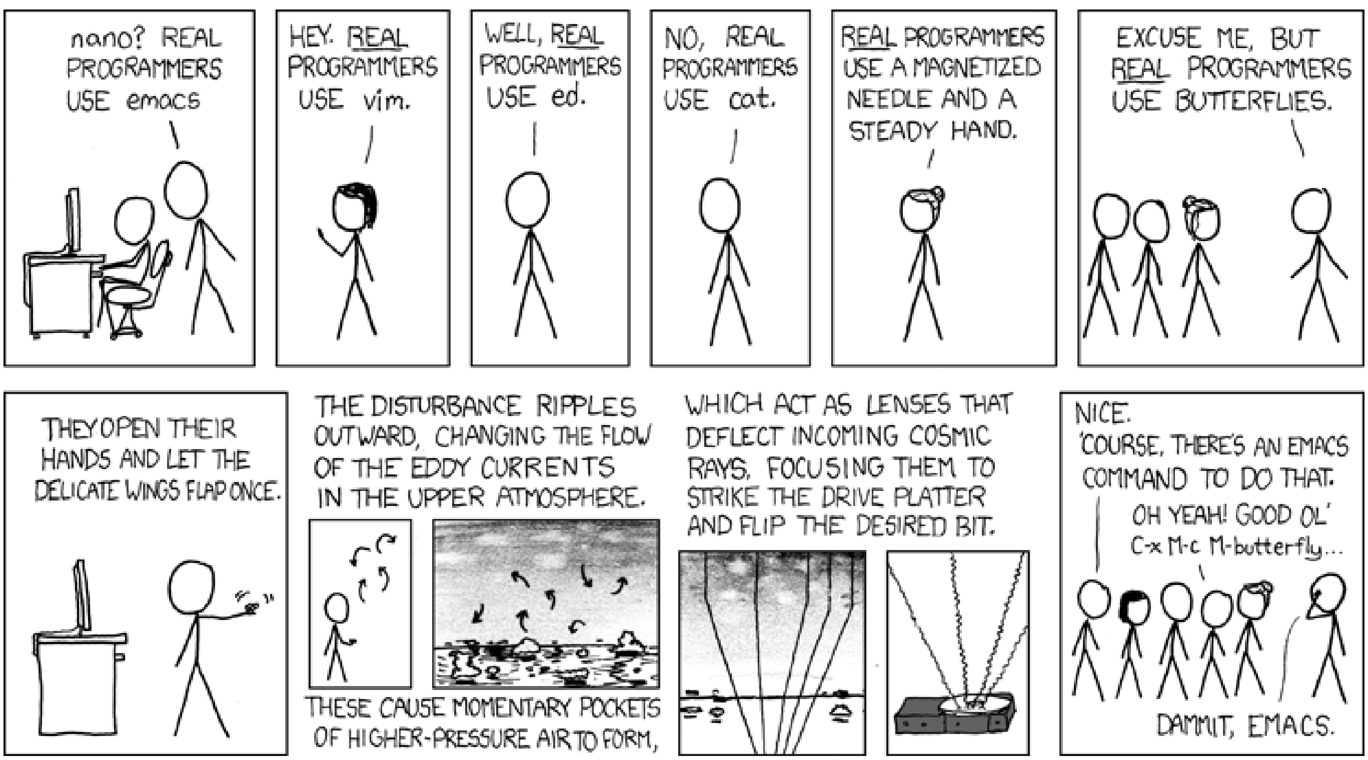
Present

Julia

Outline

- Julia - A Fresh Approach to Numerical Computation
- Julia - A *Fast* Approach to Numerical Computation
- Programming Paradigms and Multiple Dispatch
- Hands on Matching Pursuit

Preface



Everyone is entitled to their personal opinion, workflow and preferences. The focus should be on the scientific progress.

Julia : Raison d'être and Goals [5]

In short, because we are greedy.

- The speed of C
- The dynamism of Ruby
- The obvious syntax of Matlab
- The generalizability of Python
- The adhesivity of the Shell
- ...

Julia - A fresh approach to numerical computation [1]

Draft in high level language $\mapsto$ Reimplement in low level Language

Additional effort:

- Map datastructures correctly
- Ensure composability of datastructures and functions or methods
- Implement interfaces to Open Source Projects
- Add functionality to these
- ...

This is known as the **Two Language Problem** in computer science.

Let's compare Julia with Python

Python

- Object Oriented
- Dynamicly Interpreted
- Performant (?)

Julia

- Dynamicly Typed
- Just-In-Time Compiled
- Highly performant

The need for speed

```
f (generic function with 2 methods)
```

```
# Consider the function
f(x,y) = exp(-(x-y)^2)
```

```
1.1253517471925912e-7
```

```
# Lets see what happens
f(1, 5)
```

```
CodeInfo(
1 ─ %1 = x - y
│   %2 = Core.apply_type(Base.Val, 2)
│   %3 = (%2)()
│   %4 = Base.literal_pow(Main.workspace285.:^, %1, %3)
│   %5 = -%4
│   %6 = Main.workspace285.exp(%5)
└──      return %6
)
```

```
# And under the hood
@code_lowered f(0.2, 3.0)
```

```
[0.379052, 1.40459, 0.589247, -0.778208, -1.24411, 2.10893, -0.696557, -2.43836, 0.260874,
```

```
begin
    # For two arrays
    x = fill(1, 100000)
    x̂ = randn(100000)
end
```

```
MethodError: no method matching ^(::Vector{Float64}, ::Int64)
Closest candidates are:
^(!Matched::Union{AbstractChar, AbstractString}, ::Integer) at strings/basic.jl:718
^(!Matched::Complex{var"#s79"} where var"#s79"<:AbstractFloat, ::Integer) at complex.jl:8
^(!Matched::Complex{var"#s79"} where var"#s79"<:Integer, ::Integer) at complex.jl:820
...
```

1. macro expansion @ *none:0* [inlined]
2. literal_pow @ *none:0* [inlined]
3. f(::Vector{Int64}, ::Vector{Float64}) @ ***Other: 2***
4. macro expansion @ *timing.jl:210* [inlined]
5. (::Main.workspace285.var"#1#2")() @ ***Local: 2***
6. (::PlutoUI.var"#49#52"{Base.Iterators.Pairs{Union{}, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}, Main.workspace285.var"#1#2", Tuple{}})() @ *Terminal.jl:77*
7. with_logstate(::Function, ::Any) @ *logging.jl:491*
8. with_logger @ *logging.jl:603* [inlined]
9. macro expansion @ *Terminal.jl:76* [inlined]
10. macro expansion @ *Suppressor.jl:165* [inlined]
11. macro expansion @ *Terminal.jl:75* [inlined]
12. macro expansion @ *Suppressor.jl:127* [inlined]
13. macro expansion @ *Terminal.jl:74* [inlined]
14. macro expansion @ *Suppressor.jl:206* [inlined]
15. var"#with_terminal#46"(::Base.Iterators.Pairs{Union{}, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}, ::typeof(PlutoUI.with_terminal), ::Function) @ *Terminal.jl:73*
16. with_terminal(::Function) @ *Terminal.jl:72*
17. top-level scope @ ***Local: 1***

```julia
with_terminal() do
    @time f(x, x̂)
end
```

```
  0.075599 seconds (62.15 k allocations: 5.979 MiB, 26.11% gc time, 71.91% compila
tion time)
```

```julia
with_terminal() do
    @time fastf(x, x̂)
end
```

The error above is due to the missing vectorization of the function `f`. We can add this quite easily.

```
fastf (generic function with 1 method)
```

```julia
# Can we get it down? Hide at first
function fastf(x::AbstractVector{X}, y::AbstractVector{Y}) where {X, Y}
    # We want a common type
    T = promote_type(X, Y)
    # Convert the input to a common type
    x = convert.(T, x)
    y = convert.(T, y)
    # Preallocation of the result
    res = Vector{T}(undef, length(x))
    # Vectorize
    # We assume length(x) == length(y) -> Errors are not catched!
    # Additionally we add the @simd macro to instruct the compiler
    # to parallelize
    @simd for i in 1:length(x)
        # If we want, we could also use the @fastmath macro here
        # which does not improve our performance ( in this case )
        res[i] = f(x[i], y[i])
    end
    # The return argument
    return res
end
```

```
fastf! (generic function with 1 method)
```

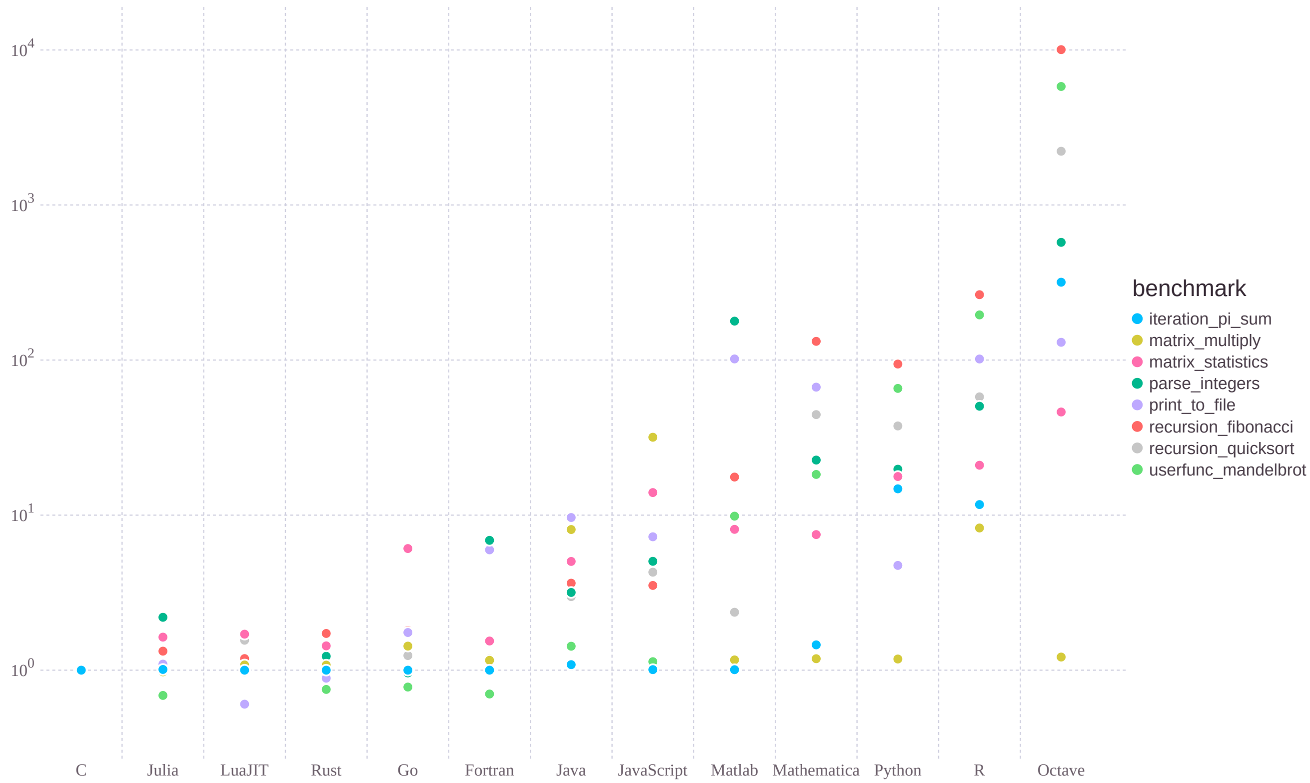
```julia
# Mutating function and even faster because we assume the same type
function fastf!(res::AbstractVector{T}, x::AbstractVector{T}, y::AbstractVector{T})
where {T}
    # Vectorize
    # We assume length(x) == length(y) -> Errors are not catched!
    # Additionally we add the @simd macro to instruct the compiler
    # to parallelize
    @inbounds @simd for i in 1:length(x)
        # If we want, we could also use the @fastmath macro here
        # which does not improve our performance ( in this case )
        res[i] = f(x[i], y[i])
    end
    # The return argument
    return
end
```

```
MethodError: no method matching ^(::Vector{Float64}, ::Int64)
Closest candidates are:
^(!Matched::Union{AbstractChar, AbstractString}, ::Integer) at strings/basic.jl:718
^(!Matched::Complex{var"#s79"} where var"#s79"<:AbstractFloat, ::Integer) at complex.jl:8
^(!Matched::Complex{var"#s79"} where var"#s79"<:Integer, ::Integer) at complex.jl:820
...
```

1. **macro expansion** @ *none:0* [inlined]
2. **literal_pow** @ *none:0* [inlined]
3. **f**(::Vector{Int64}, ::Vector{Float64}) @ ***Other: 2***
4. **macro expansion** @ *timing.jl:210* [inlined]
5. (::Main.workspace285.var"#mybench#7")(::Function, ::Vector{Int64}, ::Vararg{Any, N} where N) @ ***Local: 3***
6. (::Main.workspace285.var"#5#6")() @ ***Local: 5***
7. (::PlutoUI.var"#49#52"{Base.Iterators.Pairs{Union{}, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}, Main.workspace285.var"#5#6", Tuple{}})() @ *Terminal.jl:77*
8. **with_logstate**(::Function, ::Any) @ *logging.jl:491*
9. **with_logger** @ *logging.jl:603* [inlined]
10. **macro expansion** @ *Terminal.jl:76* [inlined]
11. **macro expansion** @ *Suppressor.jl:165* [inlined]
12. **macro expansion** @ *Terminal.jl:75* [inlined]
13. **macro expansion** @ *Suppressor.jl:127* [inlined]
14. **macro expansion** @ *Terminal.jl:74* [inlined]
15. **macro expansion** @ *Suppressor.jl:206* [inlined]
16. **var"#with_terminal#46"**(::Base.Iterators.Pairs{Union{}, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}, ::typeof(PlutoUI.with_terminal), ::Function) @ *Terminal.jl:73*
17. **with_terminal**(::Function) @ *Terminal.jl:72*
18. **top-level scope** @ ***Local: 1***

```julia
with_terminal() do
    # Benchmarking
    mybench(f::Function, args...) = @time f(args...)

    mybench(f, x, x̂)
    mybench(fastf, x, x̂)

    res = similar(x̂)
    mybench(fastf!, res, convert.(eltype(x̂), x), x̂)
end
```

Benchmarks



Microbenchmarks of Julia vs. different Languages as currently available **here**

BUT This is an old plot. Be careful.

▶ Just In Time (JIT) Compilation

Some Performance Pitfalls

▶ Benchmarking

▶ Memory Layout

▶ Referecing

Batteries included

Within the **LinearAlgebra** package - included in the standard library - we have BLAS and LAPACK at our fingertips.

```
10.0
```

```julia
# Simple dot
BLAS.dot(10, fill(1.0, 10), 1, fill(1.0, 20), 2)
```

```
[6, 9, 12]
```

```julia
# a*x + y
BLAS.axpy!(2, [1;2;3], [4;5;6])
```

```
(3×3 Matrix{Float64}:                , 3×3 Matrix{Float64}:                , [2, 3, 3])
   1.0           0.0  -2.16163e-15    0.744725   0.194975   1.78919
   3.97649e-16   0.0   1.0           -0.58645    1.86899    2.56491
  -2.68114e-16   1.0   1.07246e-15    0.523223  -0.726183  -0.207043
```

```julia
# Solve A X = B via LU(A) and overwrites B with the solution
begin
    A = randn(3,3)
    B = A * [1 0 0; 0 0 1; 0 1 0]
    LAPACK.gesv!(A, B)
end
```

Classical Dispatch

Functional

Global Namespace

$$f \in \mathcal{F}$$

Unique Functions

$$f(x_1, x_2, x_3, \ldots)$$

Results in

```
z = 3+i4
r = 0.1
complex_real_add(z, r)
```

Object Oriented

Local Namespace

$$f \in \mathcal{O}$$

Unique Functions in Namespace

$$o.\, f(x_1, x_2, x_3, \ldots)$$

Can result in

```
z = 3+i4
r = 0.1
z + r
```

Multiple Dispatch

Global namespace

$$f \in \mathcal{F}$$

With unique arguments

$$f : \mathcal{X}_1 \times \mathcal{X}_2 \cdots \mapsto \mathcal{Y}$$

```
z = 3+i4
r = 0.1
z + r
```

▶ Is this useful?

▶ But why exactly is this useful?

Dual Numbers

$$z = a + b\epsilon$$

with

$$\epsilon^2 = 0$$

```julia
# Type definition
struct DualNumber{T} <: Number
    a::T
    b::T
end
```

```julia
# Addition
Base.:+(x::DualNumber{T}, y::DualNumber{T}) where T = DualNumber(x.a + y.a, x.b + y.b)
```

z = DualNumber(1.0, 0.2)

```julia
z = DualNumber(1.0, 0.2)
```

DualNumber(100.0, 20.0)

```julia
sum([z for i in 1:100])
```

```
# Multiplication
Base.:*(x::DualNumber, y::DualNumber) = DualNumber(
    x.a*y.a, x.a*y.b+y.a*x.b
    )
```

```
DualNumber(1.0, 0.4)
```

```
z*z # Works
```

```
DualNumber(1024.0, 5120.0)
```

```
prod([(z^3 + z^2) for i in 1:10])
```

```
begin
    # Add multiplication
    Base.:*(x::Number, y::DualNumber{T}) where T = DualNumber(convert(T, x)*y.a, convert(T,x)*y.b)
    Base.:*(y::DualNumber{T}, x::Number) where T = DualNumber(convert(T, x)*y.a, convert(T,x)*y.b)
    # Add subtraction
    Base.:-(x::Number, y::DualNumber{T}) where T = DualNumber(convert(T,x)-y.a, -y.b)
    Base.:-(y::DualNumber{T}, x::Number) where T = DualNumber(y.a-convert(T,x), y.b)
    Base.:-(x::DualNumber, y::DualNumber) = x+(-1*y)
end
```

```
f (generic function with 2 methods)
```

```
f(x) = x^2 - 3*(x-2)^4
```

```
DualNumber(-2.0, 2.8000000000000003)
```

```
f(z)
```

```
-24.966299999999993
```

```
f(0.3)
```

```
-218
```

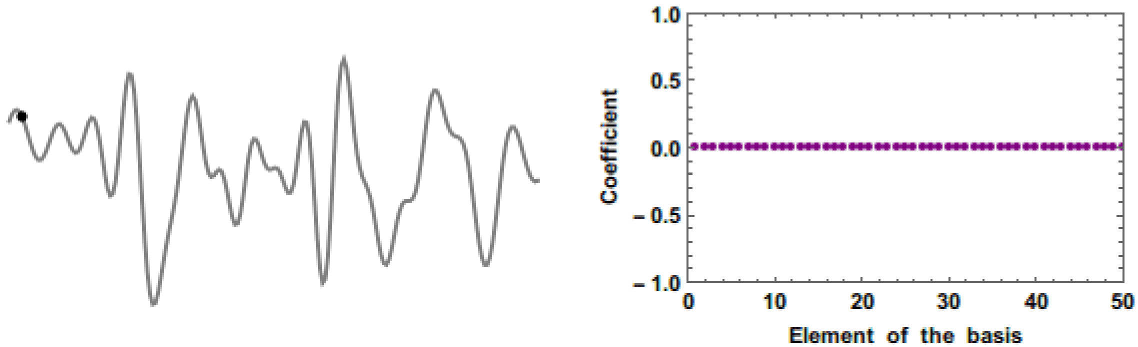
```
f(5)
```

```
191.2137 - 372.13199999999995im
```

```
f(0.3+im*3)
```

Matchig Pursuit [4]

$$\min_{\Xi} \quad \|\Xi\|_0, \quad \text{s.t.} \quad Y = \Psi(X)\Xi$$



Pseudo-Code

- Compare each element of the normalized dictionary $\Psi(X)$ to the signal Y via the inner product
- Use the largest resemblance as a coefficient $\xi = \Xi_{i,j}$
- Subtract the recovered signal and repeat until converged

We will use a more sophisticated version **based on this paper** where our goal is to find the right support Λ of the coefficient vector.

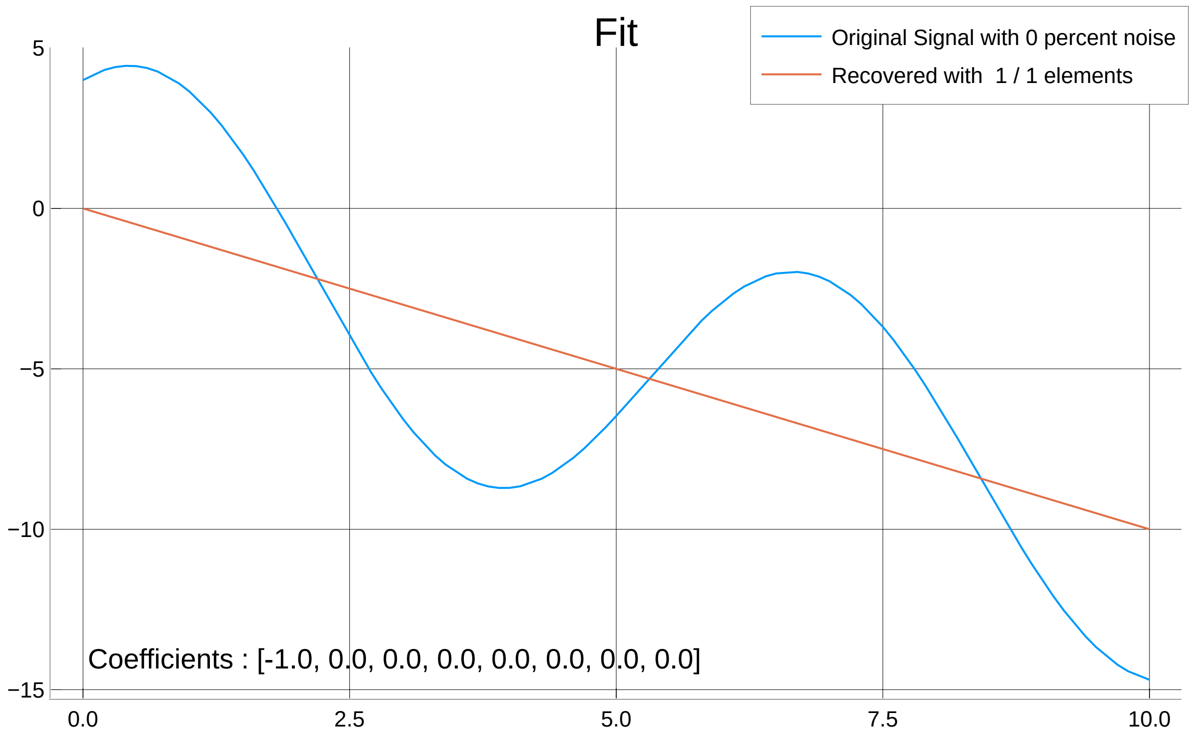
Note

`LinearAlgebra` is already imported

```
gOMP (generic function with 4 methods)
```

```julia
# Define the algorithm
function gOMP(y0::AbstractVector, Ψ::AbstractMatrix, K::Int = 2, S::Int = 1;
max_iter::Int = 100, ϵ::Real = eps())
    # Get the dimensions
    m = length(y0)
    m_psi, n = size(Ψ)
    # Assert the dimensionality
    @assert m == m_psi "Please provide consistent input sizes"
    # Assert the selector
    @assert S <=  min(K, m/K) "S <= min(K, m/K)"

    # Normalize
    ψ = deepcopy(Ψ)
    normalize!.(eachcol(ψ), 2)


    # Preconditioning
    P = ψ'pinv(ψ*ψ')
    # New matrix
    ψ = P*ψ
    y = P*y0
    # Iteration
    iters = 0

    # Support
    Λ = zeros(Bool, n)
    u = zeros(eltype(y), n)
    r = y
    amps = zeros(eltype(y), n)


    # Find the magnitudes
    for i in 1:max_iter
        # Compute the similarities via magnitude
        amps .= ψ'r
        # Get the largest entry
        idx = sortperm(abs.(amps), rev = true)[1:S]
        # Update the support
        Λ[idx] .= true
        # Update the coefficients
        u[Λ] .= (y' / ψ')[1, Λ]
        # Update r
        r .= y - ψ*u
        # Convergence
        if norm(r,2) < ϵ || sum(Λ) >= K
            # Just for debug
            #@debug "Early break after $i iterations with $(norm(r,2))"
            break
        end
    end
    # Last time to get the right coefficients
    u[Λ] .= (y0' / Ψ')[1, Λ]
    return u

end
```

```
# Generate some test data
begin
    # Independent variable
    t = 0.0:0.1:10.0;
    # Signal
    y = 3.0*sin.(t).*exp.(-t./50.0) + 4.0*cos.(t) - t;
    # Dictionary
    ψ = [
        t t.^2 t.^3 sin.(t) ones(eltype(y), length(y)) cos.(t) exp.(t) sin.(t).*exp.
(-t./50.0)
    ];
    nothing
end
```



Lets explore the data by adding a slider for the sparsity and a noise level.

Sparsity :

Noise :

```
gOMP (generic function with 4 methods)
```

```
# What about parallism?

# Simple

function gOMP(Y::AbstractMatrix, Ψ::AbstractMatrix, args...; kwargs...)

    # we know that the coefficients can be derived independent
    A = zeros(size(Ψ, 2), size(Y, 1)) # Init
    #Threads.@threads for i in 1:size(Y, 1)
    for i in 1:size(Y, 1)
        A[:, i] .= gOMP(Y[i, :], Ψ, args...; kwargs...)
    end
    return A
end
```

```
Y = 100×101 Matrix{Float64}:
 3.9926   4.17834  4.3087   4.38064  …  -14.2115  -14.4429  -14.5998  -14.6769
 4.00903  4.20684  4.30776  4.3943      -14.2196  -14.4291  -14.5958  -14.6988
 4.00066  4.15286  4.30094  4.40755     -14.2102  -14.4186  -14.6069  -14.6678
 4.02103  4.17958  4.3141   4.4129      -14.2252  -14.4364  -14.5861  -14.6815
 4.01213  4.18151  4.33174  4.40018     -14.2218  -14.4331  -14.6001  -14.698
 4.00256  4.15751  4.30317  4.40488  …  -14.2306  -14.4041  -14.5932  -14.6867
 4.00827  4.16841  4.31768  4.40545     -14.2219  -14.4204  -14.5807  -14.6984
 ⋮                                   ⋱                                ⋮
 4.01715  4.16797  4.29437  4.38201     -14.2401  -14.4049  -14.5691  -14.6966
 3.98759  4.18168  4.30862  4.38865  …  -14.2154  -14.4305  -14.5838  -14.6859
 3.98808  4.18097  4.31157  4.41082     -14.222   -14.4262  -14.5837  -14.6836
 3.99307  4.17842  4.32208  4.39345     -14.2258  -14.4091  -14.5824  -14.684
 4.02232  4.1989   4.32337  4.41797     -14.2194  -14.4256  -14.5855  -14.7017
 4.01931  4.1682   4.30764  4.39363     -14.2308  -14.4153  -14.5976  -14.6775
```

```
Y = vcat([(y+1e-2*randn(size(y)))' for i in 1:100]...)
```

```
  0.209139 seconds (165.72 k allocations: 90.200 MiB, 19.42% gc time, 47.75% compi
lation time)
```

```
with_terminal() do
    @time gOMP(Y, ψ, 4)
end
```

```
8×1 Matrix{Float64}:
 -0.9938917657989252
 -0.0017979565777127562
 -1.033544637337606e-6
  0.003759089684817375
  0.0
  4.002800632607579
  0.0
  2.948770296238229
```

```
sum(gOMP(Y, ψ, 4), dims = 2)/100
```

Ecosystem & Package Development

JuliaHub provides us with a nice, searchable database for all registered packages.

As an example for Package Development, we can have a look at **DataDrivenDiffEq.jl**.

And : **JuliaCon 2021** is around the corner!