The Julia programming language and some applications

1. A brief overview of Julia

Introduction

Julia is a fairly recent programming language, started in 2009. It's part of crowded field that includes

- Matlab
- R
- Python
- Stata
- C
- Fortran

and many others.

Motivation

I am not an advocate of any particular language, but rather believe in choosing the right tool for the job. I find Julia particularly good for scientific computing, optimization, parallel computing. Some of its avantages are

- Free and open source
- Fast
- Growing library of packages for e.g. optimization, data visualization, machine learning, parallel computing, etc.
- Choice of good integrated development environments (IDEs)
- Creating packages and Jupyter notebooks

Sometimes, learning about a new language, the best practices for that language and some of the packages available in that language can can point us in the right direction to solve a particular problem.

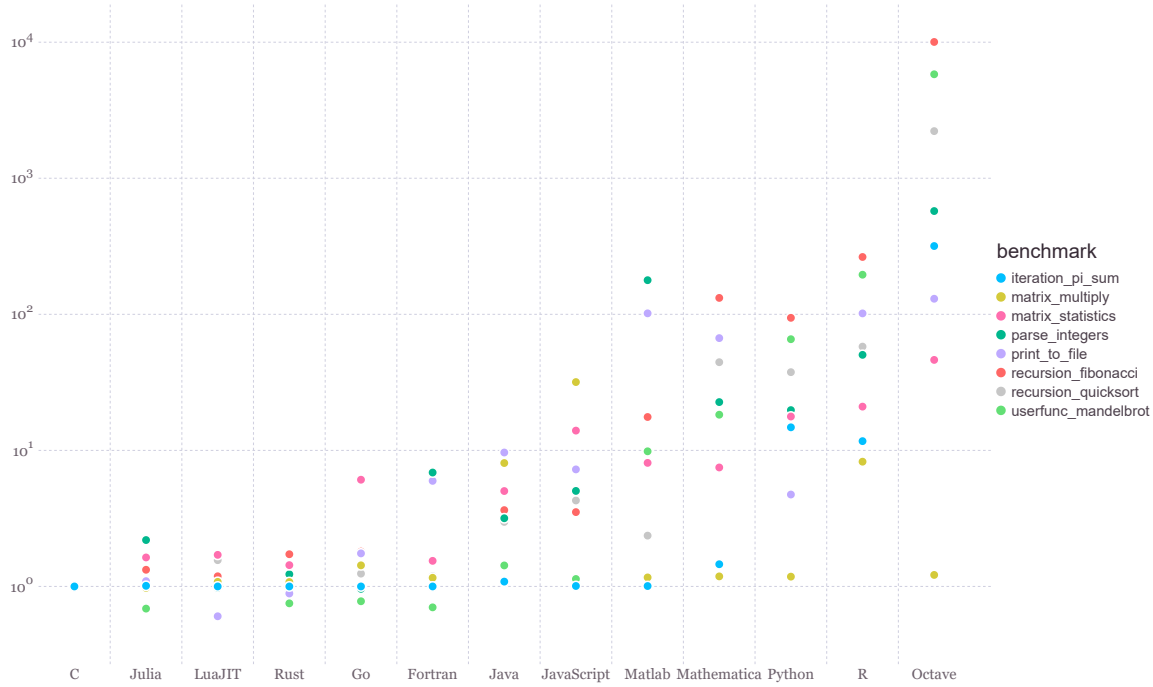
The Julia ecosystem

Here are some examples of well-established Julia packages:

- Linear Algebra: `LinearAlgebra.jl`, `SparseArrays.jl`
- Data and data visualization: `DataFrames.jl`, `CSV.jl`, `ReadStatTables.jl`, `Plots.jl`, `PyPlot.jl`
- Optimization: `JuMP.jl`, `Optim.jl`
- Differentiation: `ForwardDiff.jl`, `FiniteDifferences.jl`
- Econometrics: `GLM.jl`
- Parallel computing: `Distributed.jl`

- Machine learning: `MLJ.jl`

Performance



(source: https://julialang.org/benchmarks/)

Lingering issues

- Still a new language so regular updates mean updating code regularly
- Smaller library of well established packages
- Some things that "just work" in other languages won't in Julia
- Documentation is sometimes lacking

2. Three applications

- Web scraping
- Parallel computing
- Optimization in JuMP

Web scraping

According to Śpiewanowski et al. 2022, "web scraping refers to the process of collecting data from web pages either manually or using automation tools or specialized software."

Web scraping has been used in economics for collecting data on (see Śpiewanowski et al. 2022 and references therein)

- job vacancies (Khun and Shen 2012, Adams et al. 2020 using Burning Glass Technologies data)
- tribunal decisions (Adams et al. 2021)

- prices (Yilmaz et al. 2022, Cavallo 2018)
- etc.

Before doing anything, we should check that we can scrape data from the website we are interested in:

- check the terms and conditions!
- check the `robots.txt` file on the website.

Sometimes, website will provide users access to an Application Programming Interface (API), which facilitate access to the information. Conversely, there are many websites for which the simple methods introduced below won't work because:

- a user account is needed to access the website
- there are restrictions on the type of users who can access the website (Tho kindly provided this example with age restrictions - thanks!)
- some websites will quickly kick us out after a few requests. Possible solutions include using packages to change our IP address, including a random waiting time before the next request, etc.

Now, on to the actual scrapping. For the purpose of this application, I would like to collect milk prices on Morrisons. First, let's start by scraping some information from the results page after search for the keyword semi skimmed milk pint''.

In [ ]:
```
using HTTP, Cascadia, Gumbo
using DataFrames
keyword = "semi-skimmed-milk-pint"

# Base URL
url_base       = "https://groceries.morrisons.com/search?entry="
results_url    = string(url_base, keyword)              # Construct the URL for the
results_html   = HTTP.get(results_url)                  # Get the HTML for the searc
results_parsed = parsehtml(String(results_html.body))   # Parse the HTML for the sea
```

In [ ]:
```
# Method 1: regular expressions.
results_string = string(results_parsed)
d = DataFrame(url = [])
f = findall(r"class=\"fop-contentWrapper\" role=\"presentation\"><a href=\"(.*?)\">"
for k=1:length(f)
    m = match(r"class=\"fop-contentWrapper\" role=\"presentation\"><a href=\"(.*?)\"
    push!(d, Dict(:url => string("https://groceries.morrisons.com",m.captures[1])))
end
d
```

In [ ]:
```
# Method 2: selectors.
d = DataFrame(url = [])
results_selector = eachmatch(Selector("#main-content > div:nth-child(2) > div.main-c

for child in results_selector[1].children
    push!(d, Dict(:url => string("https://groceries.morrisons.com",child[2][1][1].at
end
d
```

Now that we have a list of urls of products, we can load the page of each product and extract the information we need. Let's use Method 2 to do that.

In [ ]:
```
d.price = Vector{Union{Missing, Float64}}(missing, length(d.url))
for k = 1:3                                                    # Scrape data for first
    sleep(rand(1)[1])                                          # Random waiting time
    product_html    = HTTP.get(d.url[k])                       # Get the HTML for the p
    product_parsed = parsehtml(String(product_html.body))      # Parse the HTML for the
    product_selector = eachmatch(Selector("#overview > section.bop-section.bop-basic

    d.price[k] = parse(Float64,string(product_selector[1].children[1].attributes["co
end
d
```

Parallel computing

- Bootstrapping. Suppose we want to obtain bootstrapped standard errors and let K be the number of bootstraps. If it takes time to obtain the estimates for one sample, then bootstrapping will be very computationally intensive. We can use parallel computing to alleviate this problem.

In [ ]:
```
using Distributed      # a Julia package to do parallel computing
addprocs()             # adding workers
workers()              # checking the number of workers
```

In [ ]:
```
@everywhere function bootstrap(k)    # an example bootstrap function, that we want t
    sleep(0.1)
    return "Some estimates"
end
```

If $K = 100$, it would take roughly 10 seconds to obtain the bootstrapped standard errors:

In [ ]:
```
K  = 100
x0 = rand(K)

start = time()
for k=1:K
bootstrap(k)
end
println(string("Time elapsed: ", time() - start))
```

Let us use a parallel version of the code above:

In [ ]:
```
println(string("Number of workers: ", nworkers()))
start = time()
Distributed.pmap(k -> bootstrap(k), collect(1:K))
println(string("Time elapsed: ", time() - start))
```

- Numerical differentiation. Suppose we want to compute (numerically) the derivative the gradient of some function

$$f : \mathbb{R}^m \to \mathbb{R}$$

Using a simple forward method, we would need to evaluate the function $m+1$ times, which might be computationally intensive. Once again, we can use parallel computing to alleviate this problem.

In [ ]:
```
@everywhere begin
    using NonLinearProg
    function fun(x)
        sleep(0.1)
        return sum(x.*x)
    end
    function insert(x,xk,k)
        new_x = copy(x)
        new_x[k] = xk
        return new_x
    end
end
```

In [ ]:
```
m  = 100
x0 = rand(m)

# Without parallel computing
start = time()
NonLinearProg.derivative(fun,x0)
println(string("Time elapsed: ", time() - start))

# With parallel computing
println(string("Number of workers: ", nworkers()))
start = time()
g = Distributed.pmap(k -> NonLinearProg.derivative(xk -> fun(insert(x0,xk,k)), x0[k]
println(string("Time elapsed: ", time() - start))
```

Optimization in JuMP

Suppose we want to solve the following utility maximization problem

$$\max_{x_1 \geq 0, x_2 \geq 0} x_1^{\alpha} x_2^{1-\alpha}$$

subject to

$$p_1 x_1 + p_2 x_2 \leq m$$

We solve this problem in JuMP as follows:

In [ ]:
```
using JuMP     # a Julia optimization interface package
using Ipopt    # a well known solver
m  = 5.0
p1 = 1.0
p2 = 2.0
alpha = 0.5
model = Model(Ipopt.Optimizer)                          # initialize the model, usi
@variable(model, x1 >= 0)                               # add x1 variable (and boun
@variable(model, x2 >= 0)                               # add x2 variable (and boun
@constraint(model, p1*x1 + p2*x2 <= m)                  # add constraint
@NLobjective(model, Max, (x1^alpha) * (x2^(1-alpha)))   # objective function
optimize!(model)                                        # solve problem
println([value(x1); value(x2)])                         # print optimal values
```

We can easily adapt this code to solve the utility maximization problem for many consumers at a time. For example:

In [ ]:

```
N = 1000
m  = 5 .+ rand(N)
p1 = 1.0 .+ rand(N)
p2 = 2.0 .+ rand(N)
alpha = rand(N)
model = Model(Ipopt.Optimizer)
@variable(model, x1[i=1:N] >= 0)
@variable(model, x2[i=1:N] >= 0)
@constraint(model, [i=1:N], p1[i]*x1[i] + p2[i]*x2[i] <= m[i])
@NLobjective(model, Max, sum((x1[i]^alpha[i]) * (x2[i]^(1-alpha[i])) for i=1:N))
optimize!(model)
println(value.(x1)[1:10])
```