GENERIC PROGRAMMING IN JULIA

On example of AlternatingProjections.jl: a personal experience

Oleg Soloviev[1,2]

15 October 2019

[1]Numerics for Control and Identification Group
Delft Center for Systems and Control
Delft University of Technology

[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

MATLAB	PYTHON	JULIA
Create a matrix		
`A = [1 2; 3 4]`	`A = np.array([[1, 2], [3, 4]])`	`A = [1 2; 3 4]`
2 x 2 matrix of zeros		
`A = zeros(2, 2)`	`A = np.zeros((2, 2))`	`A = zeros(2, 2)`
2 x 2 matrix of ones		
`A = ones(2, 2)`	`A = np.ones((2, 2))`	`A = ones(2, 2)`
2 x 2 identity matrix		
`A = eye(2, 2)`	`A = np.eye(2)`	`A = I # will adopt` `# 2x2 dims if demanded by`

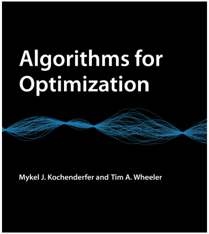
- Fast to develop
- Fast to execute
- Just a new shiny thing
- Easy to learn
- They say it is very close to the "whiteboard coding"

Creating Matrices

MATLAB	PYTHON	JULIA
Create a matrix		
`A = [1 2; 3 4]`	`A = np.array([[1, 2], [3, 4]])`	`A = [1 2; 3 4]`
2 x 2 matrix of zeros		
`A = zeros(2, 2)`	`A = np.zeros((2, 2))`	`A = zeros(2, 2)`
2 x 2 matrix of ones		
`A = ones(2, 2)`	`A = np.ones((2, 2))`	`A = ones(2, 2)`
2 x 2 identity matrix		
`A = eye(2, 2)`	`A = np.eye(2)`	`A = I # will adopt` `# 2x2 dims if demanded by`

JULIA FITS WELL ALGORITHM DESCRIPTIONS

A talk from JuliaCon I've seen —they've written the whole book which is compiled in Julia



iteration for Adam are:

$$\text{biased decaying momentum: } \mathbf{v}^{(k+1)} = \gamma_v \mathbf{v}^{(k)} + (1-\gamma_v)\mathbf{g}^{(k)} \tag{5.29}$$

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

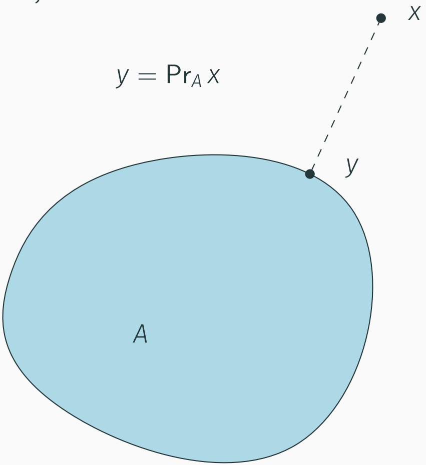
$$\text{corrected decaying momentum: } \hat{\mathbf{v}}^{(k+1)} = \mathbf{v}^{(k+1)} / (1-\gamma_v^k) \tag{5.31}$$

$$\text{corrected decaying sq. gradient: } \hat{\mathbf{s}}^{(k+1)} = \mathbf{s}^{(k+1)} / (1-\gamma_s^k) \tag{5.32}$$

$$\text{next iterate: } \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \hat{\mathbf{v}}^{(k+1)} / \left(\epsilon + \sqrt{\hat{\mathbf{s}}^{(k+1)}}\right) \tag{5.33}$$

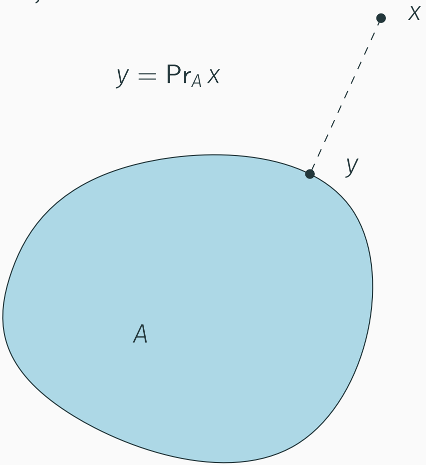
```julia
mutable struct Adam <: DescentMethod
    α # learning rate
    γv # decay
    γs # decay
    ϵ # small value
    k # step counter
    v # 1st moment estimate
    s # 2nd moment estimate
end
function init!(M::Adam, f, ∇f, x)
    M.k = 0
    M.v = zeros(length(x))
    M.s = zeros(length(x))
    return M
end
function step!(M::Adam, f, ∇f, x)
    α, γv, γs, ϵ, k = M.α, M.γv, M.γs, M.ϵ, M.k
    s, v, g = M.s, M.v, ∇f(x)
    v[:] = γv*v + (1-γv)*g
    s[:] = γs*s + (1-γs)*g.*g
    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



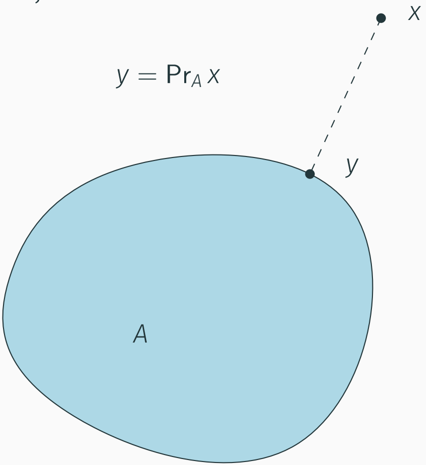
Projection on the convex set

$$y = \mathrm{Pr}_A\, x$$



How should I code this?

Projection on the convex set

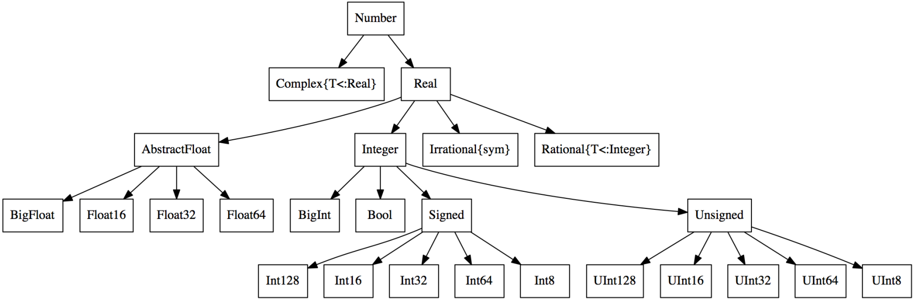


How should I code this?

Is it possible to code abstract concepts?

JULIA'S ABSTRACT AND CONCRETE TYPES

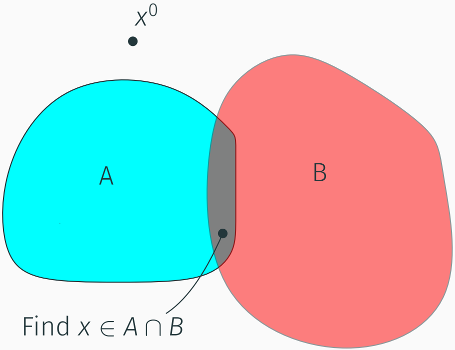
ABSTRACT TYPES FOR CONCEPTS, CONCRETE FOR DATA



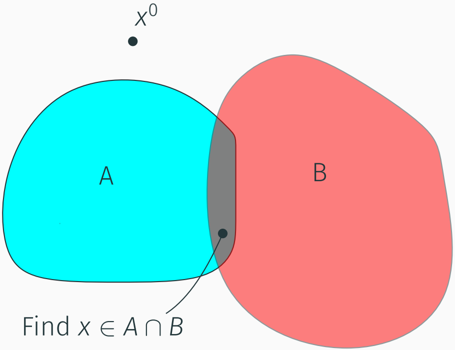
Number types tree

Abstract, concrete and primitive types

ALTERNATING PROJECTIONS AND ITS RELATIVES



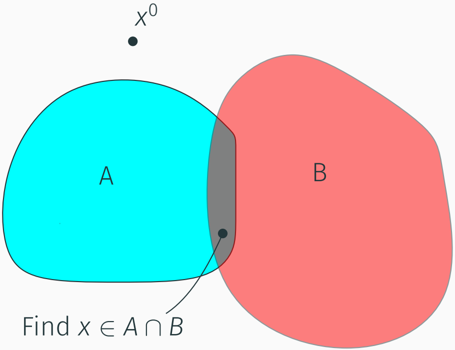
1) just any would do
2) closest to x^0



Find $x \in A \cap B$

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

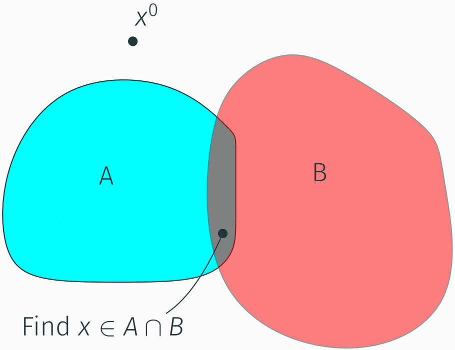
E.g. 1) linear system



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

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

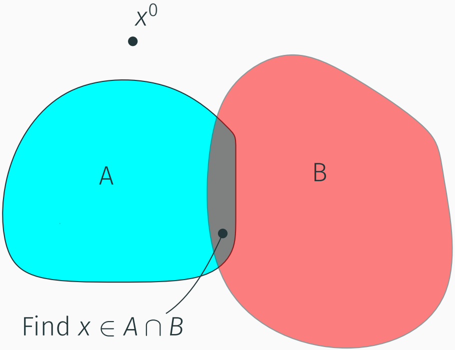
FEASIBILITY PROBLEM



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

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

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

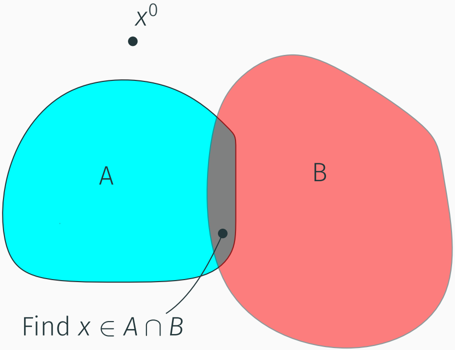


E.g. 1) linear system
2) phase retrieval problem (PR):
$A = \{x \in \mathbb{C}^{M\times M} : |x| = p\}$
$B = \{x \in \mathbb{C}^{M\times M} : |\mathcal{F}x| = P\}$
p, P are the intensities in pupil and focal planes

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

- Von Neumann: A, B — convex $\Longrightarrow$ use alternating projections (AP)
- H. Thao Nguyeng *et al*: A, B should be (sub)transversal

FEASIBILITY PROBLEM

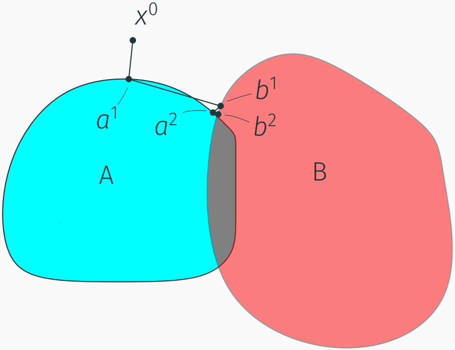


Find $x \in A \cap B$

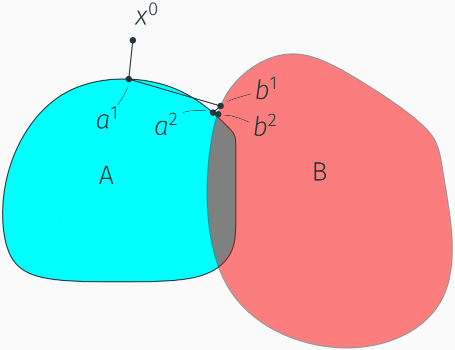
1) just any would do
2) closest to x^0

E.g. 1) linear system
2) phase retrieval problem (PR):
$A = \{x \in \mathbb{C}^{M\times M} : |x| = p\}$
$B = \{x \in \mathbb{C}^{M\times M} : |\mathcal{F}x| = P\}$
p, P are the intensities in pupil and focal planes

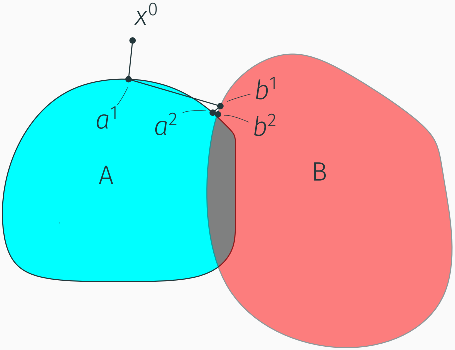
- Von Neumann: A, B — convex $\Longrightarrow$ use alternating projections (AP)
- H. Thao Ngueng *et al*: A, B should be (sub)transversal
- The sets in PR problem are transversal



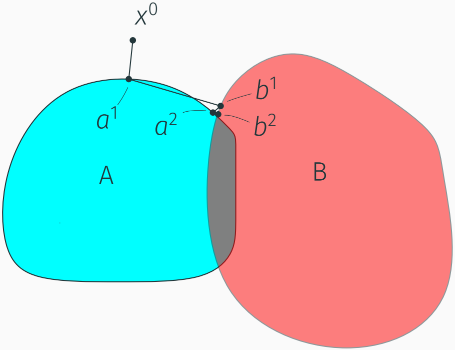
$$a^1 = \mathrm{Pr}_A\, x^0$$



$$a^1 = \mathrm{Pr}_A\, x^0$$
$$b^1 = \mathrm{Pr}_B\, a^1$$



$$a^1 = \Pr_A x^0$$
$$b^1 = \Pr_B a^1$$
$$a^2 = \Pr_A b^1$$



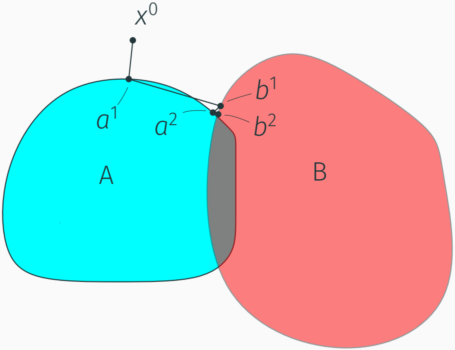
$a^1 = \mathrm{Pr}_A\, x^0$

$b^1 = \mathrm{Pr}_B\, a^1$

$a^2 = \mathrm{Pr}_A\, b^1$

$\dots$

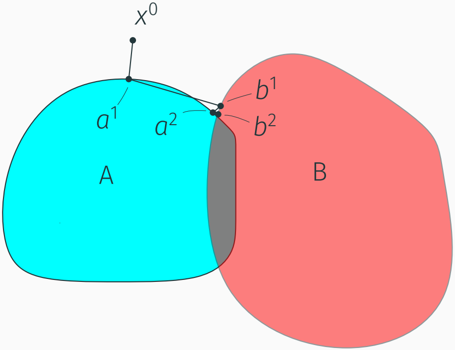
$b^k = \mathrm{Pr}_B\, a^k,\ a^{k+1} = \mathrm{Pr}_A\, b^k$



$a^1 = \mathrm{Pr}_A\, x^0$
$b^1 = \mathrm{Pr}_B\, a^1$
$a^2 = \mathrm{Pr}_A\, b^1$
$\dots$
$b^k = \mathrm{Pr}_B\, a^k, a^{k+1} = \mathrm{Pr}_A\, b^k$

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

$$\text{GS: } X^k = \mathrm{Pr}_P\, \mathcal{F}\, x^k,\ x^{k+1} = \mathrm{Pr}_p\, \mathcal{F}^{-1}\, X^k$$



$$a^1 = \mathrm{Pr}_A\, x^0$$
$$b^1 = \mathrm{Pr}_B\, a^1$$
$$a^2 = \mathrm{Pr}_A\, b^1$$
$$\dots$$
$$b^k = \mathrm{Pr}_B\, a^k, a^{k+1} = \mathrm{Pr}_A\, b^k$$

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

$$\text{GS: } X^k = \mathrm{Pr}_P\, \mathcal{F}\, x^k,\ x^{k+1} = \mathrm{Pr}_p\, \mathcal{F}^{-1} X^k$$

$$\text{TIP: } h^k = \mathrm{Pr}_{\mathcal{H}}\, i/_* o^k,\ o^{k+1} = \mathrm{Pr}_{\mathcal{O}}\, i/_* h^k$$

FEASIBLESET, PROBLEM, AND ALGORITHM

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 (?)

```
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:

AP METHOD REALISED IN JULIA (FOR *ANY* PROBLEM)

We are ready to program it! For any problem, any initial guess, any algorithm:

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

AP METHOD REALISED IN JULIA (FOR *ANY* PROBLEM)

We are ready to program it! For any problem, any initial guess, any algorithm:

```
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
```

We are ready to program it! For any problem, any initial guess, any algorithm:

```
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
```

AP METHOD REALISED IN JULIA

```
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 ᵏ = forward(xᵏ)
                yᵏ = project(ỹ ᵏ, B)
                x̃ ᵏ⁺¹ = backward(yᵏ)
                xᵏ⁺¹ = project(x̃ ᵏ⁺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
export APMethod, FeasibleSet, project, ConvexSet, FeasibilityProblem, AP

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

# algortihms
include(”GerchbergSaxton.jl”)
```

SUPPORT CONSTRAINT: DEFINING

```
abstract type SupportConstrained <: ConvexSet end

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

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

SUPPORT CONSTRAINT: TESTING

```
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
```

SUPPORT CONSTRAINT: TESTING

```
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])
```

SUPPORT CONSTRAINT: TESTING

Projection doesn't depend on the support dimension

```
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
```

SUPPORT CONSTRAINT AND AP

```
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:

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

$$\Pr_A = p \cdot \frac{x}{|x|}$$

AMPLITUDE CONSTRAIN: DEFINING

```julia
abstract type  AmplitudeConstrainedSet <: FeasibleSet end
export AmplitudeConstrainedSet

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

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

AMPLITUDE CONSTRAIN: TESTING

```
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
```

AMPLITUDE CONSTRAIN: TESTING

```
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> project(y, A)
3-element Array{Complex{Float64},1}:
6.123233995736766e-17 + 1.0im
-1.0 + 1.0000000000000002im
3.0000000000000004 - 3.9999999999999996im
```

AMPLITUDE CONSTRAIN, FOURIER, AND AP = GS

```
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,x⁰, method::DR)` to use reflection-based operators

vector GS: new type of set or change `solve(pr,x⁰, method::vectorAP)` or new type of problem?

DIFFERENT WAYS OF ORTHOGONALISATION: FIRST ATTEMPTS

```
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 realingment of the sets to remove any
                ↪  linear pahse  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
```

```
function apstep(xᵏ, A::FeasibleSet, B::FeasibleSet, forward, backward)
        ỹᵏ = forward(xᵏ)
        yᵏ = project(ỹᵏ, B)
        x̃ᵏ⁺¹ = backward(yᵏ)
        xᵏ⁺¹ = project(x̃ᵏ⁺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)`

JULIAN WAY: MULTIPLE DISPATCH

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
`gs.solve(.,.,.)` and `tip.solve(.,.,.)`	vs	`solve(.,.,::GS)` and `solve(.,.,::TIP)`
`gs.solve(.,.,.)` and `tip.solve(.,.,.)`	vs	`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 `solve(pr,x⁰)` 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=PLP8iPy9hna6StY9tIJIUN3F_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