Optimization for Learning - FRTN50
Introduction to Julia

Written by: Mattias Fält

Latest update: November 4, 2019

Introduction

This document is intended to give a brief introduction to Julia for students taking the FRTN50 - "Optimization for Learning" course at the Department of Automatic Control, Lund.

The goal with the first section is to motivate why we choose to use Julia as the programming language in this course. Hopefully it will provide you with enough reasons as to why it could be valuable to learn a bit of Julia. It takes a lot of practice to become a good programmer in any language, which is neither the goal nor the expectation in this course. Most of the exercises can be solved with concepts that are common to most programming languages, however, through the hand-ins, you will have to get familiar with the syntax and some of the basic concepts in Julia.

The rest of the document will introduce some of the resources and references that will be useful throughout this course.

Why Julia

As a student at LTH, you are probably used to working in `MATLAB` and `Java` and you may even have tried a bit of `Python`. `Julia` (`https://julialang.org/`) is a new language that most students have not encountered so it is reasonable to ask why you should use it in this course. There are surely some disadvantages to using `Julia`, but when it comes to writing optimization algorithms there are many more advantages.

Julia

- **is easy to use:** It is a high level language combining ideas from `Ruby`, `Python`, `Lisp`, `Haskel`, with a syntax that is simple to get started with, similar to `MATLAB`.
- **is very fast**: The just in time compilation allows speeds that are very similar to a `C` implementation, without the complexity. High performance libraries in other languages, like Tensorflow for `Python`, are written in `C` or `C++` to be able to be fast. This is not necessary in `Julia`, high performance libraries can be written purely in `Julia`.

- **is made for scientific computing:** Just like `MATLAB`, `Julia` is deigned to handle mathematical operations, matrices and similar concepts in a convenient way.
- **is open source:** The whole language is open for anyone to see and contribute to (and written mostly in `Julia`), so it is possible to understand what is going on behind the scene. And since it's free you don't need a license to use it once you leave LTH. For comparison commercial `MATLAB` prices are roughly: 8000kr/year + Optimization Toolbox 4000kr/year + Parallel Computing Toolbox 4000kr/year + Simulink 12600kr/year ....
- **has great built in functionality:** Matrix computations, Statistics, Parallel computing and similar functionality is built into standard libraries and shipped with `Julia`.
- **has lots of free packages:** Just like many other programming languages, Julia has a *good package manager* with lots of packages (2000+) for most applications. And they are all free and open source.
- **is extremely powerful:** By relying on *dynamic dispatch*, a concept where *the argument types of a function call decides which implementation is run*, it is possible to write high level code that is efficient for many types, for example both for sparse and dense matrices, complex and integer numbers and so on.

These are some of the main reasons why Julia is a fast growing language, and why it is increasingly becoming the language used for researchers in optimization. It en enables us to test out powerful algorithms quickly in a high level language without having to re-implement them in `C` or `C++` to make them fast.

How to get started with Julia

Julia comes with a simple but powerful REPL (Read Evaluate Print Loop) which makes it is easy to run code from a terminal window. But for handling anything more than a few lines of code, it is useful to have an IDE (Integrated Development Environment), similar to the interface of `MATLAB`, or the Eclipse editor which is often used with `Java`. The standard choice for `Julia`, which is also the one we will be able to help you with, is the Juno version of the Atom editor. Installation instructions for Julia (we recommend version 1.2 or newer) and the editor can be found here: `http://docs.junolab.org/latest/man/installation/`

In the next section we will introduce some simple concepts and tools, you can go to `http://docs.junolab.org/latest/` for more detailed documentation on Juno, and the full documentation for the Julia language can be found at `https://docs.julialang.org/en/v1/`.

Basic Julia in Juno

At the bottom of the editor you should have a console called the REPL. You can write and execute simple commands in this box and quickly see the results. The behavior and syntax is similar to that on MATLAB, at least on the surface. Evaluate some simple mathematical expressions to make sure everything is running as expected. On the first run, Julia will probably take some time installing some pre-installed packages.

Now create a new file in Juno/Atom with the file extension `.jl`, this will enable syntax highlighting for Julia. Write some simple lines like

```julia
a = 1
b = 2
a+b
```

You can evaluate a line at a time using `Shift+Enter`, or the whole file with `Control+Shift+Enter`. All the variables that are defined here will also be shared with the REPL below.

There are two equivalent ways to define a function:

```julia
# Comments are made using the `#` symbol
function f1(x)
  x^2 + x
end

# Another way is the short syntax:
f2(x) = sin(x) + x

""" And the documentation for f2(x) = sin(x) + x
can be written like this
"""
function f3(x)
  a = x^2
  return sqrt(a)
end

# You can now call the functions:
six = f1(2)
```

To find documentation for a function, press `?` in the REPL. This will activate `help?` mode. Try for example `?sin` to see how it works.

Note that we did not include any `return` statement in the two first definitions, by default a function in Julia will return the result of the last line, so this is often omitted.

Run the code above to make sure it works properly. Now try to make a change in the function `f1` and re-evaluate `f1(2)`. Did the value change? It is important to remember to first reevaluate the function definition before calling it, or the old version will still be called.

It is also possible to use special characters in your code. These can be entered in Juno using standard LaTeX commands, such as `\alpha` and `\Delta` for α and Δ:

```julia
x = 1.0
Δx = 0.1
α = x + Δx
```

Standard libraries

Simple operations and function definitions work without using any special libraries:

```julia
# Create a (column) vector:
v = [1,2]
# A matrix:
A = [1 2; 3 4]
# Or a 1x2 row matrix
b = [5 6]
# Multiply
b*A*v
```

However, if we want access to more specific methods we have to use some of the available packages. Julia comes with a set of standard libraries, like `LinearAlgebra` and `Statistics` that define functions for most of the basic needs. To get access to those functions we have to call

```julia
using LinearAlgebra, Statistics
# We can now calculate determinants:
det(A)
# And mean and standard deviations for example:
m, s = mean(v), std(v)
```

But anyone can create a Julia package, and they are not all included in the standard download. To access those packages, we first have to download them. Thankfully, Julia has a very nice package manager that can be accessed using the `]` symbol in the REPL. One of the packages that we will be using to display graphs is the `Plots.jl` package. To add it, just open the package manager with `]` and write `add Plots`. This will download the package and install all necessary dependencies.

We can now try to use it

```julia
using Plots
v = randn(10)
plot(v)
# Create a new plot
plot(v .+ 1, color=:green)
# Add new plot to existing window
plot!(v .- 1, linewidth=3)
```

This should open a plot window in Atom. There are many options for how plots should be displayed with `Plots.jl`, so called back-ends, but for simplicity we will stick with the default one.

The first time you write `using` on a package, it may take some time to precompile, especially for a big library like `Plots.jl`. The same is true for the first call to a function, however, as you will see, the second call to `plot` will be much faster.

You might be asked to supply figures in your hand-ins. A convenient way to export a plot is the `savefig` command:

```
# Save current figure as a png
savefig("myplot.png")
savefig("myplot.pdf")
# the figures are saved in the current directory be default
# print-working-directory:
pwd()
```

Advanced concepts

There is always a lot to learn when starting with a new language. We have tried to collect some of the useful features below that could serve as a reference when trying to solve and understand the hand-ins. There are a three main concepts just in the simple line `plot!(v .- 1, linewidth=3)`, that could be useful to understand.

Broadcasting (element-wise operations)

The first is that we write `v .+ 1` instead of `v + 1`. The reason is that addition (`+`) between a vector and a number is not defined. The `.+` instead represents element-wise addition, similar to how it is written in MATLAB. However, this behavior (called `broadcast`) is not limited to simple operations like `+`,`-`,`*`, it can be applied to any function, even the ones you define yourself!

```julia
function g1(x, y)
    x^2*y + 1
end

a = [1,2,3,4,5]
b = [6,7,8,9,10]
# Apply the function g1 to each of the elements in a and b
g1.(a, b)
```

Optional and keyword arguments

The plot commands above were called with *keyword-arguments* `linewidth=3` and `color=:green`. These are extra arguments that are referenced to by name and are defined after a `;` in the definition of a function:

```julia
function g2(x, y; square=false, extra = 0)
  a = x + y + extra
  if square
    return a^2
  else
    return a
end

# g2 can be called with no keyword arguments
a1 = g2(1,2)
# Which is the same as with the default arguments:
a2 = g2(1,2, square=false, extra=0)
# But can also be called with any of the keyword arguments set:
a3 = g2(1,2, extra=5)
```

It is also possible to add optional arguments with default values after normal arguments:

```julia
function loop(val, n=10, k=5; start = 1)
  s = 0.0
  for i = start:n
    for j = start:k
      s += val
    end
  end
  return s
end

# The second argument is optional, all of the following are valid calls:
loop(2.0)
# Set n=12
loop(2.0, 12)
# Set n=12, k=4
loop(2.0, 12, 4)
# Set everything
loop(2.0, 12, 4, start=2)
# Set only n=12, and start=2
loop(2.0, 12, start=2)
# Only start=12
loop(2.0, start=2)
```

Even functions can be sent as inputs to other functions, this is used in cases like this:

```julia
v = [1,2,-3,4]
s1 = sum(v) # Should be 4
# Sum the absolute value of each element
s2 = sum(abs, v) # Should be 10
```

References and in-place operations

The last note is regarding how Julia treats most types, such as vectors and matrices, as references when a function is called. This makes it possible for a function to change the values in the input arguments (MATLAB will create a copy if the input arguments are modified). When a function changes existing variables, it is customary to add an `!` to the name of the function. This explains why we write `plot!` when updating an existing plot. Some common examples:

```julia
function setfirstzero!(A)
  # Set the element in first row and column to 0
  A[1,1] = 0
end

A = randn(4,4)
# The first element in A will now be set to 0
setfirstzero!(A)

function allzero!(A)
  # Element-wise - set all elements in A to zero
  # Same as A[:] .= 0
  A .= 0
end

allzero!(A)
sum(A) # This is 0

# Note that the functions above change the elements inside of A, the following
↪  does not!
function notworking!(A)
  # Create a NEW variable, also called A, and set it to 4 by 4 zeros
  A = zeros(4,4)
  # We are no longer referring to the same same matrix
  return A
end

A = randn(4,4)
B = notworking!(A)
sum(A) # This is not zero
sum(B) # This is
```

In-place operations can be very useful to avoid forcing the computer to find new space for variables (allocating memory):

```julia
y = randn(10000)
a = randn(10000)
b = randn(10000)

""" calculate a^2 + b^2 element-wise and return vector with result """
function squareadd(a, b)
  # Create a vector of same length and type as a to store the result
  y = similar(a)
  for i in eachindex(a) # Same as 1:length(a)
    y[i] = a[i]^2 + b[i]^2
  end
  return y
end

""" calculate a^2 + b^2 element-wise and store in y """
function squareadd!(y, a, b)
  for i in eachindex(a)
    y[i] = a[i]^2 + b[i]^2
  end
  return
end

y = squareadd(a,b)
# @time will measure time and allocations
@time squareadd(a,b)

# Store result directly in y
squareadd!(y, a, b)
# The first run will take longer since it is compiling the function,
# so we run it once before computing the time
@time squareadd!(y, a, b)
```

As you see it can be considerably faster to reuse the space that is already allocated. This is always considered when trying to write optimized algorithms, and will see it in some of the exercises in the course. One common function is `mul!`:

```julia
# Calculate A*b and store it in y, without allocating any extra memory
mul!(y, A, b)
```

Types

Julia is not an object-oriented language, but types can be created to generate a similar behavior. For example, to represent a point in two dimensions we might want to create something like this:

```julia
# Create a structure (type), x and y will be stored as Float64
struct Point2D
  x::Float64
  y::Float64
end

# We can now create points
p1 = Point2D(1, 2)
p2 = Point2D(4.0, 6.0)

# and access the fields
p1x = p1.x # 1.0

# and create functions
distance(a::Point2D, b::Point2D) = sqrt((a.x - b.x)^2 + (a.y - b.y)^2)

distance(p1,p2) # should be 5.0
```

The notation `a::Point2D` means that the function will only accept calls where `a` is a `Point2D`.

If we want to be able to store points of other precisions, for example Integers, we could instead create a *parametric* type:

```julia
struct MyPoint2D{T}
  x::T
  y::T
end

# We can now create points, of any type T we want
p1 = MyPoint2D{Int64}(1, 2)
p2 = MyPoint2D{Int64}(4, 6)

# Allow only calls where a and b has a common T
function distance(a::MyPoint2D{T}, b::MyPoint2D{T}) where T
  sqrt((a.x - b.x)^2 + (a.y - b.y)^2)
end

distance(p1,p2) # should be 5.0
```

Although you will probably not have to create these kinds of types yourself, it helps to explain the why an expressions like `[1 2 3; 4 5 6]` is printed with the type `Array{Int64,2}`. All vectors and matrices are implemented in the type `Array{T,N}`, where `T` is the element type and `N` is the number of dimensions, 1 for vectors, 2 for Matrices.

Documentation

You should now have seen most of the important features in Julia that you be could use when solving the hand-ins. For more detailed explanations, see the Julia documentation `https://docs.julialang.org/`, or ask the assistants during the exercises!