GENERIC PROGRAMMING IN JULIA

On example of AlternatingProjections.jl: a personal experience

Oleg Soloviev\textsuperscript{1,2}

15 October 2019

\textsuperscript{1}Numerics for Control and Identification Group
Delft Center for Systems and Control
Delft University of Technology

\textsuperscript{2}Flexible Optical B.V.
WHY JULIA?
A NEW AND PROMISING LANGUAGE

- Fast to develop
- Fast to execute
A NEW AND PROMISING LANGUAGE

- Fast to develop
- Fast to execute
- Just a new shiny thing
A NEW AND PROMISING LANGUAGE

- Fast to develop
- Fast to execute
- Just a new shiny thing

- Easy to learn

### Creating Matrices

<table>
<thead>
<tr>
<th></th>
<th>MATLAB</th>
<th>PYTHON</th>
<th>JULIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a matrix</td>
<td>A = [1 2; 3 4]</td>
<td>A = np.array([[1, 2], [3, 4]])</td>
<td>A = [1 2; 3 4]</td>
</tr>
<tr>
<td>2 x 2 matrix of zeros</td>
<td>A = zeros(2, 2)</td>
<td>A = np.zeros((2, 2))</td>
<td>A = zeros(2, 2)</td>
</tr>
<tr>
<td>2 x 2 matrix of ones</td>
<td>A = ones(2, 2)</td>
<td>A = np.ones((2, 2))</td>
<td>A = ones(2, 2)</td>
</tr>
</tbody>
</table>
| 2 x 2 identity matrix | A = eye(2, 2) | A = np.eye(2) | A = I # will adopt # 2x2 dims if demanded by
A NEW AND PROMISING LANGUAGE

- Fast to develop
- Fast to execute
- Just a new shiny thing
- Easy to learn
- They say it is very close to the “whiteboard coding”

### Creating Matrices

<table>
<thead>
<tr>
<th></th>
<th>MATLAB</th>
<th>PYTHON</th>
<th>JULIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a matrix</td>
<td>$A = [1 2; 3 4]$</td>
<td>$A = np.array([[1, 2], [3, 4]])$</td>
<td>$A = [1 2; 3 4]$</td>
</tr>
<tr>
<td>$2 \times 2$ matrix of zeros</td>
<td>$A = zeros(2, 2)$</td>
<td>$A = np.zeros((2, 2))$</td>
<td>$A = zeros(2, 2)$</td>
</tr>
<tr>
<td>$2 \times 2$ matrix of ones</td>
<td>$A = ones(2, 2)$</td>
<td>$A = np.ones((2, 2))$</td>
<td>$A = ones(2, 2)$</td>
</tr>
</tbody>
</table>
| $2 \times 2$ identity matrix | $A = eye(2, 2)$ | $A = np.eye(2)$ | $A = I # will adopt
# 2x2 dims if demanded by$ |
A talk from JuliaCon I’ve seen — they’ve written the whole book which is compiled in Julia.

Julia fits well algorithm descriptions.

Iteration for Adam are:

biased decaying momentum: \( v^{(k+1)} = \gamma_v v^{(k)} + (1 - \gamma_v) g^{(k)} \)  \hspace{1cm} (5.29)

biased decaying sq. gradient: \( s^{(k+1)} = \gamma_s s^{(k)} + (1 - \gamma_s) \left( g^{(k)} \odot g^{(k)} \right) \)  \hspace{1cm} (5.30)

corrected decaying momentum: \( \hat{v}^{(k+1)} = v^{(k+1)}/(1 - \gamma_v^k) \)  \hspace{1cm} (5.31)

corrected decaying sq. gradient: \( \hat{s}^{(k+1)} = s^{(k+1)}/(1 - \gamma_s^k) \)  \hspace{1cm} (5.32)

next iterate: \( x^{(k+1)} = x^{(k)} - \alpha \hat{v}^{(k+1)}/(\epsilon + \sqrt{\hat{s}^{(k+1)}}) \)  \hspace{1cm} (5.33)

```julia
mutable struct Adam <: DescentMethod
    α # learning rate
    γv # decay
    γs # decay
    ε # small value
    k # step counter
    v # 1st moment estimate
    s # 2nd moment estimate
end

function init!(M::Adam, f, Φf, x)
    M.k = 0
    M.v = zeros(length(x))
    M.s = zeros(length(x))
    return M
end

function step!(M::Adam, f, Φf, x)
    α, γv, γs, ε, k = M.α, M.γv, M.γs, M.ε, M.k
    v[:] = γv * v + (1 - γv) * Φf(x)
    s[:] = γs * s + (1 - γs) * Φf(x)^2
    M.k = k += 1
    v_hat = v ./ (1 - γv^k)
    s_hat = s ./ (1 - γs^k)
    return x - α * v_hat ./ (sqrt.(s_hat) .+ ε)
end
```
Projection on the convex set

\[ y = \Pr_A x \]
Projection on the convex set

\[ y = \text{Pr}_A x \]

How should I code this?
Abstract math concepts are not bytes in a computer.

Projection on the convex set $A$

$$y = \text{Pr}_A x$$

How should I code this? Is it possible to code abstract concepts?
JULIA’S ABSTRACT AND CONCRETE TYPES
Number types tree

Abstract, concrete and primitive types
ALTERNATING PROJECTIONS AND ITS RELATIVES
Find $x \in A \cap B$

1) just any would do
2) closest to $x^0$
FEASIBILITY PROBLEM

Find $x \in A \cap B$

1) just any would do
2) closest to $x^0$

E.g. 1) linear system

$\bullet$ Von Neumann: $A, B$ — convex $\Rightarrow$ use alternating projections (AP)

$\bullet$ H. Thao Ngueng et al: $A, B$ should be (sub)transversal

$\bullet$ The sets in PR problem are transversal
FEASIBILITY PROBLEM

E.g. 1) linear system
2) phase retrieval problem (PR):

\[ A = \{ x \in \mathbb{C}^{M \times M} : |x| = p \} \]
\[ B = \{ x \in \mathbb{C}^{M \times M} : |\mathcal{F}x| = P \} \]

\( p, P \) are the intensities in pupil and focal planes

Find \( x \in A \cap B \)

1) just any would do
2) closest to \( x^0 \)
FEASIBILITY PROBLEM

Find \( x \in A \cap B \)

1) just any would do
2) closest to \( x^0 \)

E.g. 1) linear system
2) phase retrieval problem (PR):

\[
\begin{align*}
A & = \{ x \in \mathbb{C}^{M \times M} : |x| = p \} \\
B & = \{ x \in \mathbb{C}^{M \times M} : |\mathcal{F} x| = P \}
\end{align*}
\]

\( p, P \) are the intensities in pupil and focal planes

\[ x^0 \]

\[ \bullet \]

- Von Neumann: \( A, B \) — convex \[ \implies \] use alternating projections (AP)

H. Thao Ngueng et al: \( A, B \) should be (sub)transversal

The sets in PR problem are transversal
FEASIBILITY PROBLEM

E.g. 1) linear system
2) phase retrieval problem (PR):

$A = \{ x \in \mathbb{C}^{M \times M} : |x| = p \}$

$B = \{ x \in \mathbb{C}^{M \times M} : |\mathcal{F}x| = P \}$

$p, P$ are the intensities in pupil and focal planes

Find $x \in A \cap B$

1) just any would do
2) closest to $x^0$

- Von Neumann: $A, B$ — convex $\implies$ use alternating projections (AP)
- H. Thao Ngueng et al: $A, B$ should be (sub)transversal
FEASIBILITY PROBLEM

Find $x \in A \cap B$

E.g. 1) linear system
2) phase retrieval problem (PR):

$A = \{x \in \mathbb{C}^{M \times M} : |x| = p\}$
$B = \{x \in \mathbb{C}^{M \times M} : |\mathcal{F}x| = P\}$

$p, P$ are the intensities in pupil and focal planes

1) just any would do
2) closest to $x^0$

- Von Neumann: $A, B$ — convex $\implies$ use alternating projections (AP)
- H. Thao Ngueng et al: $A, B$ should be (sub)transversal
- The sets in PR problem are transversal
Alternating projections for the feasibility problem

\[ a^1 = \text{Pr}_A x^0 \]
Alternating projections for the feasibility problem

\[ a^1 = \text{Pr}_A x^0 \]
\[ b^1 = \text{Pr}_B a^1 \]
Alternating projections for the feasibility problem

\[ \begin{align*}
  a^1 &= \text{Pr}_A x^0 \\
  b^1 &= \text{Pr}_B a^1 \\
  a^2 &= \text{Pr}_A b^1 
\end{align*} \]
Alternating projections for the feasibility problem

\[ a^1 = \text{Pr}_A x^0 \]
\[ b^1 = \text{Pr}_B a^1 \]
\[ a^2 = \text{Pr}_A b^1 \]
\[ \ldots \]
\[ b^k = \text{Pr}_B a^k, a^{k+1} = \text{Pr}_A b^k \]
Extension with forward and backward operators and not-so-convex sets:

\[
\begin{align*}
a^1 &= \text{Pr}_A x^0 \\
b^1 &= \text{Pr}_B a^1 \\
a^2 &= \text{Pr}_A b^1 \\
\vdots \\
b^k &= \text{Pr}_B a^k, a^{k+1} = \text{Pr}_A b^k
\end{align*}
\]

GS: \( X^k = \text{Pr}_p \mathcal{F} X^k, \ x^{k+1} = \text{Pr}_p \mathcal{F}^{-1} X^k \)
ALTERNATING PROJECTIONS FOR THE FEASIBILITY PROBLEM

Extension with forward and backward operators and not-so-convex sets:

\[ a^1 = \text{Pr}_A x^0 \]
\[ b^1 = \text{Pr}_B a^1 \]
\[ a^2 = \text{Pr}_A b^1 \]
\[ \ldots \]
\[ b^k = \text{Pr}_B a^k, a^{k+1} = \text{Pr}_A b^k \]

GS: \( X^k = \text{Pr}_P F X^k \), \( x^{k+1} = \text{Pr}_P F^{-1} X^k \)

TIP: \( h^k = \text{Pr}_H i/ o^k, o^{k+1} = \text{Pr}_O i/ h^k \)
FEASIBLESET, PROBLEM, AND ALGORITHM
ORTHOGONALISING THE CONCEPTS OF AP METHOD

Concepts as independent from each other as possible:

Three main **abstract types** with subtypes

1. **Set** -> Convex Set -> Linear subspace
2. **Problem** -> Feasibility problem
3. **Algorithm** -> AP

Their **concrete types** (implementations)

1. $ax = b$
2. PR Problem for given $p, P$
3. AP with parameters (?)
ORTHOGONALISING THE CONCEPTS OF AP METHOD

Concepts as independent from each other as possible:

Three main abstract types with subtypes

1. **Set** -> Convex Set -> Linear subspace
2. **Problem** -> Feasibility problem
3. **Algorithm** -> AP

Their concrete types (implementations)

1. \( ax = b \)
2. PR Problem for given \( p, P \)
3. AP with parameters (?)

```plaintext
abstract type FeasibleSet end

abstract type ConvexSet <: FeasibleSet end

abstract type Problem end

struct FeasibilityProblem <: Problem
    A::FeasibleSet
    B::FeasibleSet
    forward
    backward
end

abstract type APMethod end

struct AP <: APMethod
    maxit
    maxε
end
```
THE FIRST PROGRAM
We are ready to program it! For any problem, any initial guess, any algorithm:

```julia
function solve(p::Problem, x₀, alg::APMethod)
    error("Don't know how to solve ", typeof(p), " with method ", typeof(alg))
end

function project(x, feasset::FeasibleSet)
    error("Don't know how to project on ", typeof(feasset))
end

# If x allows subtraction, we can immediately write reflection operation
# for all cases (to be used in DR and DRAP):
reflect(x, feasset::FeasibleSet) = 2 * project(x, feasset) - x
```
We are ready to program it! For any problem, any initial guess, any algorithm:

```julia
function solve(p::Problem, x₀, alg::APMethod)
    error("Don't know how to solve ", typeof(p), ", ",
          typeof(alg))
end
```

If \( x \) allows subtraction, we can immediately write reflection operation for all cases (to be used in DR and DRAP):

```
reflect(x, feasset::FeasibleSet) = 2 * project(x, feasset) - x
```
We are ready to program it! For any problem, any initial guess, any algorithm:

```julia
function solve(p::Problem, x₀, alg::APMethod)
    error("Don't know how to solve ", typeof(p), " with method ", typeof(alg))
end

function project(x, feasset::FeasibleSet)
    error("Don't know how to project on ", typeof(feasset))
end
```

If \(x\) allows subtraction, we can immediately write reflection operation for all cases (to be used in DR and DRAP):

\[
\text{reflect}(x, \text{feasset}) = 2 \times \text{project}(x, \text{feasset}) - x
\]
We are ready to program it! For any problem, any initial guess, any algorithm:

```julia
function solve(p::Problem,x₀,alg::APMethod)
    error("Don't know how to solve ", typeof(p), " with method ",
        typeof(alg))
end

function project(x, feasset::FeasibleSet)
    error("Don't know how to project on ", typeof(feasset))
end
```

If `x` allows subtraction, we can immediately write reflection operation for all cases (to be used in DR and DRAP):

```julia
reflect(x, feasset::FeasibleSet) = 2 * project(x, feasset) - x
```
function solve(p::FeasibilityProblem, x⁰, alg::AP)
    A = p.A
    B = p.B
    forward = p.forward
    backward = p.backward
    maxit = alg.maxit
    maxϵ = alg.maxϵ

    k = 0
    xᵏ = x⁰
    ϵ = Inf

    while k < maxit && ϵ > maxϵ
        ỹ ᵖ = forward(xᵏ)
        yᵏ = project(ỹ ᵖ, B)
        ̃x ᵖ⁺¹ = backward(yᵏ)
        ̃x ᵗ⁺¹ = project(̃x ᵗ⁺¹, A)
        xᵏ⁺¹ = project(̃x ᵗ⁺¹, A)
        ϵ = LinearAlgebra.norm(xᵏ⁺¹ - xᵏ)
        xᵏ = xᵏ⁺¹
        # println(ϵ)
        k += 1
    end

    println("To converge with $ϵ accuracy, it took me $k iterations")
    return xᵏ
end
GROWING FLESH ON BONES
A convex and not convex examples of often used feasible sets

```julia
using APMethod, FeasibleSet, project, ConvexSet, FeasibilityProblem, AP

# Constraints
include("SupportConstraint.jl")
include("AmplitudeConstraint.jl")

# Algorithms
include("GerchbergSaxton.jl")
```
abstract type SupportConstrained <: ConvexSet
end

struct ConstrainedBySupport <: SupportConstrained
    support::Array{Bool}
end
export ConstrainedBySupport

function project(x, feasset::ConstrainedBySupport)
    return feasset.support .* x
end
```julia
using AlternatingProjections

S = ConstrainedBySupport([true, false, true])
ConstrainedBySupport(Bool[1, 0, 1])

x = [1, 2, 3]
3-element Array{Int64,1}:
1
2
3

project(x, S) == [1, 0, 3]
true
```
```julia
 julia> using AlternatingProjections

 julia> S = ConstrainedBySupport([true, false, true])
 ConstrainedBySupport(Bool[1, 0, 1])

 julia> x = [1, 2, 3]
 3-element Array{Int64,1}:
 1
 2
 3

 julia> project(x, S) == [1, 0, 3]
 true

 julia> S2 = ConstrainedBySupport([1, 0, 1])
 ConstrainedBySupport(Bool[1, 0, 1])
```
Projection doesn’t depend on the support dimension

```julia
julia> S3 = ConstrainedBySupport([0 0 1 0 0; 0 0 1 0 0; 1 1 1 1 1; 0 0 1 0 0; 0 0 1 0 0];
julia> x = rand(Int8, 5,5)
5×5 Array{Int8,2}:
  30  -92  16  127   5
  108  100  126  -111  86
  37  -38  -26  -53  -3
  27  114  -103  29   29
  85  -7  -86  -68  -49
julia> project(x, S3)
5×5 Array{Int8,2}:
  0  0  16  0  0
  0  0  126 0  0
  37  -38  -26  -53  -3
  0  0  -103 0  0
  0  0  -86 0  0
```
julia> P = ConstrainedBySupport([1, 0, 1, 0, 1])
ConstrainedBySupport(Bool[1, 0, 1, 0, 1])

julia> Q = ConstrainedBySupport([1, 1, 1, 0, 0])
ConstrainedBySupport(Bool[1, 1, 1, 0, 0])

julia> prb = FeasibilityProblem(P, Q, identity, identity)
FeasibilityProblem(ConstrainedBySupport(Bool[1, 0, 1, 0, 1]),
ConstrainedBySupport(Bool[1, 1, 1, 0, 0]), identity, identity)

julia> mth = AP(200, 0.001)
AP(200, 0.001)

julia> sol = solve(prb, [1, 1, 1, 1, 1], mth)
To converge with 0.0 accuracy, it took me 2 iterations
5-element Array{Int64,1}:
1
0
1
0
0
Just in the same way as explaining in a mathematical prove, we should be able to extend AP to Gerchberg-Saxton method for phase retrieval.

Just introduce the correct sets and explain how to project on them:

$$A = \{ x \in \mathbb{C}^{M \times M} : |x| = p \}$$

$$\Pr_A = p \cdot \frac{x}{|x|}$$
abstract type AmplitudeConstrainedSet <: FeasibleSet end
export AmplitudeConstrainedSet

struct ConstrainedByAmplitude <: AmplitudeConstrainedSet
  amp::Array{T} where T <: Real  #todo nongnegative
end
export ConstrainedByAmplitude

function project(x, feasset::ConstrainedByAmplitude)
  return feasset.amp .* exp.(im * angle.(x))
end
**AMPLITUDE CONSTRAINT: TESTING**

```julia
julia> A = ConstrainedByAmplitude([1, sqrt(2), 5])
ConstrainedByAmplitude([1.0, 1.4142135623730951, 5.0])

julia> y = [2im, -2 + 2im, 6 - 8im]
3-element Array{Complex{Int64},1}:
 0 + 2im
-2 + 2im
 6 - 8im

julia> project(y, A) ≈ [im, -1 + im, 3 - 4im]
true
```
julia> A = ConstrainedByAmplitude([1, sqrt(2), 5])
ConstrainedByAmplitude([1.0, 1.4142135623730951, 5.0])

julia> y = [2im, -2 + 2im, 6 - 8im]
3-element Array{Complex{Int64},1}:
  0 + 2im
  -2 + 2im
  6 - 8im

julia> project(y, A) ≈ [im, -1 + im, 3 - 4im]
true
julia> y = zeros(ComplexF32, 10, 10);
julia> y[1:5,1:5] = randn(ComplexF32, 5, 5);

julia> using FFTW
julia> Y = fft(y)

julia> pr = FeasibilityProblem(ConstrainedByAmplitude(abs.(y)),
                           ConstrainedByAmplitude(abs.(Y)), fft, ifft);

julia> gs = AP(3000, 1e-18);

julia> z = solve(pr, ones(size(y)), gs)
To converge with 3.6638411145107034e-16 accuracy, it took me 3000
iterations ...

julia> abs.(fft(z)) ≈ abs.(Y)
true

julia> abs.(z) ≈ abs.(y)
true
julia> y = zeros(ComplexF32,10,10);  
julia> y[1:5,1:5] = randn(ComplexF32, 5,5);

julia> using FFTW  
julia> Y = fft(y)

julia> pr = FeasibilityProblem(ConstrainedByAmplitude(abs.(y)),  
← ConstrainedByAmplitude(abs.(Y),fft, ifft);  
julia> gs=AP(3000,1e-18);

julia> z= solve(pr, ones(size(y)), gs)  
To converge with 3.6638411145107034e-16 accuracy, it took me 3000  
← iterations  
...

julia> abs.(fft(z)) ≈ abs.(Y)  
true

julia> abs.(z) ≈ abs.(y)  
true

Check it on a more serious example, dude!
METHODOLOGY
Further extensions are possible in a similar way: add new types and extend methods on them

**TIP:** PositiveSupport <: ConvexSet and function deconvolve(i,x)

**DR** change solve(pr,x⁰, method::DR) to use reflection-based operators
Further extensions are possible in a similar way: add new types and extend methods on them

**TIP:** `PositiveSupport <: ConvexSet` and `function deconvolve(i,x)`

**DR** change `solve(pr,xº, method::DR)` to use reflection-based operators

**vector GS:** new type of set or change `solve(pr,xº, method::vectorAP)` or new type of problem?
struct GS <: APMethod #todo should be sets part of this or added to the step! only?
    a::AmplitudeConstraint
    A::AmplitudeConstraint
end

GS(a::Array, A::Array) = GS(AmplitudeConstraint(a), AmplitudeConstraint(A))

function init!(alg::GS, x⁰)
    a = alg.a
    A = alg.A
    if !( size(a) == size(A) == size(x⁰) )
        println("cannot intialise Gerchberg-Saxton, dimensions do not match")
        # raise error or pad a smaller array with zeroes
        # also can include realignment of the sets to remove any linear phase term. But then need to make GS mutable
        return
    end
end

function step!(alg::GS, xᵏ)
    a = alg.a
    A = alg.A
    return project(a, ifft(project(A, fft(xᵏ)))))
end
```plaintext
function apstep(xᵏ, A::FeasibleSet, B::FeasibleSet, forward, backward)
  ŷ k = forward(xᵏ)
  y k = project(ŷ k, B)
  x̃ k⁺¹ = backward(y k)
  x k⁺¹ = project(x̃ k⁺¹, A)
end

function apsolve(A, B, ::Type{T}; x⁰=zeros(size(A)), maxit = 20, maxϵ = 0.01) where {T<:APMethod}
  alg = T(A,B)
  xprev = x⁰
  x = xprev
  i = 0
  ϵ = Inf

  while i < maxit && ϵ > maxϵ
    x = apstep(xprev, alg.a, alg.A, alg.forward, alg.backward)
    ϵ = LinearAlgebra.norm(x - xprev)
    xprev = x
  i += 1
end

println("To converge with $ϵ accuracy, it took me $i iterations")
return x
end
```
Multiple dispatch: data defines the behaviour of the function (all variables of the function)

OOP: object has a collection of methods (functions), that is only one variable (object type) defines the behaviour of the function

Classes & inheritance vs types and methods

gs.solve(.,.,.) and tip.solve(.,.,.) vs solve(.,.,::GS) and solve(.,.,::TIP)
Multiple dispatch: data defines the behaviour of the function (all variables of the function)

OOP: object has a collection of methods (functions), that is only one variable (object type) defines the behaviour of the function

Classes & inheritance vs types and methods

```
gs.solve(.,.,.) and tip.solve(.,.,.) vs solve(.,.,::GS) and solve(.,.,::TIP)
gs.solve(.,.,.) and tip.solve(.,.,.) vs solve(::XX,.,::GS) and solve(::XX,.,::TIP)
```
Multiple dispatch: data defines the behaviour of the function (all variables of the function)

OOP: object has a collection of methods (functions), that is only one variable (object type) defines the behaviour of the function

Classes & inheritance vs types and methods

\texttt{gs.solve(.,.,.) and tip.solve(.,.,.)} vs \texttt{solve(.,.,::GS) and solve(.,.,::TIP)}

\texttt{gs.solve(.,.,.) and tip.solve(.,.,.)} vs \texttt{solve(::XX,.,::GS) and solve(::XX,.,::TIP)}

The second approach is more flexible (can add new things without worrying about the existing ones) and generalisation of auto-method selection \texttt{solve(pr,x^o)} is easy
CONCLUSIONS
· Writing in Julia promotes “Generic programming” paradigm
· It is indeed similar to mathematical description
· Abstract types represent mathematical concepts and concrete types their implementations
· The orthogonalisation of the types and methods is an iterative process which might bring better understanding and opens possibilities for experimenting with novel methods
https://github.com/olejorik/AlternatingProjections.jl
Instructions on how to get started and on the workflow included
https://olejorik.github.io/AlternatingProjections.jl/docs/build/index.html#Workflow-and-package-structure-1

References:

- https://julialang.org/
- https://www.youtube.com/playlist?list=PLP8iPy9hna6StY9tIFIUN3F_co9A0zh0H — JuliaCon 2019
- https://julia.quantecon.org/more_julia/generic_programming.html
- https://docs.junolab.org/latest/ — IDE
- https://plugins.jetbrains.com/plugin/10413-julia — IDE
- https://www.youtube.com/watch?v=QVmU29rCjaA — developing packages in Julia