Julia, l’une unique solution d’un problème d’optimisation.
ou de la difficulté d’estimer des convolutions de lois gammas.

O. Laverny $^{1,2}$
June 22, 2021

1 Institut Camille Jordan, UMR 5208, Université Claude Bernard Lyon 1, Lyon, France
2 SCOR SE
1. Un problème d’optimisation mathématiquement 'simple'

2. La notion de nombre de Julia.

3. Multiple dispatch and type-agnostic code

4. Conclusion
Un problème d’optimisation mathématiquement ’simple’
Recall: A random vector $X \in \mathbb{R}^d_+$ can be characterized through its cumulant generating function:

$$K(t) = \ln \mathbb{E}\left(e^{\langle t, X \rangle}\right)$$

Definition (Multivariate Gamma Convolutions, see Bondesson\(^1\))

$$X \sim G_{d,n}(\alpha, s) \iff K(t) = \sum_{i=1}^{n} -\alpha_i \ln (1 - \langle s_i, t \rangle)$$

Goal: Estimation from multivariate dataset, $d \leq \approx 4$ and $n \leq \approx 100$

Problem: No estimation procedures are available in the literature. Indeed, this is a hard inverse problem: to estimate some distribution in $G_{d,n}$ for the random vector $X$, we seek parameters of distributions of independent gamma random variables $Y_1, \ldots, Y_n$ such that $X = s' Y$, that is:

$$X_1 = s_{1,1} Y_1 + \ldots + s_{1,n} Y_n$$

$$\ldots$$

$$X_d = s_{d,1} Y_1 + \ldots + s_{d,n} Y_n$$

The orthonormal Laguerre basis of $L_2(\mathbb{R}^d_+)$

**Definition (Tensorized Laguerre basis, see Comte\(^2\), Mabon\(^3\) and Dussap\(^4\))**

For all $\mathbf{p} \in \mathbb{N}^d$, $\varphi_{\mathbf{p}}(\mathbf{x}) = \prod_{i=1}^d \varphi_{p_i}(x_i)$ where $\varphi_p(x) = \sqrt{2} \sum_{k=0}^p \binom{p}{k} \frac{(-2x)^k}{k!} e^{-x}$.

These functions form an orthonormal basis of $L_2(\mathbb{R}^d_+)$.

Therefore, every density $f$ that is square-integrable can be expended as:

$$f(\mathbf{x}) = \sum_{\mathbf{p} \in \mathbb{N}^d} a_{\mathbf{p}} \varphi_{\mathbf{p}}(\mathbf{x})$$

where $a_{\mathbf{p}} = \int \varphi_{\mathbf{p}}(\mathbf{x}) f(\mathbf{x}) \, d\mathbf{x} = \sqrt{2} \sum_{k \leq p} \binom{p}{k} \frac{(-2)^k}{k!} \mathbb{E} \left( X^k e^{-1,X} \right)$

**Final loss:** $L(\alpha, \mathbf{s}) = \|f - \hat{f}\|_2^2 = \sum_{k \leq m} (\hat{a}_k - a_k(\alpha, \mathbf{s}))^2$


If our observations are exact, from a density in the class, this is equivalent to a moment problem on the Thorin measure $\nu$ with atoms $s$ and weights $\alpha$. But:

(i) Denote $\mu_k = \mathbb{E}(X^ke^{\langle -1, X \rangle})$, the $k$th derivative of $M(t) = \exp \circ K(t)$ taken at $t = -1$.

(ii) Then (Faa di bruno) there is a bijection between derivatives of $K$ and Laguerre coefficients. The derivatives of $K(t) = \int \ln(1 - \langle s, t \rangle) \nu(s)$ are given as a linear combination of moments of the Thorin measure.

(iii) Unfortunately observations are usually outside the multivariate cone of moments, and Lassere hierarchies are not tractable.

The L2 Laguerre loss is more stable.
An algorithm with a lot of flaws

**Algorithm:** Laguerre coefficients \((a_k(\alpha, s))_{k \leq m}\) of \(G_{d,n}(\alpha, s)\) random vectors.

**Input:** Shapes \(\alpha \in \mathbb{R}^d_+\), scales \(s \in \mathcal{M}_{n,d}(\mathbb{R}^+)_+\), and truncation threshold \(m \in \mathbb{N}^d\).

**Result:** Laguerre coefficients \((a_k)_{k \leq m}\) of the \(G_{d,n}(\alpha, s)\) density

Compute the simplex version of the scales \(x_i = \frac{s_i}{1+|s_i|}\) for all \(i \in 1,...,n\).

Let \(\kappa_0 = -\sum_{i=1}^{n} \alpha_i \ln (1 - |x_i|)\) and \(a_0 = \mu_0 = \exp(\kappa_0)\).

foreach \(0 \neq k \leq m\) do
  Let \(a_k = \mu_k = 0\), \(j\) be the index of the first \(k_i\) that is non-zero, \(p = k\) and set \(p_j = p_j - 1\).
  Let \(\kappa_k = (|k| - 1)! \sum_{i=1}^{n} \alpha_i x_i^k\)
  foreach \(l \leq p\) do
    Set \(\mu_k += (\mu_l)(\kappa_{k-l})(\binom{p}{l})\) according to efficient Faà di Bruno’s algorithm from Miatto*
    Set \(a_k += \mu_l(\binom{k}{l})\frac{(-2)^{|l|}}{l!}\)
  end
  Set \(a_k += \mu_k \frac{(-2)^{|k|}}{k!}\)
end

\(a = \sqrt{2^d}\ a\)

Return \(L(\alpha, s) = \|f - \hat{f}\|_2^2 = \sum_{k \leq m} (\hat{a}_k - a_k)^2\)

---

Algorithm: Laguerre coefficients \((a_k(\alpha, s))_{k \leq m}\) of \(G_{d,n}(\alpha, s)\) random vectors.

Input: Shapes \(\alpha \in \mathbb{R}^d_+\), scales \(s \in M_{n,d}(\mathbb{R}^+)_+\), and truncation threshold \(m \in \mathbb{N}^d\)

Result: Laguerre coefficients \((a_k)_{k \leq m}\) of the \(G_{d,n}(\alpha, s)\) density

Compute the simplex version of the scales \(x_i = \frac{s_i}{1 + |s_i|}\) for all \(i \in 1, \ldots, n\).

Let \(\kappa_0 = -\sum_{i=1}^n \alpha_i \ln (1 - |x_i|)\) and \(a_0 = \mu_0 = \exp(\kappa_0)\)

foreach \(0 \neq k \leq m\) do

Let \(a_k = \mu_k = 0\), \(j\) be the index of the first \(k_i\) that is non-zero, \(p = k\) and set \(p_j = p_j - 1\).

Let \(\kappa_k = (|k| - 1)! \sum_{i=1}^n \alpha_i x_i^k\)

foreach \(l \leq p\) do

Set \(\mu_k \leftarrow \mu_k (\kappa_{k-l})^l\)

Set \(a_k \leftarrow a_k \frac{(-2)^l}{l!}\)

end

Set \(a_k \leftarrow \mu_k \frac{(-2)^{|k|}}{|k|!}\)

end

\(a = \sqrt{2^d} a\)

Return \(L(\alpha, s) = \|f - \hat{f}\|_2^2 = \sum_{k \leq m} (\hat{a}_k - a_k)^2\)
Mince, recursif donc non distribuable.

**Algorithm:** Laguerre coefficients \((a_k(\alpha, s))_{k\leq m}\) of \(G_{d,n}(\alpha, s)\) random vectors.

**Input:** Shapes \(\alpha \in \mathbb{R}^d_+\), scales \(s \in \mathcal{M}_{n,d}(\mathbb{R}_+)\), and truncation threshold \(m \in \mathbb{N}^d\)

**Result:** Laguerre coefficients \((a_k)_{k\leq m}\) of the \(G_{d,n}(\alpha, s)\) density

Compute the simplex version of the scales \(x_i = \frac{s_i}{1 + |s_i|}\) for all \(i = 1, ..., n\).

Let \(\kappa_0 = -\sum_{i=1}^{n} \alpha_i \ln (1 - |x_i|)\) and \(a_0 = \mu_0 = \exp(\kappa_0)\)

**foreach** \(0 \neq k \leq m\) **do**

Let \(a_k = \mu_k = 0\), \(j\) be the index of the first \(k_i\) that is non-zero, \(p = k\) and set \(p_j = p_j - 1\).

Let \(\kappa_k = (|k| - 1)! \sum_{i=1}^{n} \alpha_i x_i^k\)

**foreach** \(l \leq p\) **do**

Set \(\mu_k += (\mu_l) (\kappa_{k-l}) (p_l)\)

Set \(a_k += \mu_l (\binom{k}{l} \frac{(-2)^{|l|}}{|l|})\)

end

Set \(a_k += \mu_k \frac{(-2)^{|k|}}{k!}\)

end

\(a = \sqrt{2^d} a\)

Return \(L(\alpha, s) = \|f - \hat{f}\|_2^2 = \sum_{k \leq m} (\hat{a}_k - a_k)^2\)
Algorithm: Laguerre coefficients \((a_k(\alpha, s))_{k \leq m}\) of \(G_{d,n}(\alpha, s)\) random vectors.

**Input:** Shapes \(\alpha \in \mathbb{R}^d_+\), scales \(s \in \mathcal{M}_{n,d}(\mathbb{R}_+)\), and truncation threshold \(m \in \mathbb{N}^d\)

**Result:** Laguerre coefficients \((a_k)_{k \leq m}\) of the \(G_{d,n}(\alpha, s)\) density

Compute the simplex version of the scales \(x_i = \frac{s_i}{1+|s_i|}\) for all \(i \in 1,...,n\).

Let \(\kappa_0 = -\sum_{i=1}^{n} \alpha_i \ln (1 - |x_i|)\) and \(a_0 = \mu_0 = \exp(\kappa_0)\)

```
foreach \(0 \neq k \leq m\) do
    Let \(a_k = \mu_k = 0\), \(j\) be the index of the first \(k\) that is non-zero, \(p = k\) and set \(p_j = p_j - 1\).
    Let \(\kappa_k = (|k| - 1)! \sum_{i=1}^{n} \alpha_i x_i^k\)
    foreach \(l \leq p\) do
        Set \(\mu_k += (\mu_l) (\kappa_{k-l}) (p_l)\) Set \(a_k += \mu_l \binom{k}{l} \frac{(-2)^{|l|}}{l!}\)
    end
    Set \(a_k += \mu_k \frac{(-2)^{|k|}}{k!}\)
end

\(a = \sqrt{2^d} a\)
```

Return \(L(\alpha, s) = \|f - \hat{f}\|_2^2 = \sum_{k \leq m} (\hat{a}_k - a_k)^2\) as dense polynomials into \(\mu_0\) and \(x\) ...

Euh, c’est convexe?
To sum up the problems and solutions we have for the moment, we can say that our loss is:

1. Combinatorial $\implies$ Arbitrary precision
2. Highly recursive $\implies$ Compiled code
3. Highly non-convex $\implies$ Global optimization

(i) Arbitrary precision is available in many languages, usually with a wrapper of “mpfr”
(ii) Global optimization routines are available in many languages.
(iii) Compilation can be done in many languages also, beforehand or “JIT”.

Few languages allows the three concepts together without recoding a whole library. Julia does.
La notion de nombre de Julia.
Julia has no requisite for a type to be usable as a number. A number type is just any type that implements the things you need to do with it. Our algorithm requires only the functions $+,-,\times,\div,\exp$ and $\ln$ to be implemented (integer powers have a type-agnostic method in Base). Arbitrary precision:

(i) BigFloat: “mpfr” interface integrated in Base, but allocate like crazy
(ii) Arb: Not native code, crashes a lot.
(iii) DoubleFloats.jl
(iv) MultiFloats.jl

But more things can be done:

(i) Dual numbers to compute gradients with forward AD.
(ii) Measurements.jl

Thanks to multiple dispatch, we can compare all the different types of numbers through the same type-agnostic code.
Timings for \((d, n, m, N) = (2, 20, 80, 1000)\)

<table>
<thead>
<tr>
<th>Type</th>
<th>Error</th>
<th>Time</th>
<th>Allocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float64</td>
<td>2.68084e110</td>
<td>0.121672</td>
<td>286112.0</td>
</tr>
<tr>
<td>MultiFloat{Float64,1}</td>
<td>4.1367e109</td>
<td>0.124149</td>
<td>296656.0</td>
</tr>
<tr>
<td>Float128</td>
<td>4.51278e74</td>
<td>1.84456</td>
<td>467600.0</td>
</tr>
<tr>
<td>Double64</td>
<td>2.10301e78</td>
<td>0.516504</td>
<td>467600.0</td>
</tr>
<tr>
<td>MultiFloat{Float64,2}</td>
<td>3.84925e78</td>
<td>0.26139</td>
<td>486208.0</td>
</tr>
<tr>
<td>MultiFloat{Float64,3}</td>
<td>-6.13933e44</td>
<td>0.66715</td>
<td>676464.0</td>
</tr>
<tr>
<td>MultiFloat{Float64,4}</td>
<td>3.1583e14</td>
<td>1.33451</td>
<td>864800.0</td>
</tr>
<tr>
<td>MultiFloat{Float64,5}</td>
<td>2.367e-8</td>
<td>2.39511</td>
<td>1.05234e6</td>
</tr>
<tr>
<td>MultiFloat{Float64,6}</td>
<td>2.36701e-8</td>
<td>4.46531</td>
<td>1.25013e6</td>
</tr>
<tr>
<td>MultiFloat{Float64,7}</td>
<td>2.36701e-8</td>
<td>6.50383</td>
<td>1.44102e6</td>
</tr>
<tr>
<td>MultiFloat{Float64,8}</td>
<td>2.36701e-8</td>
<td>9.84249</td>
<td>1.63157e6</td>
</tr>
<tr>
<td>BigFloat</td>
<td>2.36701e-8</td>
<td>12.6092</td>
<td>8.16053e9</td>
</tr>
</tbody>
</table>

See my discourse thread (link) for more details
Multiple dispatch and type-agnostic code
Multiple dispatch : principe et bienfaits.

Running previous timings was as simple as:

```plaintext
... 
for T in list_of_types 
    inputs = T(inputs) 
    result[T] = Algorithm(inputs) # Type agnostic function 
end
... 
```

Seek type stability: A separate method of the Algorithm function will be compiled for each set of concrete input types, and if the rest of the stack is also type-stable, the compiler will specialize and compile everything to LLVM, providing a huge performance boost.

The compiler is there to help:

```plaintext
@code_native, @code_llvm, @code_warn_type, 
@btime, @elapsed, @allocated...
```
There is an insight in the community to code native type-agnostic stuff. This is a very good opportunity for us:

(i) ForwardDiff.jl: Automatic differentiation of loss functions.
(ii) Optim.jl: LBFGS, ParticleSwarm, Ants colonies, ...
(iii) Convex.jl: Disciplined Convex Programming (not for us)
(iv) COSMO.jl: Native general conic programming, to pair with Convex.jl.
(v) NonConvex.jl: Many Others algorithms

All these algorithms were coded once and works on all types. You could even differentiate through a gradient descent using Dual numbers, or optimize with ball arithmetics, it just works.

Easy DSL through Macros!
Modifying a package is quite easy. Simply add the package and dev it:

```julia
julia > Pkg.add("Optim")
julia > Pkg.dev("Optim")
```

Then modify the source to parallelize the right loop:

```julia
... Threads.@threads for i in 1:n_particles
    score[i] = value(f, X[:, i])
end ...
```

Commit, and a simple pull-request is enough to contribute your changes!

```
$ git add .
$ git commit -m "Threading the PSO loop"
```
Conclusion
Outside of the Julia ecosystem, optimisation in arbitrary precision usually requires re-implementation of at least one library:

(i) In python, you may use “mpmath” or “numba”, but you cannot use both. (and you still lack the optimisation routine)

(ii) In R, “rmpfr” cannot work with LBFS’s C++ routines: you need to recode everything.

Julia solves the two-language problem: Julia’s bare principles allows for a high level of code sharing and compatibility between codebases that did not expect each other.

Many things are still to be done in the ecosystem of packages since the language is young, and your Julia code is waited!