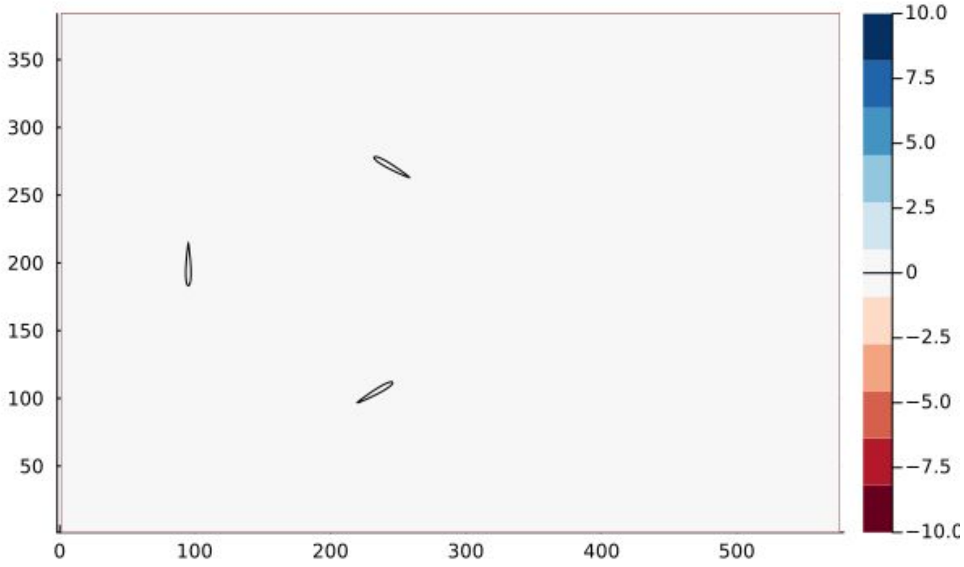
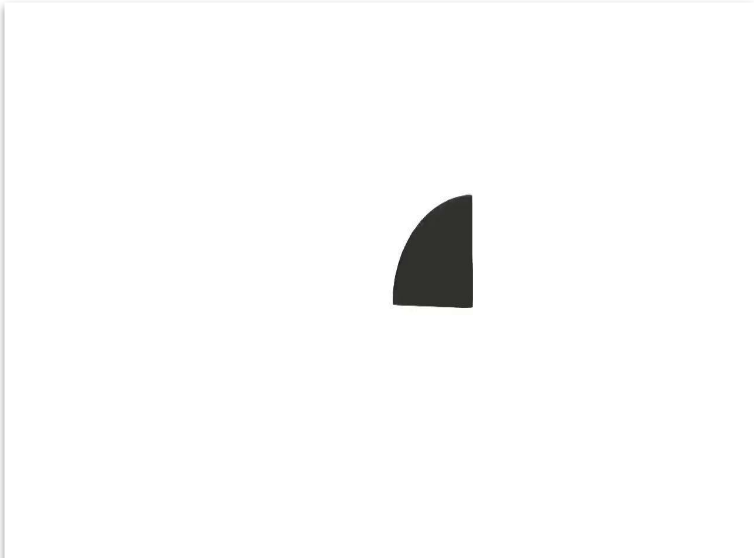
WaterLily.jl

An Introduction

Gabriel Weymouth, Bernat Font, Marin Lauber

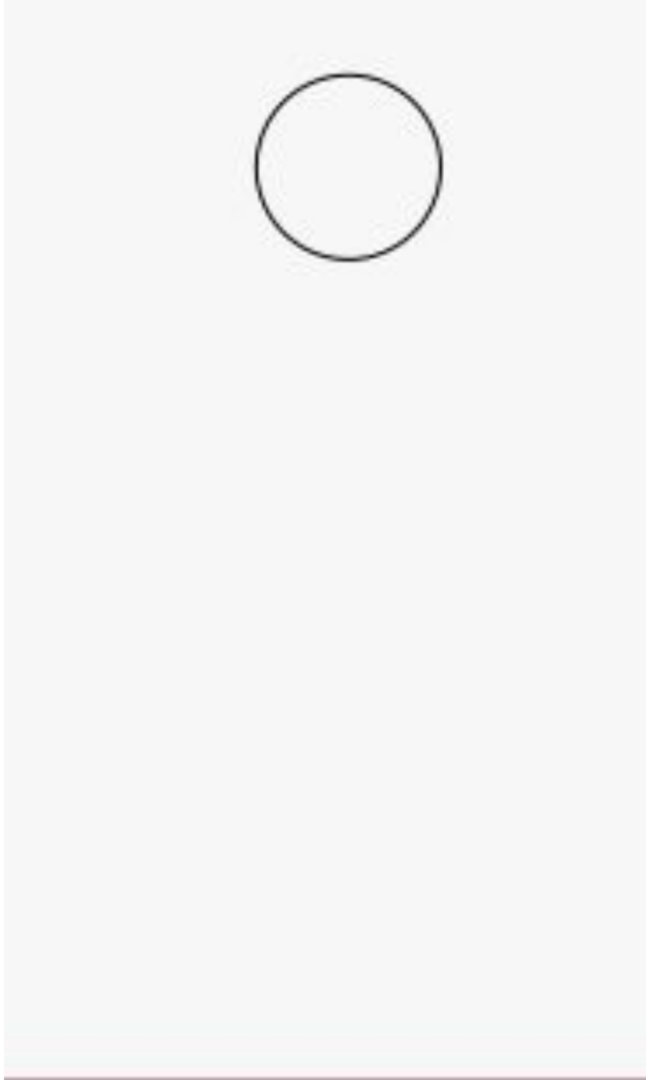
What can **WaterLily.jl** do?





Content

1. What are we doing?
    - just enough **WaterLily.jl** theory to get going
2. How to define a **body**?
    - signed distance function, parametric curves
3. How to make it **move**?
    - mapping
4. Live examples:
    - Cylinder
    - Accelerated disk
    - Vertical axis wind turbine



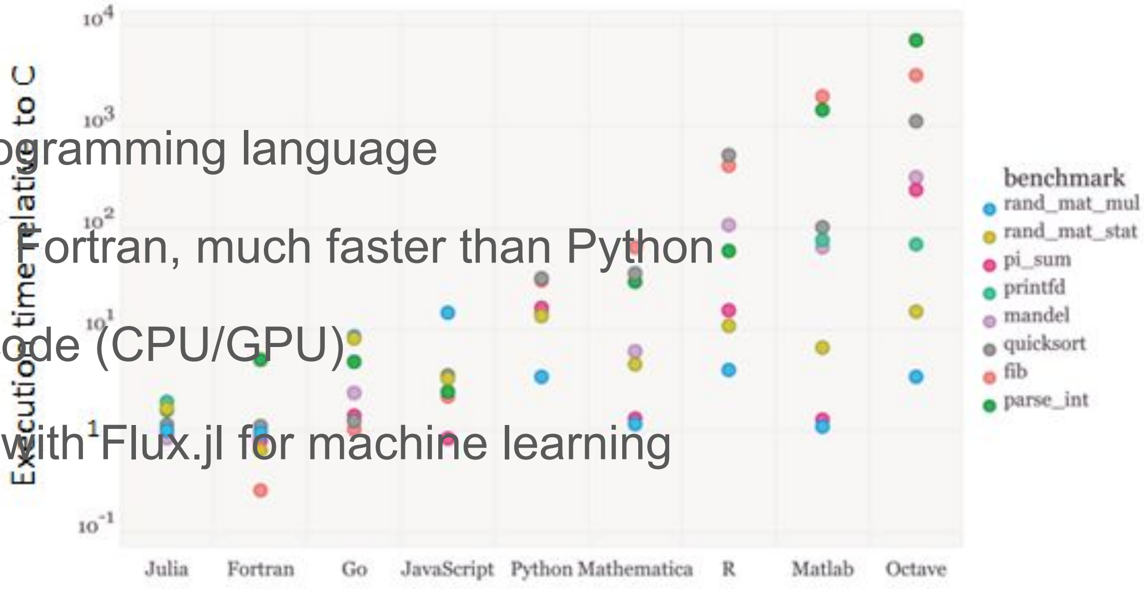
Why Julia for **WaterLily.jl**?

Fast, dynamic programming language

Much easier than Fortran, much faster than Python

Kernel agnostic code (CPU/GPU)

Potential binding with Flux.jl for machine learning



Performance comparison of various languages performing simple microbenchmarks. Benchmark execution time relative to C. (Smaller is better; C performance = 1.0.)

https://userpages.umbc.edu/~dfrey1/ench445/is-it-worth-the-effort-to-learn-Julia.html

Common and useful Julia syntax

```julia
# Array indexing like matlab, but with python syntax
a = zeros((10,10,10))
a[1:end,2:end-4,:] .= 1.0

#Array operation supported
sqrt(vec'*vec) -> dot(vec,vec)

#Can use any mathematical symbols
Re = U*L/ν

#Broadcasting operation must be explicit
a = @. [1., 2., 3.] + 1.0         or,       a = [1., 2., 3.] .+ 1.0

#Static Arrays for better memory management
using StaticArrays
a = SA[1. 2. 3.]

#to print and add end of line character…super annoying
println()

@my_macro # is a macro
my_func(...) and my_func!(...) # are not the same!
```

```julia
# Anonymous function
(x,y)->x*y

# Function arguments
function my_func(a,b=10;c=101)
    ...
end

# nested loops
for i ∈ 1:10, j ∈ 1:10
    ...
end
```

Much more at https://julialang.org/

Slack chanel extremely responsive!

WaterLily.jl: immersed-boundary, finite-volume, implicit large-eddy simulation code

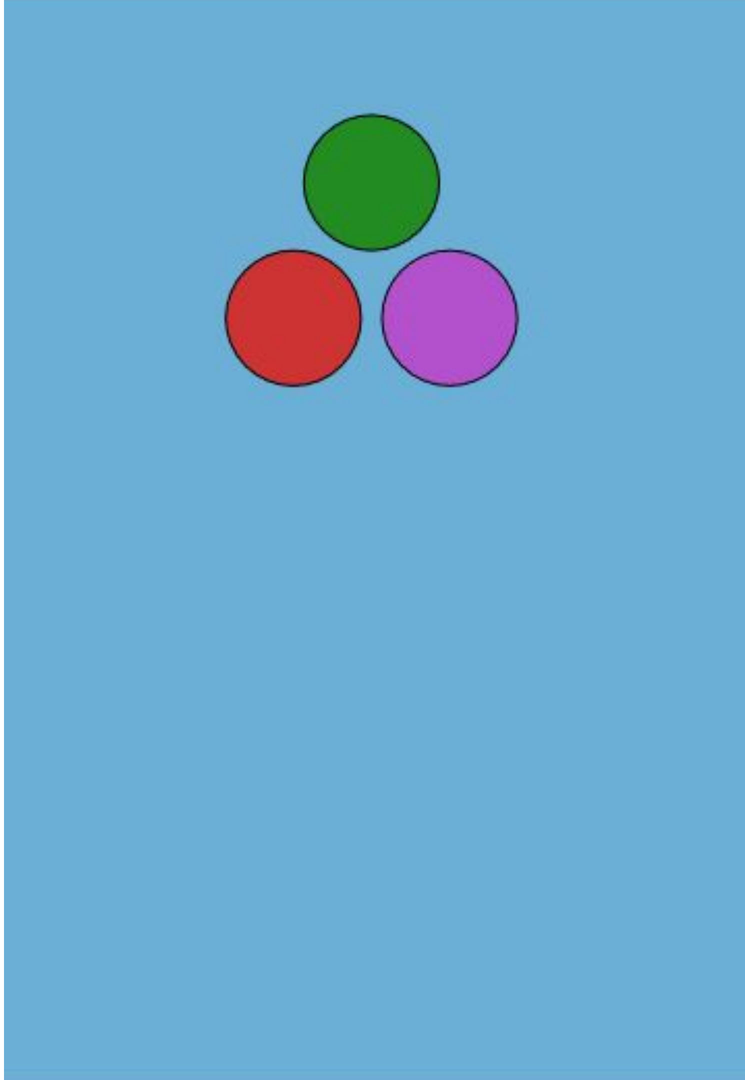
Cartesian grid -> **no meshing** required

Immersed boundary -> **arbitrary motion**

Finite-volume -> **conservation properties**

Implicit large eddy simulation -> **automatic turbulence modelling** (Re<1e6)

CPU/ GPU capability -> **very fast**



Immersed Boundary Methods

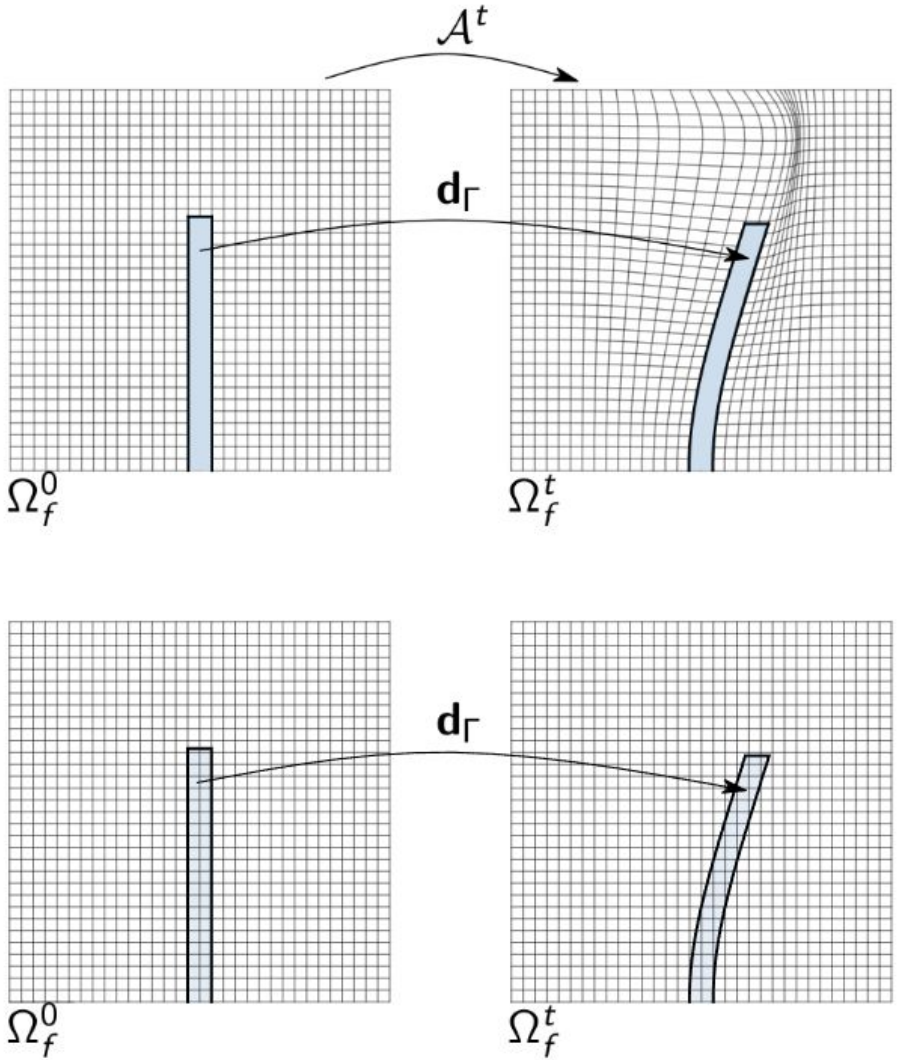
Cartesian grid using **unit cells**

Remove user-dependent mesh quality

Field are simple arrays

Arbitrary motion are possible

Fast method for the pressure equation



Boundary Data Immersion Method (BDIM)

Blend fluid and body equations in a single formulation (\mu is not viscosity!)

$$\vec{u}_\epsilon(\vec{x}) = \mu_0^\epsilon\, \vec{f} + (1 - \mu_0^\epsilon)\, \vec{b} + \boxed{\mu_1^\epsilon\, \frac{\partial}{\partial n}(\vec{f} - \vec{b}).}$$

The body and fluid equations are

$$\vec{b} = \vec{V}$$

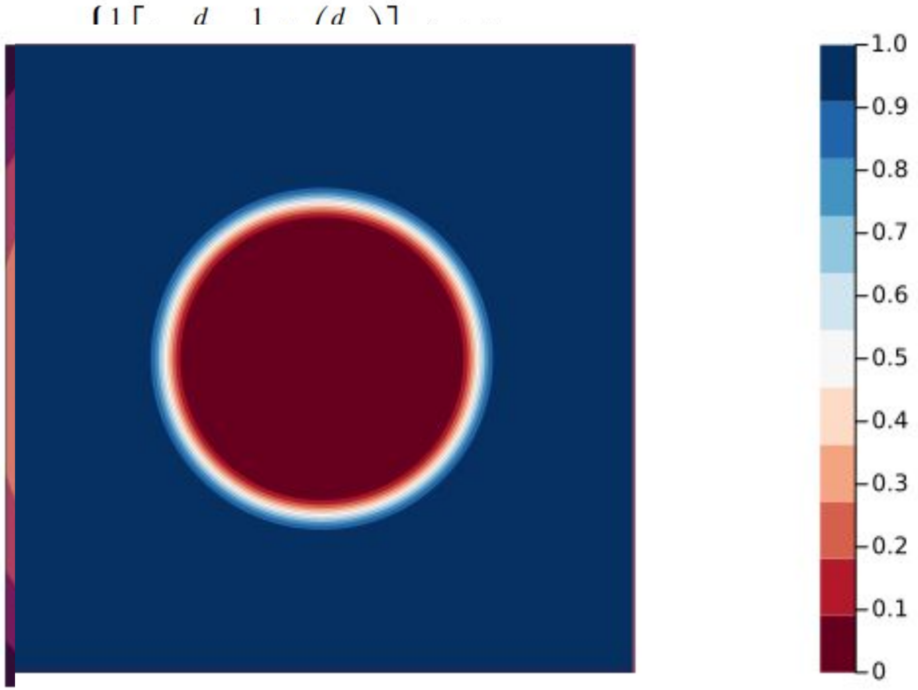
$$\vec{f}(\vec{u}, t_0 + \Delta t) = \underbrace{\vec{u}(t_0)}_{\vec{u}^0} + \underbrace{\int_{t_0}^{t_0+\Delta t} \left[-\left(\vec{u} \cdot \vec{\nabla}\right) \vec{u} + \nu \nabla^2 \vec{u} \right] \mathrm{d}t}_{\vec{R}_{\Delta t}(\vec{u})} - \underbrace{\int_{t_0}^{t_0+\Delta t} \frac{1}{\rho} \vec{\nabla} p \, \mathrm{d}t}_{\partial \vec{P}_{\Delta t}}$$

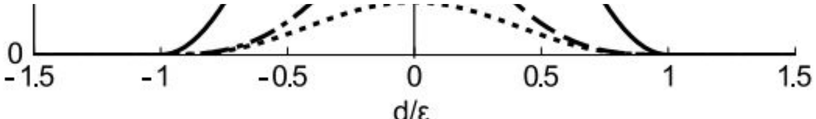
Evaluation of μ_0^ϵ , μ_1^ϵ and body velocity $\vec{V}$

The μ_0^ϵ μ_1^ϵ requires a **distance measure** to the body

Body velocity requires a **motion (map)**

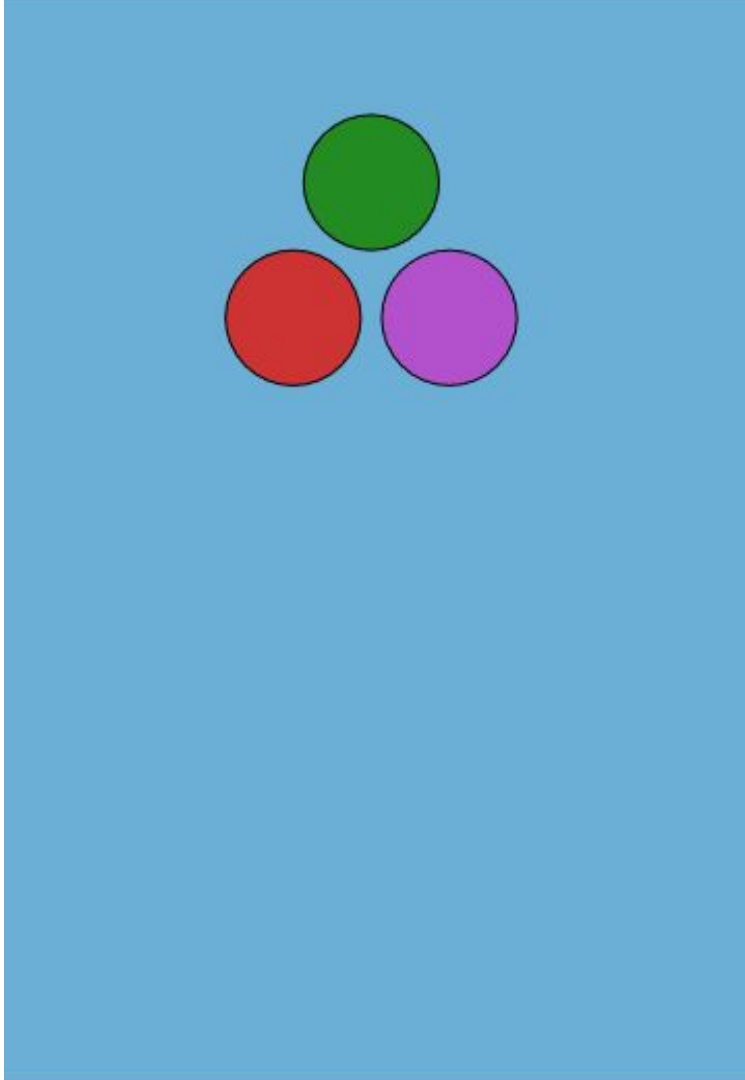
For second-order correction (and forces) a **normal vector** is also required





That’s enough theory, let’s make a simulation!

Any questions until now?



Summary of the Numerical schemes

Fully explicit 2nd order projection scheme for the time integration

Geometric multigrid Poisson solver for the pressure with Preconditioned Conjugate Gradient with Jacobi preconditioner.

Quick scheme for the convective term and central difference for the viscous term

Staggered variable arrangement on the grid (vector are face-centered and scalar are cell centered)

The ingredients required for a **Simulation**

```
1  # Prototype simulation
2  Simulation((Nx,Ny), (Ux,Uy), L; U, ν=U*L/Re, body)
```

(**Nx**,**Ny**): domain dimensions in number of cells

(**Ux**, **Uy**): velocity boundary condition

L: length scale of the problem (resolution)

U: velocity scale of the problem

nu: viscosity (really a Reynolds number)

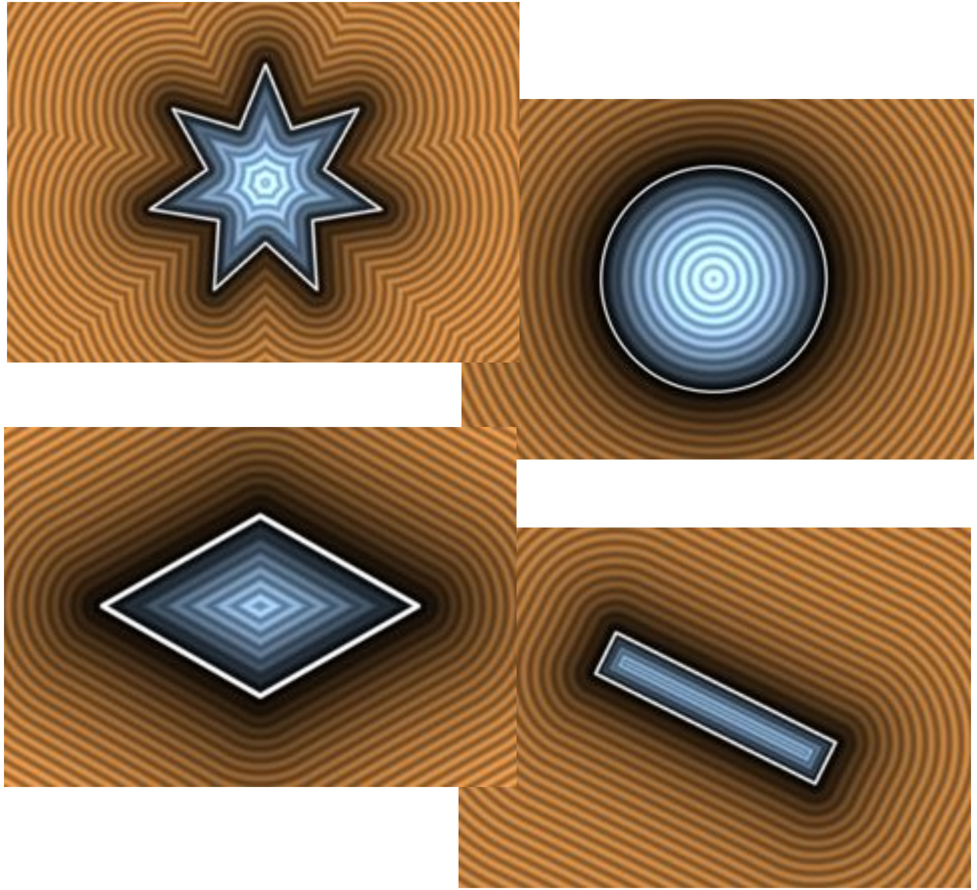
body: a body, can be of various type (AutoBody, ParametricBody, etc)

The simplest way to define a body is with (signed) distance functions

Only need an analytical expression for the SDF

```
# Prototype sdf function
function sdf(x,t)
    return √sum(abs2, x .- center) - radius
end

# make a body
body = AutoBody(sdf)
```

Implicit function can also be used (ellipse)

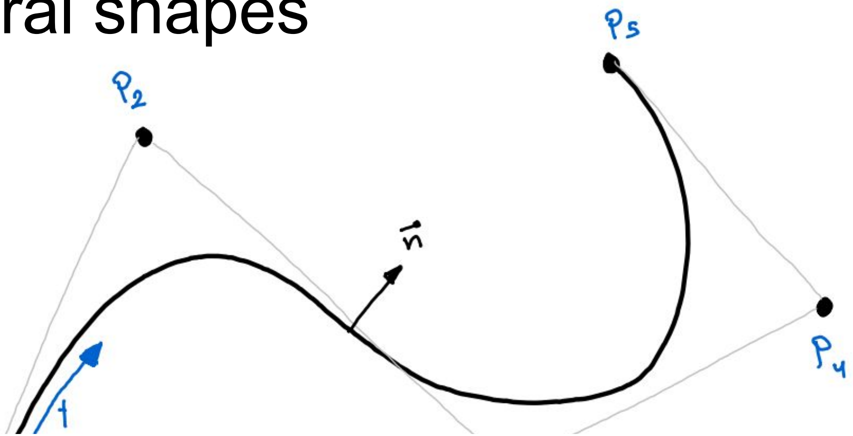


see the amazing https://iquilezles.org/articles/

AutoBody does all the hard work for you

```julia
function measure(body::AutoBody,x,t)
    # eval d=f(x,t), and n̂ = ∇f
    d = body.sdf(x,t)
    n = ForwardDiff.gradient(x->body.sdf(x,t), x)
    any(isnan.(n)) && return (d,zero(x),zero(x))

    # correct general implicit fnc f(x₀)=0 to be a pseudo-sdf
    #   f(x) = f(x₀)+d|∇f|+O(d²) ∴  d ≈ f(x)/|∇f|
    m = √sum(abs2,n); d /= m; n /= m

    # The velocity depends on the material change of ξ=m(x,t):
    #   Dm/Dt=0 → ṁ + (dm/dx)ẋ = 0 ∴  ẋ =-(dm/dx)\ṁ
    J = ForwardDiff.jacobian(x->body.map(x,t), x)
    dot = ForwardDiff.derivative(t->body.map(x,t), t)
    return (d,n,-J\dot)
end
```

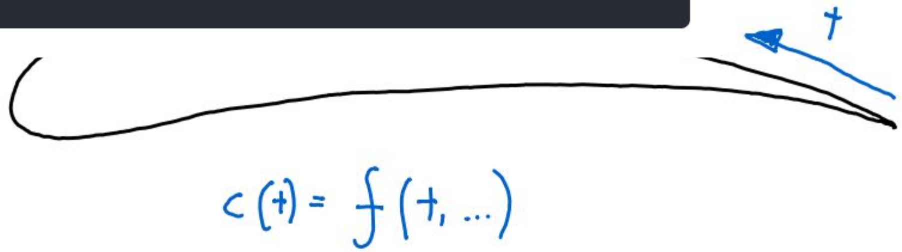
ParametricBody for more general shapes



Widen the possibilities

- NACA airfoil
- NURBS curves
- etc.

```
NACA(s) = 0.6f0*(0.2969f0s-0.126f0s^2-0.3516f0s^4+0.2843f0s^6-0.1036f0s^8)
curve(s,t) = L*SA[(1-s)^2,NACA(1-s)]
body = ParametricBody(curve,(0,1))
```



Similar to **AutoBody**, automatic differentiation to find normal and velocity

Using **ParametricBodies.jl** with **WaterLily.jl**

Until this package matures and is registered, you need to either add it via github

```
] add https://github.com/weymouth/ParametricBodies.jl
```

or download the github repo and then activate the environment

```
shell> git clone https://github.com/weymouth/ParametricBodies.jl
Cloning into 'ParametricBodies.jl'...
...
] activate ParametricBodies
] instantiate
```

https://github.com/weymouth/ParametricBodies.jl?tab=readme-ov-file

Motion via coordinate system mapping (map)

- maps a point **x** to a new point **x'**
- Points are first mapped and then the sdf is computed
- The limit is your coordinate mapping proficiency!
- Automatic diff in the background

Heave example

```
# Prototype sdf function
function map(x,t)
    return x .+ SA[0, h*sin(2π*St*t)]
end

# make a body
body = AutoBody(sdf,map)
```

```
dot = ForwardDiff.derivative(t->body.map(x,t), t)
```

Last ingredients: L the length scale of the simulation

Dimensionless solver (**u/U~1**)

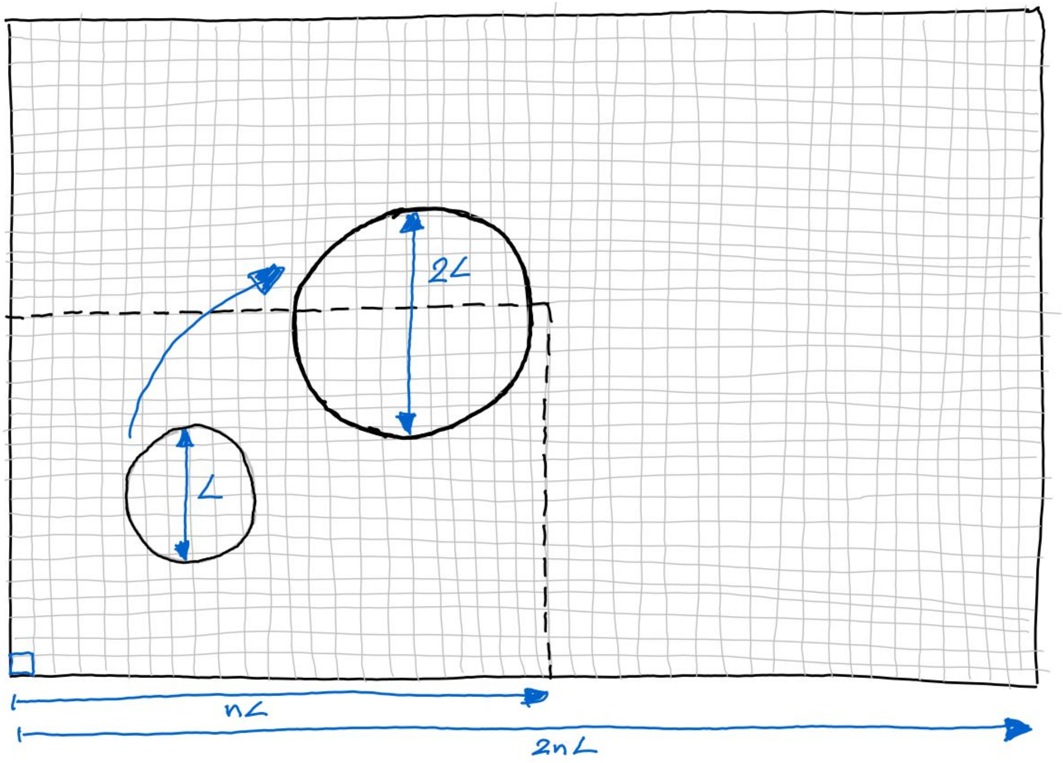
Double **L**, doubles resolution, ¼ errors

Scale viscosity to Reynolds number

$$\nu = UL/\mathrm{Re}$$

$$y^+ = \sqrt{0.026/2\mathrm{Re}^{1/7}} \Big/ \nu \approx O(1)$$

Set mapping scales (**U~1**, rotation)



Grid origin located at lower left corner

Simulation outputs

Compute **forces** (pressure, viscous)

Standard **flow metrics** (λ_2, Q-criterion, vorticity, azimuthal vorticity, kinetic energy, our own?)

Export to **vtk** (paraview) files

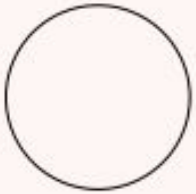
Example 1: 2D cylinder flow

2D simulation

Analytical distance function

No motion

Reynolds number, Re = 250



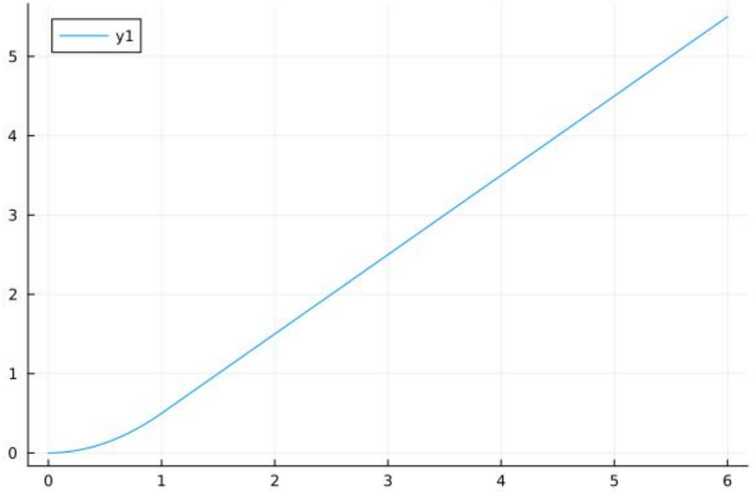
Example 2: accelerated disk

2D and 3D example (axisymmetric)

Axisymmetric problem

Re = 1000

Square acceleration profile



Specificities about GPU simulation

Backen agnostic:

- NVIDIA, AMD, Intel, Apple

Simulation must fit onto the GPU (memory issues)

Visualisation and file IO still occur on the CPU, transfer is expansive

But, 180x speed-up!

Ellipsoidal wing: 5 cycles
50’000’000 DoF
~2 hours on RTX 2080Ti

Example 2: accelerated disk

3D example on the GPU

Use axisymmetry for distance function

¼ of the domain for $tU/L<5$ (?)

Paraview output for 3D data

Accelerated disk: 6 convective time
113’250’000 DoF
~1 hours on RTX 2080Ti

Example 3: vertical axis wind turbine

2D simulation, can do spanwise periodic

Reynolds number Re = 1’000

Tip speed ratio λ = 1.4

Chord/radius ratio R = 3

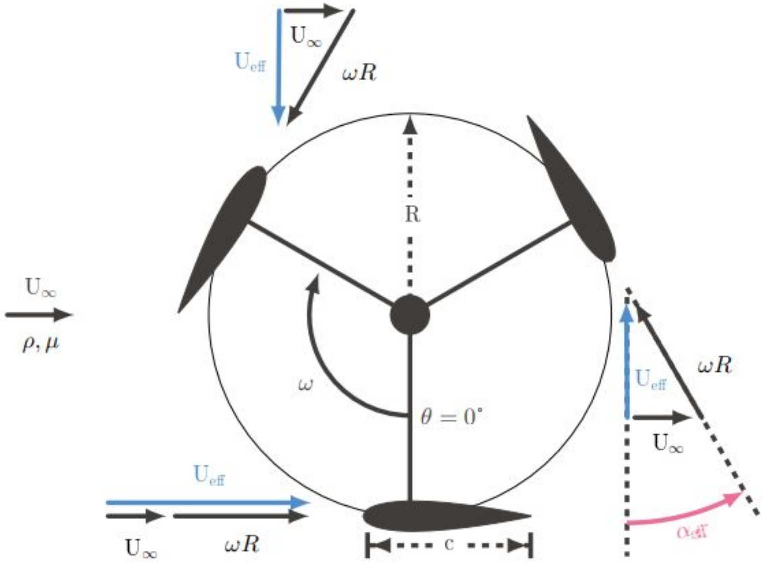


Figure 1.2: Vertical-axis wind turbine aerodynamics diagram. A fluid of density ρ and kinematic viscosity μ flows with a free stream velocity U_∞ that goes from left to right. The blade's velocity is equivalent to the rotational frequency ω times the turbine's radius R. The definitions of the blade's effective angle of attack α_{eff} and velocity U_{eff} are shown schematically.

Vertical axis wind turbine distance function is complex use **ParametricBody** !

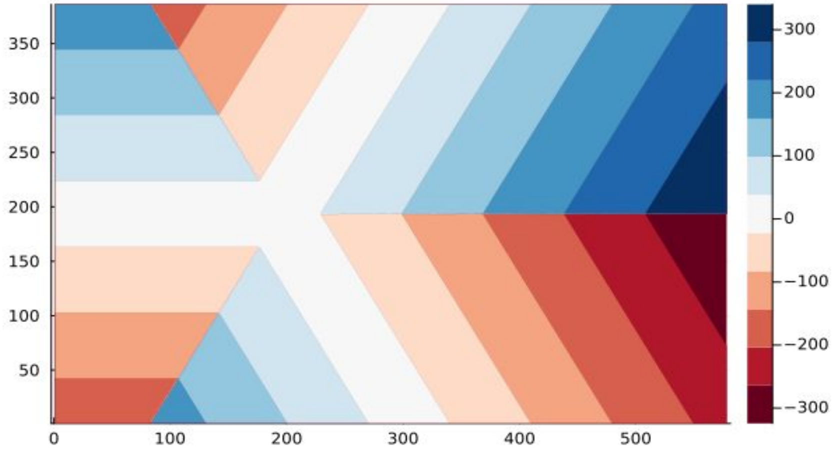
```
# naca 0012 airfoil with closed TE
NACA(s) = 0.6f0*(0.2969f0s-0.126f0s^2-0.3516f0s^4+0.2843f0s^6-0.1036f0s^8)
curve(s,t) = L*SA[(1-s)^2,NACA(1-s)]

# define the body
body = ParametricBody(curve,(0,1);map,T,mem)
```

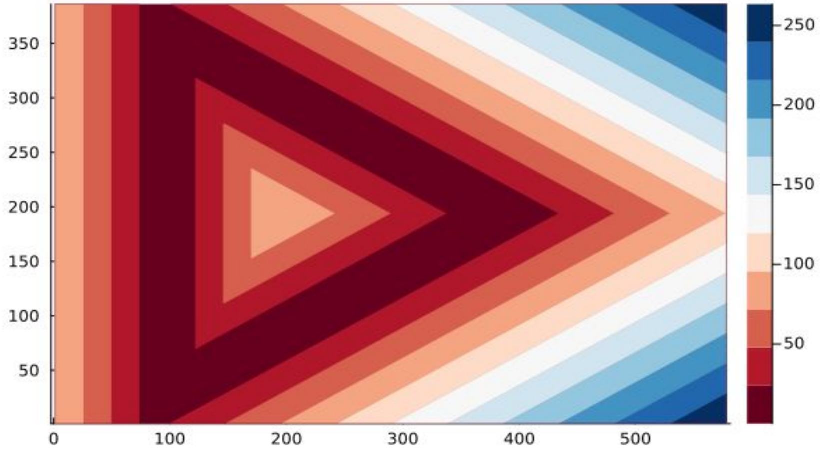
Step by step approach

1. First a NACA 0012 at an angle of attack
2. Single blade rotating
3. N blades rotating

Step by step mapping

```julia
function map(x₁,t)
    # Transform to rotating axis-centerted frame
    x₂ = Rot(ω*t)*(x₁.-0.5f0n*R*L)
    # Collapse to single-blade section
    s = floor(Int,θ(x₂)/2ϕ); x₃ = Rot(s*2ϕ+ϕ)*x₂
    # Move blade to origin and align with x-axis
    return SA[0.25f0L-x₃[2],abs(x₃[1]-R*L)]
end
```





Summary of WaterLily.jl simulation

1. Choose how to represent **body**
2. Determine the resolution **L** and domain size
3. Scale **viscosity**, **map**, etc.
4. Determine what **output** we need
5. **Run**…