A Gentle Introduction to julia for Optimisation

FROM A MATLAB-USER PERSPECTIVE

THIBAUT CUVELIER

23 SEPTEMBER, 2016

A few words about the course

- Goal: you can **model** nontrivial situations as MIPs, including **implementing** your model and **solving** it

- Two projects: modelling and implementing
  - First one: optimisation in a video game
    - **Get warmed up!**
  - Second one: more complex and realistic
    - (Most probably) organised as a **challenge**

Website

- http://www.montefiore.ulg.ac.be/~tcuvelier/do
  - Statements for the exercise sessions
  - Project information
  - Exercise book

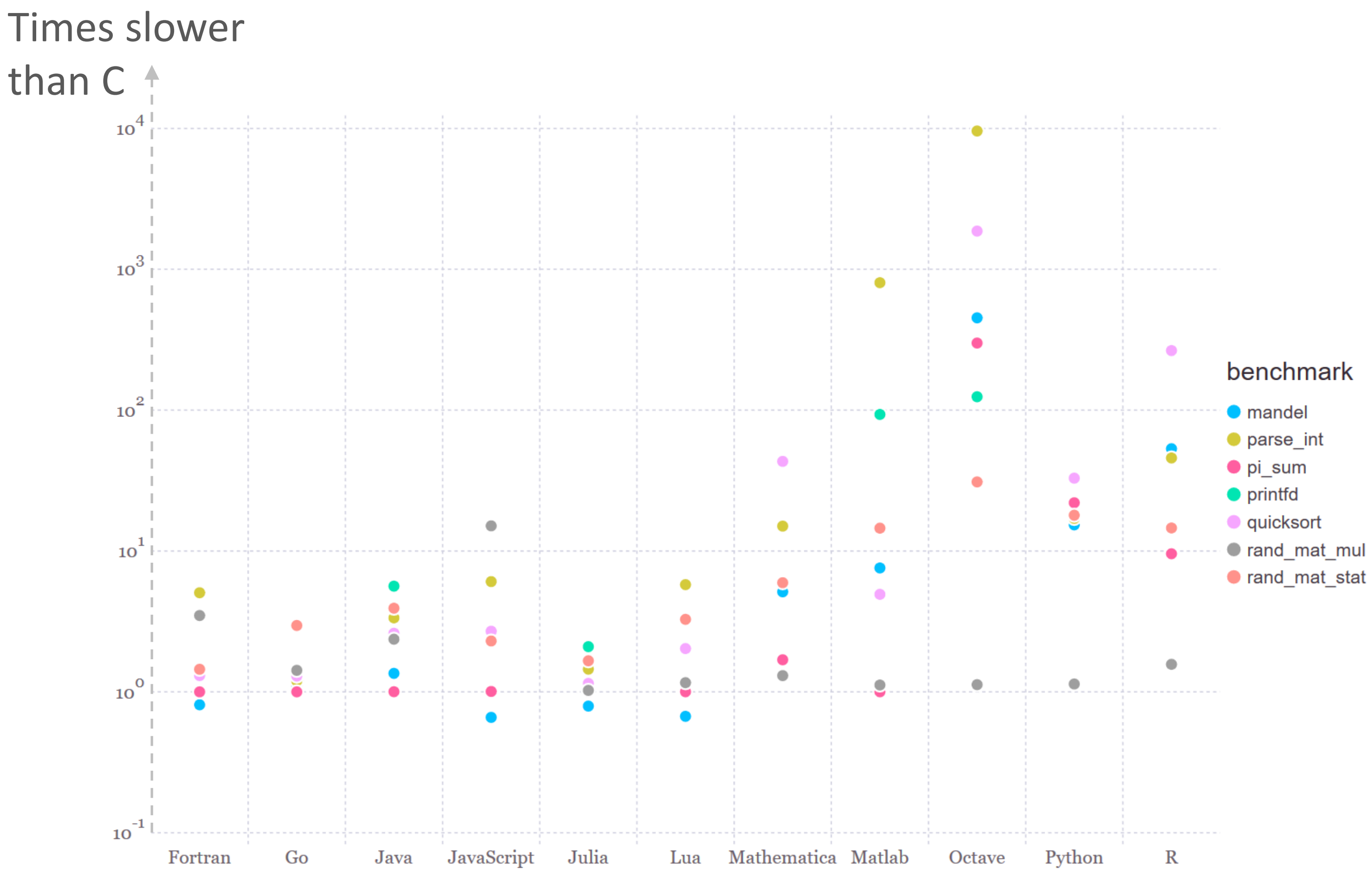
What is julia?

- A programming language
  - For scientific computing first
  - But still dynamic, “modern”… and extensible!

- Often compared to MATLAB, with a similar syntax…
  - … but much faster!
    - … without the need for compilation!
  - … with a large community!
  - … and free (MIT-licensed)!

How fast is julia?

Comparison of run time between several languages and C



Source: http://julialang.org/benchmarks/

Why julia in this course?

- Vibrant optimisation community: JuliaOpt

- Very nice modelling layers: JuMP and Convex.jl
  - Convenient to use: close to actual mathematical form

Step 1: install julia

- Website: http://julialang.org/
- Download the latest stable version (0.5 series)
  - In case of troubles, you can also use Julia 0.4

Tools that might be of use…

- An IDE:
  - Juno: Atom with Julia extensions
    - Install Atom: https://atom.io/
    - Install Juno: in Atom, **File > Settings > Install**, search for **uber-juno**
  - JuliaDT: Eclipse with Julia extensions
    - Much more experimental!
- A notebook environment: IJulia
  - See later

Step 2: basic syntax

- Variable definition: just like in MATLAB

```
julia> a = 4.2
4.2
```

- Arithmetic: as expected

```
julia> (a + 1)^2
27.040000000000003
```

- Compound assignments work **(unlike in MATLAB)**

```
julia> a *= 2
8.4
```

Array syntax

- Use brackets around, commas or spaces inside:

```
julia> a = [1, 2]    # Equivalent to: [1 2]
2-element Array{Int64,1}:
 1
 2
```

- **Indexing is done with brackets** (like in C, Java…) starting at 1 (like in MATLAB)

```
julia> a[1]
1
```

- Matrices:

```
julia> a = [[1, 2] [3, 4]]
2x2 Array{Int64,2}:
 1  3
 2  4
```

  - Use commas to separate dimensions: `a[1, 2]`
  - **A vector is not a $1 \times n$ or $n \times 1$ matrix!**

Array ranges

- Ranges work like in MATLAB, Python, or Fortran:

```
julia> a = [1, 2, 3, 4, 5];
julia> a[2:4]
3-element Array{Int64,1}:
 2
 3
 4
julia> a[1:2:5]
3-element Array{Int64,1}:
 1
 3
 5
```

Array creation

- Arrays of zeroes and ones:

```
julia> zeros(2)
2-element Array{Float64,1}:
 0.0
 0.0
julia> ones(2, 2)
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0
```

- Arrays from ranges:

```
julia> collect(1:3)
3-element Array{Int64,1}:
 1
 2
 3
```

One-line functions

- Close to mathematical way of writing the function:

```
julia> f(x) = x^2
f (generic function with 1 method)

julia> f(2)
4
```

Complex functions

- Use the keyword `function`
  Last expression is returned automatically! (Like in Scala)

```
julia> function f(x)
         return x^2
       end
f (generic function with 1 method)

julia> function f(x)
         x^2
       end
f (generic function with 1 method)
```

- Void functions? Use either:

```
return
return nothing
```

`if` conditions

- How to write the conditional expressions?
  - Usual operators: &&, ||, !
  - Use of parentheses to group terms
- Example:

```
julia> a = 1.
julia> if a >= .1 && a <= 1.1
         print(a)
       elseif abs(a - 2.0) < 1.e-8
         print(“Oh!”)
       else
         print(1 - a)
       end
1
```

for loops

- Prefer iterating over ranges (like in MATLAB):

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

- The same syntax can iterate through an array (like in MATLAB):

```
julia> for i in [1, 42]
         println(i)
       end
1
42
```

Writing files

- For actual developments, your code must survive a Julia shell session
- Hence: write your code in files!
- How?
  - Use IJulia notebooks
  - Use simple text files and include them from the shell:

```
julia> include(”/path/to/file.jl”)
julia> include(”C:\\path\\to\\file.jl”)
```

  - The latter will be used to evaluate your projects
- Note: functions do not need to have the same name as the

Step 3: a vibrant community

- Julia has a large community
- Hence many extension packages are available!
  - For plotting: Plots.jl, Gadfly, Winston, etc.
  - For graphs: Graphs.jl, LightGraph.jl, Graft.jl, etc.
  - For statistics: DataFrames.jl, Distributions.jl, TimeSeries.jl, etc.
  - For machine learning: JuliaML, ScikitLearn.jl, etc.
  - For Web development: Mux.jl, Escher.jl, WebSockets.jl, etc.
  - For this course, mainly JuMP and Convex (see later)

- A list of all registered packages: http://pkg.julialang.org/

Package manager

- How to install a package?

```
julia> Pkg.add(”PackageName”)
```

  - No .jl in the name!

- Import a package:

```
julia> import PackageName
```

- How to remove a package?

```
julia> Pkg.rm(”PackageName”)
```

- All packages are hosted on GitHub
  - Usually grouped by interest: JuliaStats, JuliaML, JuliaWeb, JuliaOpt, JuliaPlots, JuliaQuant, JuliaParallel, JuliaMaths…
  - See a list at http://julialang.org/community/

Something else than a console?

- The default console is not the sexiest interface
  - The community provides better ones!

- **Purely online**, free: JuliaBox
  - https://juliabox.com/

- Offline, based on Jupyter (still in the browser): IJulia
  - Install with:

```
julia> Pkg.add(”IJulia”)
```

  - Run with:

```
julia> using IJulia; notebook()
```

Step 4: plotting

- Plots.jl: an interface to multiple plotting engines (e.g. GR or matplotlib)
- Install the interface and one plotting engine (GR is fast):

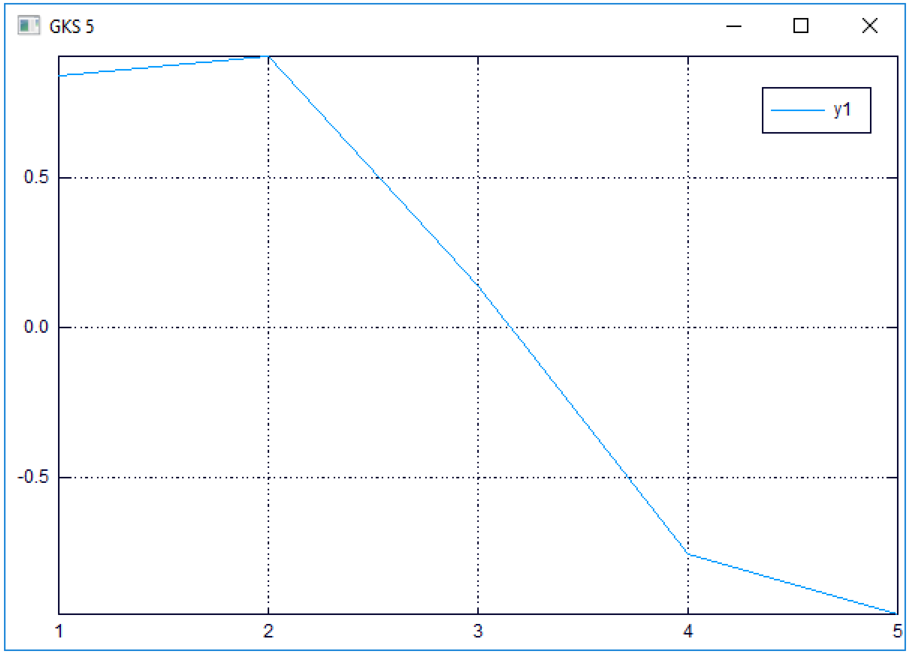
```
julia> Pkg.add(”Plots”)
julia> Pkg.add(”GR”)
julia> using Plots
```

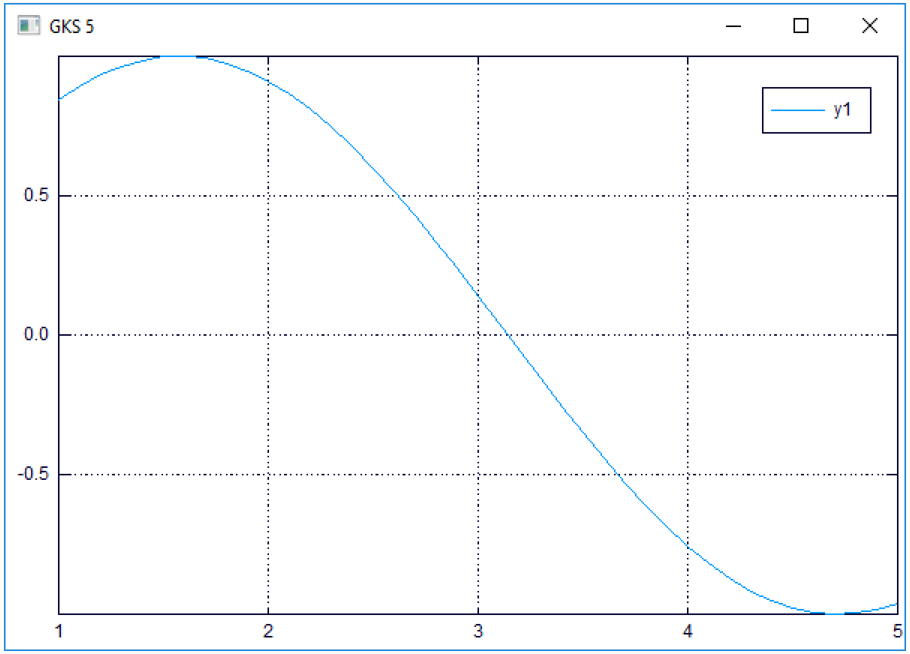
- If you prefer to use matplotlib (nicer plots, much slower than GR):
  - Install it locally (e.g. with Anaconda on Windows)
  - Then use the PyPlot module:

```
julia> Pkg.add(”PyPlot”)
```

- Documentation: https://juliaplots.github.io/

Basic plots

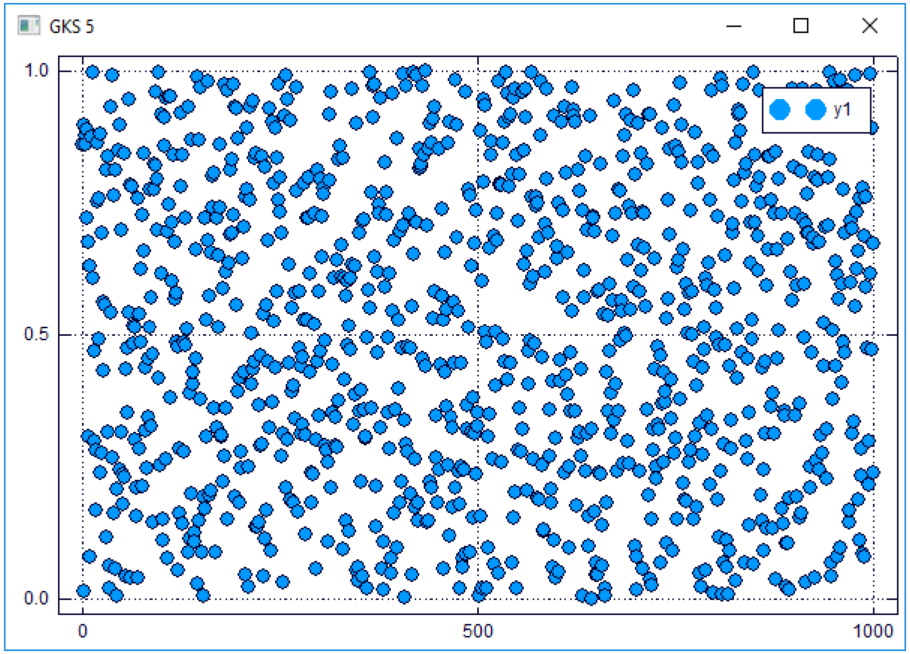
- Basic plot:
```
julia> plot(1:5, sin(1:5))
```


- Plotting a mathematical function:
```
julia> plot(sin, 1:.1:5)
```


More plots

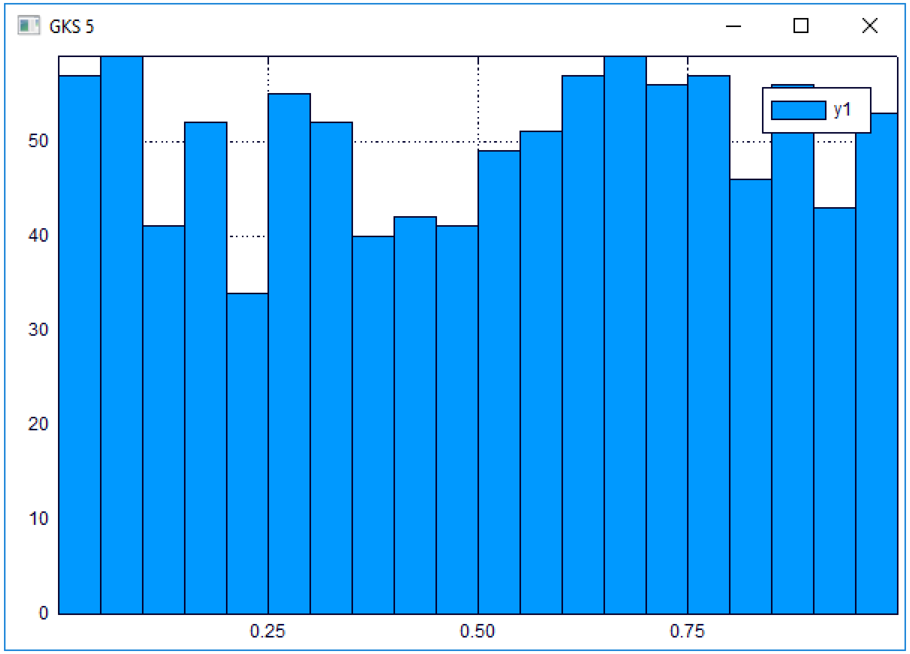
- Scatter plot:

```
julia> scatter(rand(1000))
```



- Histogram:

```
julia> histogram(rand(1000),
                 nbins=20)
```



Step 5: optimisation with JuMP

- After all, this is an optimisation course!
- JuMP's goal: provide an easy way to translate optimisation programs into code
- First: install it along with a solver

```
julia> Pkg.add(”JuMP”)
julia> Pkg.add(”Cbc”)
julia> using JuMP
```

$$
\begin{aligned}
&\max x + y \\
&\text{s.t. } 2x + y \leq 8 \\
&\qquad 0 \leq x \leq +\infty \\
&\qquad 1 \leq y \leq 20
\end{aligned}
$$

```
m = Model()
@variable(m, x >= 0)
@variable(m, 1 <= y <= 20)
@objective(m, Max, x + y)
@constraint(m, 2 * x + y <= 8)
solve(m)
```

Retrieve a solution

- When solving a model, JuMP returns a value:
  - `solve(m) == :Optimal`: found the optimal solution
  - `solve(m) == :Unbounded`: the optimal objective function is infinite
  - `solve(m) == :Infeasible`: there is no solution to the problem

Check your model!

- Objective value: `getobjectivevalue(m)`
- Variable value: `getvalue(x)`, even if x is a vector or a matrix
- Nice way to print the whole model in a readable way: `print(m)`

More complex JuMP: variables

- How to model a vector of variables?

  $$x_t \in \mathbb{R}^T$$

```
@variable(m, x[1:T])
```

- Matrix of variables?

  $$x_{t,s} \in \mathbb{R}^{T \times S}$$

```
@variable(m, x[1:T, 1:S])
```

More complex JuMP: constraints

- Constraints over a range?

$$\sum_{s \in S} x_{t,s} = 1, \qquad \forall t \in T$$

```
for t in 1:T
  @constraint(m, sum(x[t, :]) == 1)
end
```

- Dot product?

$$\sum_{t \in T} a_t \, x_t = b$$

```
@constraint(m, dot(a, x) == b)
```

- Arbitrary sum?

$$\sum_{t \in T} a_t \, x_t = b$$

```
@constraint(m, b ==
    sum{a[t] * x[t], t = 1:T})
```

Variable type

- A binary variable?

```
@variable(m, x, Bin)
```

- An integer variable?

```
@variable(m, x, Int)
@variable(m, x <= 10, Int)
```

- A semi-continuous variable? (i.e. zero or an interval)

```
@variable(m, 1 <= x <= 2,
          SemiCont)
```

- A semi-integer variable?

```
@variable(m, 1 <= x <= 2,
          SemiInt)
```

Complex indexing to define variables

- A triangular matrix of variables?
  $x_{i,j}, \quad i \in [1, I], j \in [i, J]$

```
@variable(m, x[i=1:I,j=i:J])
```

- A vector of variables whose indices satisfy a condition?
  $x_i, \quad i \in [1,3,5 \dots 11]$

```
@variable(m,
        x[i=1:12; isodd(i)])
```

- Even with multiple conditions!
  $x_{i,j}$, where:
  $i \in [1,3,5 \dots 19]$
  $j \in [i, 50]$,
  $i + j$ even
  $2i + j \leq 70$

```
@variable(m,
        x[i=1:20, j=i:50;
          iseven(i+j) &&
          2*i+j <= 70])
```

Conic constraints (not for this course)

- Second-order cone: only with the canonical form $||\boldsymbol{A}\,\boldsymbol{x} - \boldsymbol{b}||_2^2 + \boldsymbol{b}^T\boldsymbol{x} + c \leq 0$

```
@constraint(m, norm(A * x + b)
  + dot(b, x) + c <= 0)
```

Alternative syntax: `norm2{2 * x[i], i = 1:I}`

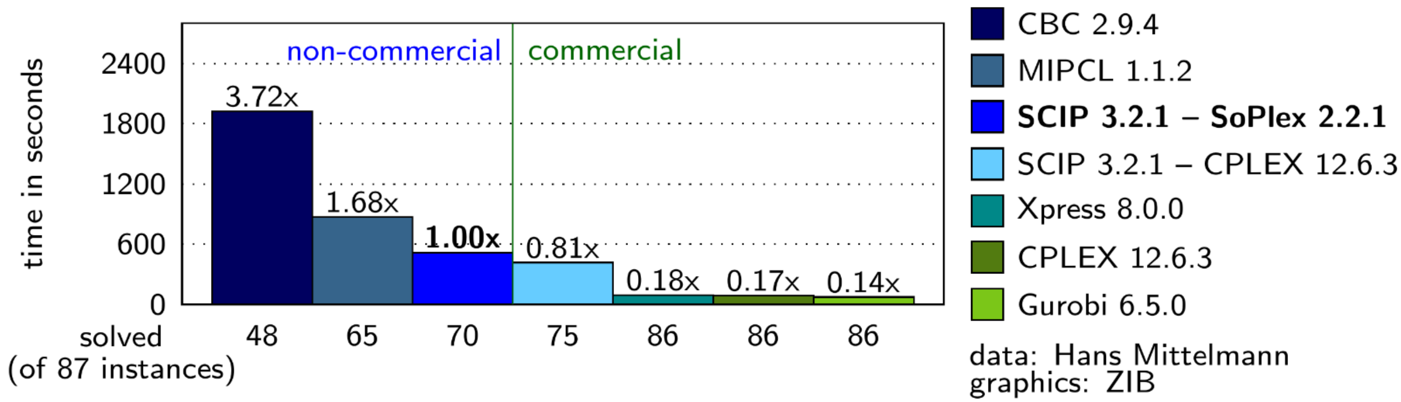
- SDP cone: impose a variable matrix as SDP

```
@variable(m, x[1:N, 1:N], SDP)
```

- SDP cone: impose a variable matrix as symmetric, add a semidefinite constraint

```
@variable(m, x[1:N, 1:N],
                    Symmetric)
@Sdconstraint(m, X >= eye(N))
```

Why commercial solvers?



Hence you are encouraged to use Gurobi: much faster, not so complicated to install

Install a faster optimisation solver: Gurobi

- Create an account **with your student email address** on http://www.gurobi.com/
- Download and install Gurobi
- Ask for an academic license online: http://www.gurobi.com/downloads/user/licenses/free-academic
- Activate the software **from within the university network**
- In Julia:
```
julia> Pkg.add(”Gurobi”)
julia> using Gurobi
```
- Force JuMP to use it:
```
julia> m = Model(solver=GurobiSolver())
```

Step 6: optimisation with Convex.jl

(not for this course)

- JuMP has limitations for convex optimisation:
  - Only its defined standard forms
  - Only SOCP and SDP (no exponential cones, no geometric programming)
- Convex.jl understands "disciplined convex programming"
  - More natural way of writing the constraints.
- Install it with:

```
julia> Pkg.add("Convex")
julia> using Convex
```

$$
\begin{aligned}
&\max x + y \\
&\text{s.t. } 2x + y \leq 8 \\
&\quad 0 \leq x \leq +\infty \\
&\quad 1 \leq y \leq 20
\end{aligned}
$$

```
x = Variable(Positive())
y = Variable(Positive())
p = minimize(x + y)
p.constraints += y >= 1
p.constraints += y <= 20
p.constraints += 2 * x + y <= 8
solve!(problem)
```

Retrieve a solution

- Status
  - `p.status == :Optimal`: found the optimal solution
  - `p.status == :Unbounded`: the optimal objective function is infinite
  - `p.status == :Infeasible`: there is no solution to the problem

Check your model!

- Objective value: `p.optval`
- Variable value: `x.value`, even when `x` is a vector or a matrix

More complex stuff

- How to model a vector of variables?

  $$x_t \in \mathbb{R}^T$$

```
x = Variable(T)
```

- Matrix of variables?

  $$x_{t,s} \in \mathbb{R}^{T \times S}$$

```
x = Variable(T, S)
```

- Constraints over a range?

  $$\sum_{s \in S} x_{t,s} = 1, \qquad \forall t \in T$$

```
for s in 1:S
  p.constraints += sum(x[:, s]) == 1)
end
```

- Dot product?

  $$\sum_{t \in T} a_t \, x_t = b$$

```
p.constraints += dot(a, x) == b
```

Integer variable type

- A binary variable? `x = Variable(:Bin)`
- An integer variable? `x = Variable(:Int)`

Examples of conic constraints

(not for this course)

- Second-order cone:

```
p.constraints += norm(A * x + b) <= c
```

- SDP cone: impose a variable matrix as SDP

```
x = Semidefinite(N) # Always a square matrix!
```

- SDP cone: add a semidefinite constraint

```
x = Variable(3, 3)
y = Variable(3, 1)
z = Variable()
p.constraints += [x y; y’ z] in :SDP
```