema infactível ou ilimitado (`INFEASIBLE_OR_UNBOUNDED`)
  - ▶ determinação de ótimo local (`LOCALLY_SOLVED`)
  - ▶ tempo máximo de CPU atingido, máximo número de nós, etc (`TIME_LIMIT`, `NODE_LIMIT`)
- Adicionalmente, podemos receber informações adicionais sobre o porquê do solver parar com o comando `raw_status`
- Também é possível saber se, após a otimização, alguma solução factível foi determinada com o comando `has_values(modelo)`. Caso o modelo não tenha uma solução primal este comando retorna `false`

# Construindo e implementando modelos

- Estes status indicam o tipo de resultado disponível pós-otimização. Isto pode ser solicitado com `primal_status` (primal) ou com `dual_status` (dual). As opções são:
  - ▶ `NO_SOLUTION`: o resultado é um vetor vazio
  - ▶ `FEASIBLE_POINT`: o resultado é um vetor factível
  - ▶ `INFEASIBLE_POINT`: o resultado é um vetor infactível
  - ▶ `NEARLY_FEASIBLE_POINT`: o resultado é um vetor factível se algumas restrições forem relaxadas

# Construindo e implementando modelos

- A solução primal pode ser obtida com `value(x)`
- Para a solução dual, basta usar `dual(x)`
- O valor objetivo pode ser obtido com `objective_value(modelo)`
- O melhor limitante para a função objetivo é determinado com `objective_bound`

# Construindo e implementando modelos

Recomendamos o seguinte código quando após resolver um problema de otimização linear inteiro misto:

```
using JuMP
modelo = Model()
@variable(model, x[1:10] >= 0)
# ... other constraints ...
optimize!(modelo)

if termination_status(model) == OPTIMAL
    sol_otima = value.(x)
    obj_otimo = objective_value(modelo)
elseif termination_status(model) == TIME_LIMIT && has_values(modelo)
    sol_sub_otima = value.(x)
    obj_sub_otimo = objective_value(modelo)
else
    error("O modelo não foi resolvido corretamente")
end
```

# Construindo e implementando modelos

Podemos acessar outros atributos como:

- `relative_gap(modelo)`: mostra o gap de integralidade
- `solve_time(modelo)`: tempo de CPU utilizado
- `simplex_iterations`: número de iterações do simplex
- `node_count`: conta o número de nós da árvore branch-and-bound explorados
- `result_count`: verifica quantas soluções alternativas o modelo possui



# Sumário

- 1 Instalação e introdução
- 2 JuMP – *Julia for Mathematical Programming*
- 3 Construindo e implementando modelos
- 4 Exemplos de implementações

# Resolvendo alguns modelos

**Problema 1:** problema linear simples

$$\begin{array}{ll}\text{minimize} & 3x_1 - 11x_2 + 5x_3 + x_4 \\ \text{sujeito a} & x_1 + 5x_2 - 3x_3 + 6x_4 \leq 7 \\ & -x_1 + x_2 + x_3 - 2x_4 \geq 3 \\ & x_1, x_2, x_3, x_4 \geq 0\end{array}$$

# Resolvendo alguns modelos

## Problema 1: implementação

```
modelo = Model(CPLEX.Optimizer)
@variable(modelo, x[i in 1:4] >=0)
@objective(modelo, Min, 3*x[1]-11*x[2]+5*x[3]+x[4] )
@constraints modelo begin
    x[1]+5*x[2]-3*x[3]+6*x[4] <=7
    -x[1]+x[2]+x[3]-2*x[4] >= 3
end

optimize!(modelo)
has_values(modelo)
termination_status(modelo)
```

# Resolvendo alguns modelos

## Problema 2: Lot sizing ilimitado

$$\begin{array}{ll}\text{minimize} & \sum_t (c_t x_t + f_t y_t + h_t s_t) \\ \text{sujeito a} & s_{t-1} + x_t - s_t = d_t, \quad \forall t \\ & x_t \leq M y_t, \quad \forall t \\ & x_t, s_t \geq 0, \quad \forall t \\ & y_t \in \{0, 1\}, \quad \forall t\end{array}$$

onde:

- $T$  é o número de períodos,  $d_t$  é a demanda,  $f_t$  custo de setup,  $c_t$  custo de produção e  $h_t$  custo de estoque
- $y_t$  se produz ou não no período  $t$ ,  $x_t$  é quanto produzir e  $s_t$  quanto estocar

# Resolvendo alguns modelos

## Problema 2: implementação

T=7

```
c=round.(1 .+ 3*rand(1,T)); f=round.(1 .+ 3*rand(1,T))  
h=round.(1 .+ 3*rand(1,T)); d=round.(100 .+ 10*rand(1,T))  
s0=15; M=maximum(d)
```

```
modelo = Model(CPLEX.Optimizer)
```

```
@variables modelo begin
```

```
    x[t in 1:T] >=0
```

```
    s[t in 1:T] >=0
```

```
    y[t in 1:T], Bin
```

```
end
```

```
@objective(modelo,Min,sum(c[t]*x[t] + f[t]*y[t] + h[t]*s[t] for t in 1:T)
```

```
@constraint(modelo, s0+x[1]-s[1] .== d[1])
```

```
@constraint(modelo,[t in 2:T], s[t-1]+x[t]-s[t] .== d[t])
```

```
@constraint(modelo, [t in 1:T], x[t] <= M*y[t])
```

```
optimize!(modelo)
```

```
has_values(modelo)
```

```
termination_status(modelo)
```

```
value.(x)
```

# Resolvendo alguns modelos

## Problema 3: Problema de schedule

$$\begin{array}{ll}\text{minimize} & \sum_j w_j y_j \\ \text{sujeito a} & \sum_j a_{tj} y_j \geq d_t, \quad \forall t \\ & y_t \geq 0 \text{ e inteiro}, \quad \forall t\end{array}$$

onde:

- $y_t$  número de trabalhadores a ser determinado no tempo  $t$
- $d_t$  demanda de trabalhadores exigidos no tempo  $t$ ,
- $w_t$  salário no turno  $t$
- $a_{tj} = 1$  se o turno  $j$  cobre o tempo  $t$  e 0 c.c

# Resolvendo alguns modelos

**Tabela:** Time windows para o problema de schedule dos trabalhadores

Time Window	Turnos				Demanda
	1	2	3	4	
6-9h	x			x	55
9-12h	x				46
12-15h	x	x			59
15-18h		x			23
18-21h		x	x		60
21-24h			x		38
0-3h			x	x	20
3-6h				x	30
custo por turno	\$135	\$140	\$190	\$188	

# Resolvendo alguns modelos

## Problema 3: implementação

```
T=8;n=4
w=[135,140,190,188]
d=[55,46,59,23,60,38,20,30]
A=[1 0 0 1;1 0 0 0;1 1 0 0;0 1 0 0;
   0 1 1 0;0 0 1 0;0 0 1 1;0 0 0 1]

modelo = Model(CPLEX.Optimizer)
@variables modelo begin
    y[t in 1:n] >=0, Int
end
@objective(modelo, Min, sum(w[j]*y[j] for j in 1:n) )
@constraint(modelo, [t in 1:T], sum(A[t,j]*y[j] for j in 1:n) >= d[t])

optimize!(modelo)
has_values(modelo)
termination_status(modelo)
value.(y)
relative_gap(modelo)
```



# Resolvendo alguns modelos

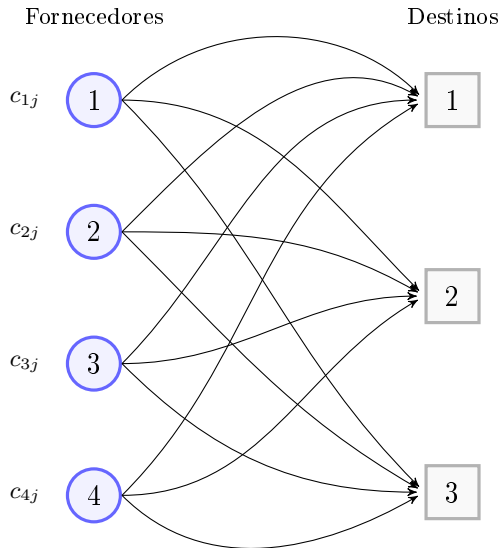
## Problema 4: problema do transporte

$$\begin{array}{ll}\text{minimize} & \sum_i \sum_j (c_{ij}x_{ij} + f_{ij}y_{ij}) \\ \text{sujeito a} & \sum_i x_{ij} = d_j, \quad \forall j \\ & x_{ij} \leq M y_{ij}, \quad \forall i, \forall j \\ & x_{ij} \geq 0, \quad \forall i, \forall j \\ & y_{ij} \in \{0, 1\}, \quad \forall i, \forall j\end{array}$$

onde:

- $y_{ij}$ : Decide se o fornecedor  $i$  envia para o destino  $j$  ( $= 1$ ), 0 caso contrário. Se sim, quanto ( $x_{ij}$ )
- $c_{ij}$  custo unitário de envio, custo fixo  $f_{ij}$  e  $d_j$  a demanda do destino  $j$ .

# Resolvendo alguns modelos



# Resolvendo alguns modelos

## Problema 4: implementação

```
n=4;m=3
c=rand(1:20,n,m); f=rand(1000:2000,n,m)
d=rand(50:100,m); M=sum(d)
modelo = Model(CPLEX.Optimizer)
@variables modelo begin
    x[i in 1:n,j in 1:m] >=0
    y[i in 1:n,j in 1:m], Bin
end
@objective(modelo, Min, sum(c[i,j]*x[i,j] + f[i,j]*y[i,j]
                           for i in 1:n for j in 1:m) )
@constraints modelo begin
    con1[j in 1:m], sum(x[i,j] for i in 1:n) == d[j]
    con2[i in 1:n,j in 1:m], x[i,j] <= M*y[i,j]
end
optimize!(modelo)
has_values(modelo)
termination_status(modelo)
value.(y)
value.(x)
```

## Resolvendo alguns modelos

**Problema 5:** problema de corte multiobjetivo

$$\begin{array}{ll}\text{minimize} & z_1 = \sum_j \left( L - \sum_i a_{ij} \cdot \ell_i \right) \cdot x_j \\ \text{minimize} & z_2 = \sum_j y_j \\ \text{sujeito a} & \sum_j a_{ij} \cdot x_j \geq d_i, \quad \forall i \\ & x_j \leq M y_j, \quad \forall j \\ & x_j \geq 0 \text{ e inteiro}, \quad \forall j \\ & y_j \in \{0, 1\}, \quad \forall j\end{array}$$

onde:

- $x_j$  é o número de padrões  $j$  são usados e  $y_j$  indica seu uso ou não
- $d_i$  indica a demanda do item  $i$
- $L$  denota a largura da bibina mestre,  $\ell_i$  a largura do item  $i$  e  $a_{ij} \in \mathbb{Z}$  denota o número de itens do tipo  $i$  no padrão de corte  $j$

## Resolvendo alguns modelos

**Problema 5:** para resolvê-lo, primeiro minimizamos cada uma das funções objetivo envolvidas.

$$\begin{array}{ll} \text{minimize} & z_2 = \sum_j y_j \\ \text{sujeito a} & \sum_j a_{ij} \cdot x_j \geq d_i, \quad \forall i \\ & x_j \leq M y_j, \quad \forall j \\ & x_j \geq 0 \text{ e inteiro}, \quad \forall j \\ & y_j \in \{0, 1\}, \quad \forall j \end{array}$$

## Resolvendo alguns modelos

**Problema 5:** para resolvê-lo, primeiro minimizamos cada uma das funções objetivo envolvidas.

$$\begin{array}{ll} \text{minimize} & z_1 = \sum_j \left( L - \sum_i a_{ij} \cdot \ell_i \right) \cdot x_j \\ \text{sujeito a} & \sum_j a_{ij} \cdot x_j \geq d_i, \quad \forall i \\ & x_j \leq M y_j, \quad \forall j \\ & x_j \geq 0 \text{ e inteiro}, \quad \forall j \\ & y_j \in \{0, 1\}, \quad \forall j \end{array}$$

## Resolvendo alguns modelos

**Problema 5:** com os valores de mínimo e máximo para a função  $z_2 \in [z_2^{min}, z_2^{max}]$  do passo anterior, nós usamos o  $\varepsilon$ -restrito para obter as soluções de Pareto ótimas

$$\begin{array}{ll} \text{minimize} & z_1 = \sum_j \left( L - \sum_i a_{ij} \cdot \ell_i \right) \cdot x_j \\ \text{sujeito a} & \sum_j a_{ij} \cdot x_j \geq d_i, \quad \forall i \\ & x_j \leq M y_j, \quad \forall j \\ & \sum_j y_j \leq \varepsilon \\ & x_j \geq 0 \text{ e inteiro}, \quad \forall j \\ & y_j \in \{0, 1\}, \quad \forall j \end{array}$$

para cada  $\varepsilon \in [z_2^{min}, z_2^{max}]$

# Resolvendo alguns modelos

## Problema 5: implementação da minimização de cada função objetivo

```
function Minimizacao_z1_e_z2(A,d,L,l,M,m,n,w)
    modelo = Model(CPLEX.Optimizer)
    @variables modelo begin
        x[j in 1:n] >=0, Int
        y[j in 1:n], Bin
    end
    @objective(modelo,Min,w*sum((L - sum(A[i,j]*l[i] for i in 1:m))*x[j]
                                for j in 1:n) +
                                (1-w)*sum( y[j] for j in 1:n) )

    @constraints modelo begin
        con1[i in 1:m], sum(A[i,j]*x[j] for j in 1:n) >= d[i]
        con2[j in 1:n], x[j] <= M*y[j]
    end
    optimize!(modelo)

    return (value.(x),value.(y))
end
```



# Resolvendo alguns modelos

## Problema 5: implementação do subproblema $\varepsilon$ -restrito

```
function E_restrito(A,d,L,l,M,m,n,E)
    modelo = Model(CPLEX.Optimizer)
    @variables modelo begin
        x[j in 1:n] >=0, Int
        y[j in 1:n], Bin
    end
    @objective(modelo, Min, sum( (L - sum(A[i,j]*l[i] for i in 1:m))*x[j]
                                for j in 1:n) )

    @constraints modelo begin
        con1[i in 1:m], sum(A[i,j]*x[j] for j in 1:n) >= d[i]
        con2[j in 1:n], x[j] <= M*y[j]
        con3, sum( y[j] for j in 1:n) <= E
    end
    optimize!(modelo)
    z1 = sum((L - sum(A[i,j]*l[i] for i in 1:m))*value(x[j]) for j in 1:n)
    z2 = sum(value(y[j]) for j in 1:n)

    return (value.(x),value.(y),z1,z2)
end
```

# Resolvendo alguns modelos

## Problema 5: colocando o problema restrito em um looping:

```
let
    global X = zeros(m,0); global Y=zeros(m,0);
    global z1 = zeros(1,0);global z2 = zeros(1,0);
    for E = z2min:z2max
        (XE,YE,z1e,z2e) = E_restrito(A,d,L,l,M,m,n,E)
        X = [X XE];Y = [Y YE];
        z1 = [z1 z1e];z2 = [z2 z2e];
    end
end
```

# Resolvendo alguns modelos

## Importante:





- Para salvar a solução de um problema, pode usar o comando `save` inserindo o nome da variável e a extensão `.jld2`.
- Para este formato, instalar e rodar as bibliotecas `FileIO` `JLD2` e usando:

```
FileIO.save("x_values.jld2", "RH", x)
```

ou seja, salva o vetor das variáveis ótimas `x` no arquivo `x_values.jld2`

- Para fazer o load do arquivo salvo, basta entrar com `FileIO.load("x_values.jld2", "RH")`

# Referências I

-  D. P. Corina, L. S. Jose-Robertson, A. Guillemin, J. High, and A. R. Braun, “Language lateralization in a bimanual language,” *Journal of Cognitive Neuroscience*, vol. 15, no. 5, pp. 718–730, 2003.
-  J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, “Julia: A fresh approach to numerical computing,” *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.
-  A. Edelman, “Julia: A fresh approach to technical computing and data processing,” tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE CAMBRIDGE, 2019.
-  I. Balbaert, *Getting Started with Julia Programming: Enter the exciting world of Julia, a high-performance language for technical computing*. Packt Publishing, 2015.

## Referências II



I. Dunning, J. Huchette, and M. Lubin, “JuMP: A modeling language for mathematical optimization,” *SIAM Review*, vol. 59, pp. 295–320, 2017.