About Julia

Julia is a young programming language (development started in 2009) and it is gaining a lot of attention in areas where efficient and accurate computations are needed. The goal is to create a free language that is both high-level and fast. Julia uses a just-in-time (JIT) compiler and compiles all code (by default) to machine code before running it. Julia can be compiled to binary executables using a package for it supporting all Julia features.

To download and install Julia:

```
https://julialang.org/downloads/platform/
```

About JuMP

Julia itself is a programming language, and JuMP is a package for Julia which makes mathematical modelling easy.

```
https://jump.dev/JuMP.jl/stable/
```

In order to solve optimization models, a solver is also needed. JuMP supports a number of open-source and commercial solvers for a variety of problem classes, including linear, mixed-integer, second-order conic, semidefinite, and nonlinear programming.

```
https://jump.dev/JuMP.jl/stable/installation/#Supported-solvers
```

We will make use of `HiGHS` and `Ipopt`, two solvers which are free to use and easy to install.

Solver HiGHS

HiGHS is a high performance software for linear optimization. It is a package of open source serial and parallel solvers for large-scale sparse linear programming (LP), mixed-integer programming (MIP), and quadratic programming (QP) models.

```
https://highs.dev
```

Solver Ipopt

Ipopt (Interior Point Optimizer, pronounced “Eye-Pea-Opt”) is an open source software package for large-scale nonlinear optimization.

```
https://coin-or.github.io/Ipopt/
```

Overview

```
Julia                ( programming language )
└─ JuMP              ( package for modelling )
   ├─ HiGHS          ( solver LP / MIP )
   └─ Ipopt          ( solver NLP )
```

Packages needed

The following packages must be installed in order to run and solve linear and non-linear optimization problems. For courses using Julia/JuMP, these packages are pre-installed on the computer system in our computer labs.

To install a package, use the command `Pkg.add("package_name")`.

```
#------------------------------------------------------------------------------
# Packages  --  import and install
#------------------------------------------------------------------------------
import Pkg;
Pkg.add("JuMP")                          # optimization modelling package
Pkg.add("HiGHS")                         # solver for LP and MIP
Pkg.add("Ipopt")                         # solver for NLP
Pkg.add("Plots")                         # plot library
Pkg.add("DataFrames")                    # structures data in a nice way
Pkg.add("JSON3")                         # read data from JSON-files
#------------------------------------------------------------------------------
```

To make use of a package, include the command `using` in the following way.

```
#------------------------------------------------------------------------------
# Packages  --  use packages in your code
#------------------------------------------------------------------------------
using JuMP
using HiGHS
using Ipopt
using Plots
using DataFrames
using JSON3
#------------------------------------------------------------------------------
```

Files

Files with code for Julia use the extension "`.jl`" and are executed (run) using the `include("filename.jl")` command. To edit Julia files, use your favorite text editor (such as `emacs`, `vim`, `atom` or `gedit`). A great option is `vscode`, a powerful software development environment.

Get started

To start Julia, do the following.

1. Start a terminal
2. Go to the correct folder (using `cd` commands)
3. Type: `julia`

To run the code given for Example 1 on page 8, write `include("example1.jl")` in Julia.

Syntax

Here are some examples of the most common commands in Julia/JuMP. This guide is not by any means complete, any constructive feedback is most welcome.

Optimization models

A mathematical optimization model is composed of parameters, variables, constraints, and an objective function. Parameters can be provided either by vectors and matrices directly in Julia, or specified in a separate file (using some standard data format like `csv` or `JSON`).

The JuMP package uses `Model` objects to describe an optimization problem. The following example gives an overview (pseudocode) of what it looks like. (More details will follow.)

```
LP_model    = Model()                  # create an empty model object "LP_model"
set_optimizer( LP_model, ... )         # specify which solver to use

@variable( LP_model, ... )             # add variables to the model
@constraint( LP_model, ... )           # add constraints to the model
@objective( LP_model, ... )            # add an objective function to the model

optimize!( LP_model )                  # solve the optimization model

objective_value( LP_model )            # display the objective value
value.( var_name )                     # display the value of variable "var_name"
```

Parameters

It is possible to use scalars, vectors and matrices to collect and store numerical values (digits) or text strings (names). Avoid using å, ä, ö in names. Here are some example:

```
n = 9                                  # parameter n is assigned the value 9

c = [ 70  60  50 ]                     # 1x3 matrix with values 70, 60 and 50,
                                       # i.e. c[1] = 70, c[2] = 60, c[3] = 50

A = [ 12  14   7  11 ;                 # matrix A is assigned the given values,
      16   9  10  12 ;                 # i.e. A[1,1] = 12, A[1,2] = 14 and so on
      10  21  12  19 ]

S       = [ 1 2 3 4 5 6 ]              # 1x6 matrix with numbers 1,2,...,6
PRIMES  = [ 3 5 7 11 13 ]              # 1x5 matrix with the first 5 odd primes
N       = [ i for i in 1:n ]           # vector with numbers 1..n

COLOR   = [ "red"  "white"  "blue" ]
ANIMAL  = [ "elk"  "horse"  "duck"  "fox"  "cat"  "dog"]
```

It is also possible to define tuples, for example, a set of pairs:

```
# Create vector of tuples with pairwise ordered numbers
PAIRS   = [ (i,j) for i in N, j in N  if  i < j ]
```

Notice how you can filter elements in the sets using the "`if`" syntax.

Model object

Start by creating an empty model object, and optionally specify which solver to use.

```
# Create an empty model object "model"
model     = Model()

# Create a model object "model_MIP", and use HiGHS as solver (LP + MIP problems)
model_MIP = Model(HiGHS.Optimizer)

# Create a model object "model_NLP", and use Ipopt as solver (NLP problems)
model_NLP = Model(Ipopt.Optimizer)
```

Solvers can be assigned (or changed) using the command `set_optimizer`.

```
set_optimizer(model, HiGHS.Optimizer)
```

Variables

Variables are defined using the command `@variable`. It is possible to declare different kinds of variables, such as binary, integer and continuous variables. It is also possible to declare upper and lower bounds. Here are some examples:

```
@variable(model, x_free)              # continuous variable (free)
@variable(model, x_lower >= 0)        # continuous variable, lower bound 0
@variable(model, x_upper <= 1)        # continuous variable, upper bound 1
@variable(model, 2 <= x_inter <= 5)   # continuous variable, 2 <= x <= 5

@variable(model, x_bin, Bin)          # binary variable, that is, either 0 or 1
@variable(model, x_int, Int)          # integer variable, only integer values OK
```

Variables with indices

Variables are often defined with an index, for example, one for each time step t, or for different types of products i. It is also possible to declare variables with multiple indices, for example X_{ij}. The following syntax is used:

```
@variable(model, x[1:6] >= 0 )       # variable vector, positive x[i], i=1,...,6
@variable(model, x[S]   >= 0 )       # equivalent declaration of x[i], using S

@variable(model, z[1:3] >= 0, Int )  # positive integer variables z[i], i=1,..,3
@variable(model, X[1:3,1:4] )        # variable matrix X[i,j], i=1,..,3 j=1,..,4

@variable(model, y[COLOR], Bin )     # binary variable for each color in COLOR
@variable(model, Y[ANIMAL,COLOR] )   # variables for all animal-color combinations

@variable(model, u[PAIRS], Bin)      # binary variable indexed over the set of
                                     # tuples PAIRS, i.e. u[(i,j)]

@variable(model, q[1:8,PAIRS] >=0 )  # variable with two indices, where the last
                                     # index comes from PAIRS, i.e. q[k,(i,j)]
```

Summation

The command `sum` can be used to state a summation, which is often useful both in the objective function and in the constraints. Here are some examples:

```
sum( x[j] for j in 1:n )            # the sum x[1] + x[2] + ... + x[n]
sum( x[j] for j in N )              # same thing, now using set N

sum( y[i] for i in COLOR )          # the sum  y["red"] + y["white"] + y["blue"]
```

Double summation

It is of course possible to model double and triple summations.

$\sum_{i=1}^{3}\sum_{j=1}^{4} X_{ij}$ $\Leftrightarrow$ `sum( X[i,j] for i in 1:3, j in 1:4 )`

$\sum_{k=1}^{8}\sum_{(i,j)\in \mathrm{PAIRS}} q_{k,(i,j)}$ $\Leftrightarrow$ `sum( q[k,(i,j)] for k in 1:8, (i,j) in PAIRS )`

Conditional summation

Sometimes it is helpful to summarize over a restricted set, or define rules for when to include a certain value.

```
# Sum over i in the range of 1 to 10 such that i is an element of PRIMES
sum( x[i] for i in 1:10  if  i in PRIMES )

# Sum over i in the set PRIMES and j in the set PRIMES, such that i is in
# the range of 2 to 5, and j is in the range of 3 to 9
sum( x[i,j] for i in PRIMES, j in PRIMES  if  2 <= i <= 5  &&  3 <= j <= 9 )

# For a fix and given j, sum over all i such that (i,j) is an element of PAIRS
sum( u[(i,j)] for i in N  if  (i,j) in PAIRS )
```

Objective function

An objective function is defined using the command `@objective`, and it is necessary to specify if it is a maximization (`Max`) or minimization (`Min`) problem. Some examples:

```
@objective(model, Max, sum( c[i]*z[i] for i in 1:3 ) )    # maximization
@objective(model, Min, 2*y["white"] + 3*y["blue"] )       # minimization

@objective(model, Min, sum( A[i,j]*X[i,j] for i in 1:3, j in 1:4 ) )
```

A non-linear objective function is defined using the command `@NLobjective`.

```
@NLobjective(model, Max, sum( x[i]^2  for i in 1:n )      # non-linear objective
```

Constraints

Linear constraints are defined using the command `@constraint`. Here are some examples:

```
@constraint(model, con1, x[1] + x[2] + 2*x[3]  <=  1 )    # inequality constraint

@constraint(model, con2, x[3] - x[4] + 3*x[5]  >=  1 )    # inequality constraint

@constraint(model, con3, x[5] + x[6]           ==  1 )    # equality constraint
```

A non-linear constraint is defined using the command `@NLconstraint`. Here is an example:

```
@NLconstraint(model, con4, 2*x[1]^2 + x[2]^2   <=  4 )    # non-linear constraint
```

For all i $(\forall\ i)$

In many situations there are constraints for each type of product to be manufactured, or for each time step, or for each element in a set. To avoid writing almost identical constraints many times, the following syntax can be used:

```
# The sum of the variable values on each row in matrix X is limited to at most 3
@constraint( model, MaxRow[i in 1:3],
   sum( X[i,j]  for j in 1:4 )  <=  3 )


# At most 2 animals of each color.
@constraint( model, Number[j in COLOR],
   sum( Y[i,j]  for i in ANIMAL )  <=  2 )


# For i in the set N and j in the set N, if the value of the parameter b[i,j]
# is 1, then there is a constraint that x[i,j] >= 5
@constraint( coverage[i in N, j in N ; b[i,j] == 1],  x[i,j]  >=  5 )


# Inventory balance constraint, for each time step t = 1,...,10
# Note that variable "Inventory" needs to be defined for t=0
@constraint( model, InventoryBalance[t in 1:10],
   Inventory[t-1] + Manufactured[t] - Demand[t]  ==  Inventory[t] )
```

Useful commands

There are many useful commands that can be used for the model object. Assume our model object is named "`model`", and it has a variable named "`x`". Here are some examples:

```
print( model )                          # display the model in the terminal
optimize!( model )                      # solve the optimization model

objective_value( model )                # display the objective value
value.( x )                             # display the value of x
```

Indexing done in different ways

In different situations, it is convenient to define parameters and variabels in different ways. Syntax for accessing their values differ somewhat, so be observant!

Vector / Matrix

```
v = [ 70, 60, 50 ]                    # 3-element vector

m = [ 70  60  50 ]                    # 1x3 matrix

A = [ 12  14   7  11 ;                # 3x4 matrix
      16   9  10  12 ;
      10  21  12  19 ]

# Access values 70, 60 and 19
v[1] ,  m[2] ,  A[3,4]
```

Vector of vectors

```
# Create a vector consisting of vectors
LIST  = [ ["A","B","C","D","E"], ["Hello","Bye"] ]

# Access values "D" and "hello"
LIST[1][4] ,  LIST[2][1]
```

Tuples

```
# Create a vector of tuples
COMB = [ ("A",1), ("A",2), ("A",4), ("A",7),
         ("B",3), ("B",5),
         ("C",1), ("C",4), ("C",5)
       ]

# Create a binary variable for each tuple
@variable(model, v[COMB], Bin)

# Address variable with index ("B",5)
julia> v[("B",5)]
```

Dictionary

```
# Create a dictionary  ( key, value )
Abbreviations = Dict([  ("ca.",  "circa (approximately)"),
                        ("cf.",  "confer (compare to)"),
                        ("e.g.", "exempli gratia (for example)"),
                        ("etc.", "et cetera (and other things)"),
                        ("i.e.", "id est (that is)")
                     ])

# Get the value for key "e.g."
julia> Abbreviations["e.g."]
"exempli gratia (for example)"
```

Example 1

This first example demonstrates how Julia/JuMP easily models a small problem. Consider the following linear programming problem with four variables och three constraints.

$$
\begin{array}{lrcrcrcrcr}
\min & z = & 8 \cdot x_1 & + & 14 \cdot x_2 & + & 12 \cdot x_3 & + & 20 \cdot x_4 & \\
 & & & & & & & & & \\
\text{s.t.} & & x_1 & + & 3 \cdot x_2 & + & 2 \cdot x_3 & + & 4 \cdot x_4 & \geq & 122 \\
 & & 7 \cdot x_1 & + & 6 \cdot x_2 & + & 6 \cdot x_3 & + & 2 \cdot x_4 & \leq & 200 \\
 & & 3 \cdot x_1 & + & x_2 & + & 2 \cdot x_3 & + & x_4 & \leq & 50 \\
 & & x_1 & , & x_2 & , & x_3 & , & x_4 & \geq & 0
\end{array}
$$

In Julia/JuMP, the corresponding model looks like this:

```
#------------------------------------------------------------------------------
# Example 1
#------------------------------------------------------------------------------

# Load packages
using JuMP, HiGHS

# Create a model object LP, and use HiGHS as solver
LP = Model(HiGHS.Optimizer)

# Define variables
@variable( LP, x1 >= 0)
@variable( LP, x2 >= 0)
@variable( LP, x3 >= 0)
@variable( LP, x4 >= 0)

# Define the objective function (minimization problem)
@objective( LP, Min, 8*x1 + 14*x2 + 12*x3 + 20*x4 )

# Define the constraints
@constraint( LP, con1,    x1 + 3*x2 + 2*x3 + 4*x4  >=  122 )
@constraint( LP, con2,  7*x1 + 6*x2 + 6*x3 + 2*x4  <=  200 )
@constraint( LP, con3,  3*x1 +   x2 + 2*x3 +   x4  <=   50 )


# Display defined problem
println("Model for Example 1:")
print(LP)

# Solve the optimization problem
solution = optimize!(LP)

# Print solution
println("Optimal Solution x*:")
println("x1 = $(value.(x1))")
println("x2 = $(value.(x2))")
println("x3 = $(value.(x3))")
println("x4 = $(value.(x4))")

println("Optimal objective value:\n $(objective_value(LP))")
```

Example 2

The problem in Example 1 can be written in a more general (compact) form

$$\begin{aligned} \min \quad z = & \sum_{j=1}^{n} c_j \cdot x_j \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} \cdot x_j \leq b_i \qquad i = 1, \ldots, m \\ & x_j \geq 0 \qquad j = 1, \ldots, n \end{aligned}$$

where the values of the parameters a_{ij}, b_i and c_j are defined according to the example. Note that it is neccessary to negate all parameter values for the first constraint, since the inequality ($\leq$) must be the same for all constraints.

```
#------------------------------------------------------------------------------
# Example 2
#------------------------------------------------------------------------------

# Load packages
using JuMP, HiGHS

# Parameters
c = [  8  14  12  20 ]

a = [ -1  -3  -2  -4 ;
       7   6   6   2 ;
       3   1   2   1 ]

b = [ -122 200 50 ]

m = length(b)     # number of constraints
n = length(c)     # number of variables


# Create a model object LP, and use HiGHS as solver
LP = Model(HiGHS.Optimizer)

# Define variables
@variable( LP, x[1:n] >= 0 )

# Define the objective function (minimization problem)
@objective( LP, Min, sum( c[j]*x[j] for j in 1:n ) )

# Define constraints (as <= constraints)
@constraint( LP, con[i in 1:m], sum( a[i,j]*x[j] for j in 1:n ) <= b[i] )


# Display defined problem
println("Model for Example 2:")
print(LP)

# Solve the optimization problem
solution = optimize!(LP)

# Print solution
println("Optimal solution:\n ", value.(x) )
println("Optimal objective value:\n $(objective_value(LP))")
```

Example 3

Another example, here with a double summation and variables with multiple indices.

$$
\begin{aligned}
\min \quad z = \quad & \sum_{k \in K} c_k \cdot y_k \\
\text{s.t.} \quad & \sum_{i \in I} \sum_{j \in J} x_{ijk} = L && k \in K \\
& x_{ijk} \leq M y_k && i \in I,\ j \in J,\ k \in K \\
& x_{ijk} \in \mathbb{Z}_+ && i \in I,\ j \in J,\ k \in K \\
& y_k \in \{0,1\} && k \in K
\end{aligned}
$$

```
#------------------------------------------------------------------------------
# Example 3
#------------------------------------------------------------------------------

# Load packages
using JuMP, HiGHS

# Parameters
c = [ 8  14  12 ]
I = [ 1 2 3 4 5 ]
J = [ 1 2 3 4 5 6 7 ]
K = [ 1 2 3 ]
L = 100
M = 20

# Create a model object MIP, and use HiGHS as solver
MIP = Model(HiGHS.Optimizer)

# Define variables
@variable( MIP, x[I,J,K] >= 0, Int )
@variable( MIP, y[K], Bin )

# Define the objective function (minimization problem)
@objective( MIP, Min, sum( c[k]*y[k] for k in K ) )

# Constraints
@constraint( MIP, con1[k in K], sum( x[i,j,k] for i in I, j in J )  ==  L )
@constraint( MIP, con2[i in I, j in J, k in K], x[i,j,k]  <=  M*y[k] )


# Display defined problem
println("Model for Example 3:")
print(MIP)

# Solve the optimization problem
solution = optimize!(MIP)

# Print solution
println("Optimal solution:")
println( "x* =\n" , value.(x) )
println( "y* =\n" , value.(y) )

println("Optimal objective value:\n $(objective_value(MIP))")
```

Example 4

A non-linear problem

$$
\begin{aligned}
\max \quad f &= \sum_{i=1}^{m} p_i x_i \\
\text{s.t.} \quad \sum_{i=1}^{m} a_i x_i^2 &\leq K \\
0 \leq x_i &\leq 500 - 0.1 \cdot p_i \qquad i = 1, \ldots, m \\
100 \leq p_i &\leq 900 \qquad\qquad\quad i = 1, \ldots, m
\end{aligned}
$$

```
#------------------------------------------------------------------------
# Example 4
#------------------------------------------------------------------------

# Load packages
using JuMP, Ipopt

# Parameters
a = [4 1 2 2 1 4 2 3 2]
K = 2000

m = length(a)


# Create a model object NLP (Non-Linear Problem), and use Ipopt as solver
NLP = Model(Ipopt.Optimizer)

# Define variables
@variable( NLP, 100 <= p[1:m] <= 900 )      # p - sales price for product i
@variable( NLP, x[1:m] >= 0 )               # x - production of product i

# Non-Linear objective  (maximization problem)
@NLobjective( NLP, Max, sum( p[i] * x[i]  for i in 1:m ) )

# Non-Linear constraint
@NLconstraint( NLP, con_capacity, sum( a[i]*x[i]^2  for i = 1:m ) <= K )

# Linear constraint
@constraint( NLP, con_limit[i in 1:m], x[i] <= 100 - 0.1*p[i] )


# Display defined problem
println("Model for Example 4:")
print(NLP)

# Solve the optimization problem
solution = optimize!(NLP)

# Print solution
println("Optimal solution:")
println( "p* =\n" , value.(p) )
println( "x* =\n" , value.(x) )

println("Optimal objective value:\n $(objective_value(NLP))")
```