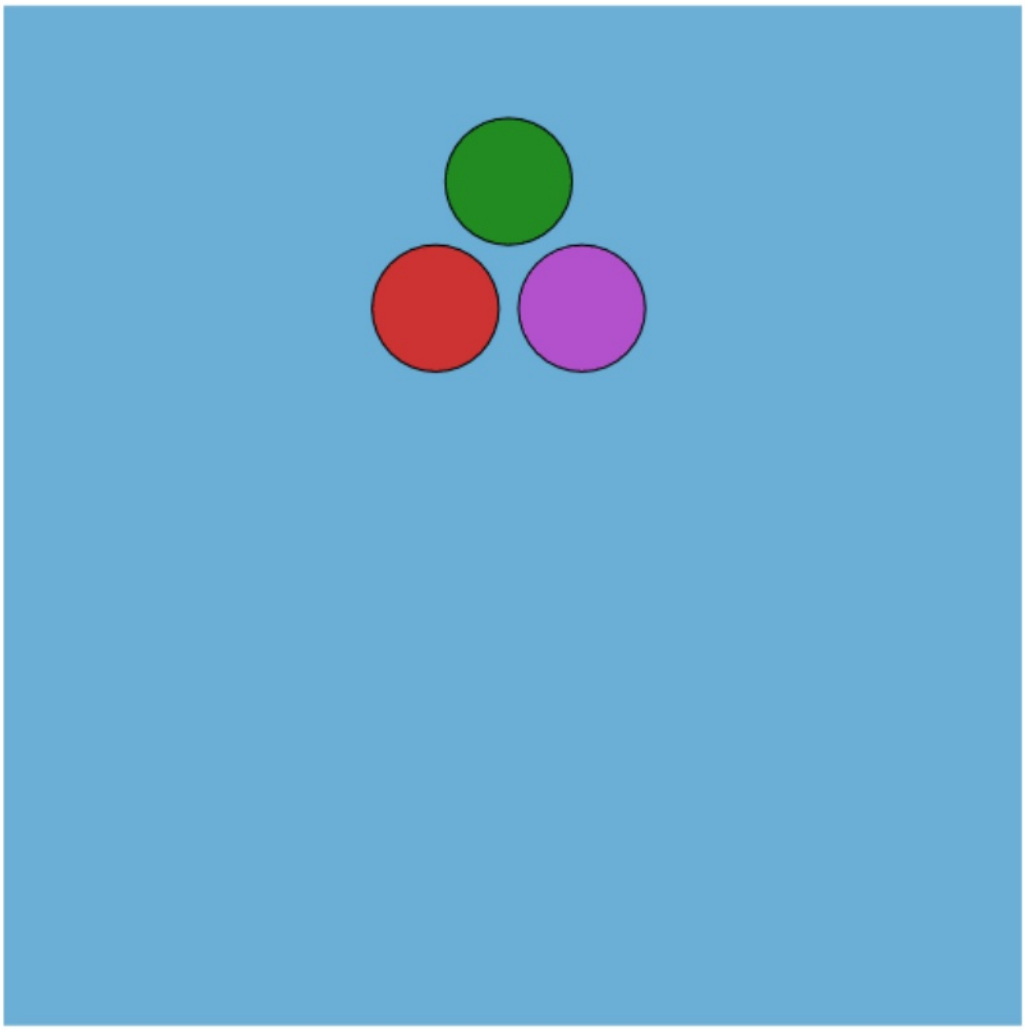
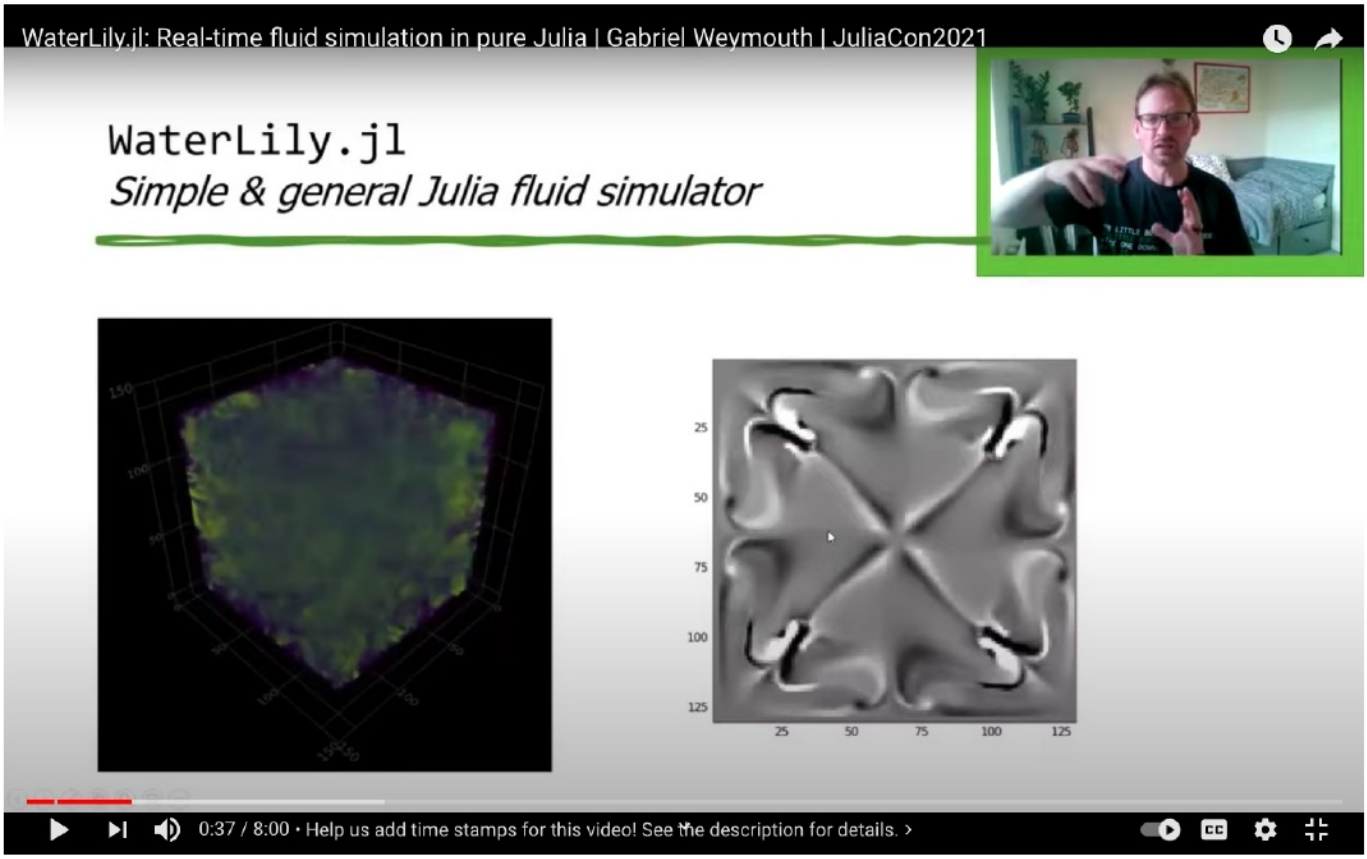
WaterLily.jl



概述

睡莲。jl是一个用纯茱莉亚编写的简单快速的流体模拟器。这是一个利用茱莉亚活跃的科学界来加速和增强流体模拟的实验项目。点击这里观看JuliaCon2021演讲:



方法/功能

睡莲。jl在笛卡尔网格上求解非定常不可压缩的2D或3D Navier-Stokes方程。压力泊松方程采用几何多重网格法求解。固体边界采用边界数据浸没法进行建模。求解器可以在串行CPU、多线程CPU或GPU后端上运行。

例子

用户可以设置边界条件，初始速度场，流体粘度(决定雷诺数)，并使用有符号距离函数浸入固体障碍物。这些例子和其他例子都可以在示例中找到。

绕圈流动

我们将模拟域的大小定义为nxm单元。圆的半径为m/8，圆心为(m/2,m/2)。流动边界条件为(U=1,0)，雷诺数为Re=U*半径/ν，其中ν(希腊语“nu”U+03BD，不是拉丁语小写“v”)为流体的运动粘度。

```
using WaterLily
function circle(n,m;Re=250,U=1)
    radius, center = m/8, m/2
    body = AutoBody((x,t)->√sum(abs2, x .- center) - radius)
    Simulation((n,m), (U,0), radius; ν=U*radius/Re, body)
end
```

倒数第二行用带符号的距离函数定义了圆的几何形状。AutoBody函数使用自动微分来自动推断其他几何参数。把圆的距离函数替换成其他的，现在你就有了围绕其他东西的流动…比如甜甜圈或茉莉亚的标志。最后，最后一行通过传入我们定义的参数来定义模拟。

现在我们可以创建一个模拟(第一行)并按时间向前运行(第三行)

```
circ = circle(3*2^6,2^7)
t_end = 10
sim_step!(circ,t_end)
```

注意，我们设置了n,m为2的幂的倍数，这在使用(非常快)多重网格求解器时很重要。我们现在可以访问和绘制任何我们喜欢的变量。例如，我们可以使用println(circ.flow.u[I])打印I::CartesianIndex处的速度，或者使用

```
using Plots
contour(circ.flow.p')
```

已经实现了一组流量度量函数，示例使用这些函数制作了如上所示的gif。

3D泰勒绿色漩涡

三维泰勒绿色漩涡演示了许多其他可用的模拟选项。首先，你可以通过传递一个向量函数uλ(i,xyz)来模拟一个非平凡的初始速度场，其中i∈(1,2,3)表示速度分量u☒，xyz =[x,y,z]是位置向量。

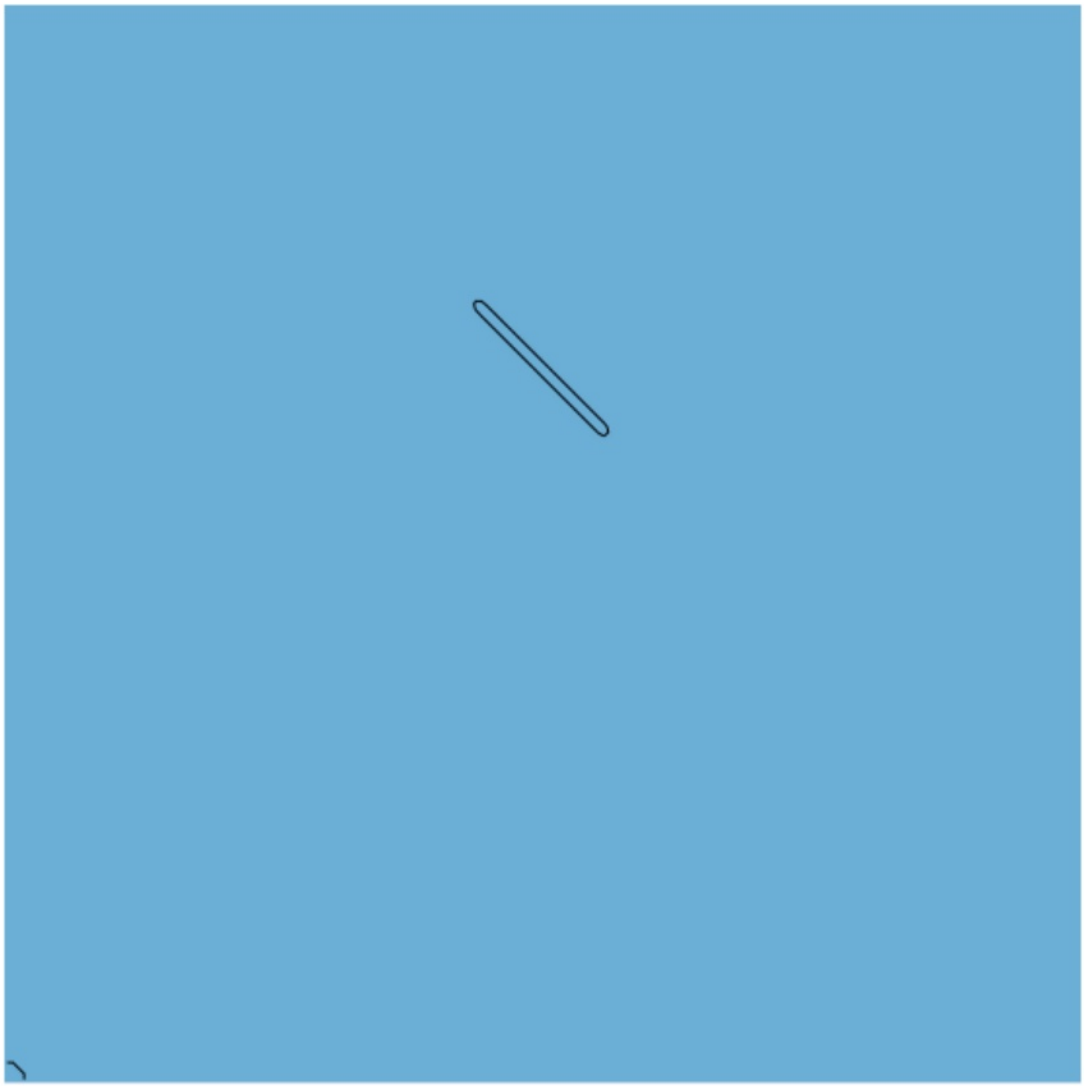
```
function TGV(; pow=6, Re=1e5, T=Float64, mem=Array)
    # Define vortex size, velocity, viscosity
    L = 2^pow; U = 1; ν = U*L/Re
    # Taylor-Green-Vortex initial velocity field
    function uλ(i,xyz)
        x,y,z = @. (xyz-1.5)*π/L                # scaled coordinates
        i==1 && return -U*sin(x)*cos(y)*cos(z) # u_x
        i==2 && return  U*cos(x)*sin(y)*cos(z) # u_y
        return 0.                               # u_z
    end
    # Initialize simulation
    return Simulation((L, L, L), (0, 0, 0), L; U, uλ, ν, T, mem)
end
```

这个例子还演示了浮点类型(T=Float64)和数组内存类型(mem= array)选项。例如，要在NVIDIA GPU上运行，我们只需要导入CUDA。jl库并初始化该设备上的模拟内存。

```
导入CUDA
@assert CUDA.functional ()
vortex = TGV(T=Float32,mem=CUDA.CuArray)
sim_step !(涡,1)
```

对于AMDGPU，请使用import AMDGPU和mem=AMDGPU. rocarray。请注意Julia 1.9是AMD gpu所必需的。

移动身体



在睡莲中，你可以通过向AutoBody传递一个坐标图来模拟移动的物体。

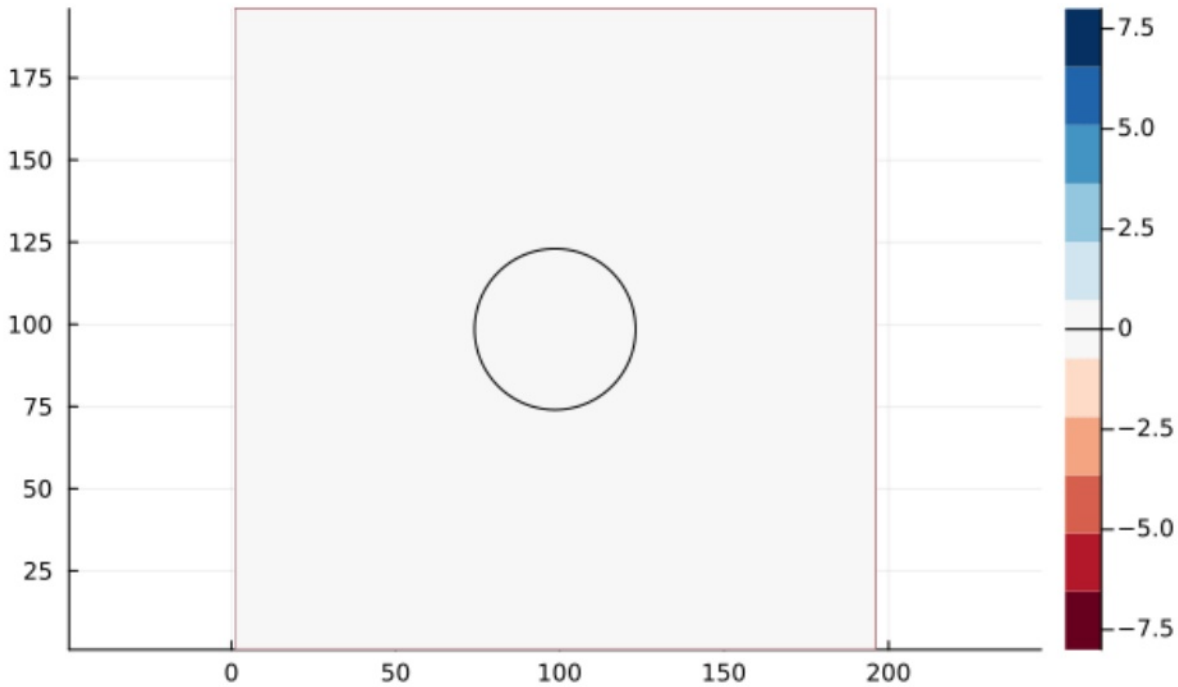
```
using StaticArrays
function hover(L=2^5;Re=250,U=1,amp=π/4,ϵ=0.5,thk=2ϵ+√2)
    # Line segment SDF
    function sdf(x,t)
        y = x .- SA[0,clamp(x[2],-L/2,L/2)]
        √sum(abs2,y)-thk/2
    end
```

```
    # Oscillating motion and rotation
    function map(x,t)
        α = amp*cos(t*U/L); R = SA[cos(α) sin(α); -sin(α) cos(α)]
        R * (x - SA[3L-L*sin(t*U/L),4L])
    end
    Simulation((6L,6L),(0,0),L;U,ν=U*L/Re,body=AutoBody(sdf,map),ϵ)
end
```

在这个例子中，sdf函数定义了一条从-L/2≤x[2]≤L/2的线段，其厚度为thk。为了使线段移动，我们定义了一个坐标变换函数map(x,t)。在这个例子中，坐标x在时间t=0时移动了(3L,4L)，这将线段的中心移动到这个点。然而，水平偏移在时间上是谐波变化的，在模拟过程中会将线段左右扫过。这个例子还使用旋转矩阵R = [cos(α) sin(α);-sin(α) cos(α)]其中角度α也是谐波变化的。综合的结果是一条细的扑动线，类似于悬停的昆虫翅膀的横截面。

这里需要注意的重要一点是使用StaticArrays来定义sdf和map。这加快了模拟的速度，因为它消除了每个网格单元和时间步的分配。

振荡流内部的圆圈



这个例子演示了一个圆上的二维振荡周期流。

```
function circle(n,m;Re=250,U=1)
    # define a circle at the domain center
    radius = m/8
    body = AutoBody((x,t)->√sum(abs2, x .- (n/2,m/2)) - radius)
```

```
    # define time-varying body force `g` and periodic direction `perdir`
    accelScale, timeScale = U^2/2radius, radius/U
    g(i,t) = i==1 ? -2accelScale*sin(t/timeScale) : 0
    Simulation((n,m), (U,0), radius; v=U*radius/Re, body, g, perdir=(1,))
end
```

g参数接受一个带有方向(i)和时间(t)参数的函数。这允许你创建一个随时间变化的空间均匀的体力。在这个例子中，该函数在“x”方向i=1上添加了一个正弦力，其他方向上没有任何作用力。

perdir参数是一个元组，指定了应该应用周期性边界条件的方向。任何数量的方向都可以被定义为周期性的，但是在这个例子中只使用了i=1方向，允许流在这个方向上自由加速。

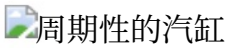
加速参考系

睡莲提供了利用时变边界条件建立速度场模拟的可能性。这可以用来模拟加速参考系中的流动。下面的例子演示了如何建立一个时变速度场的模拟。

```
using WaterLily
# define time-varying velocity boundary conditions
Ut(i,t::T;a0=0.5) where T = i==1 ? convert(T, a0*t) : zero(T)
# pass that to the function that creates the simulation
sim = Simulation((256,256), Ut, 32)
```

用Ut函数定义时变速度场。在这个例子中，“x”方向的速度被设置为a0*t，其中a0是参考系的加速度。然后用Ut函数作为第二个参数调用Simulation函数。然后，模拟将以时变的速度场运行。

周期和对流边界条件



除了标准的自由滑移(或反射)边界条件外，WaterLily还支持周期性边界条件。下面的例子演示了如何在“y”方向设置周期性边界条件的模拟。

```
using WaterLily,StaticArrays

# sdf an map for a moving circle in y-direction
function sdf(x,t)
    norm2(SA[x[1]-192,mod(x[2]-384,384)-192])-32
end
function map(x,t)
```

```
    x.-SA[0.,t/2]
end

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

# y-periodic boundary conditions
Simulation((512,384), (1,0), 32; body, perdir=(2,))
```

此外，可以将标志exitBC=true传递给Simulation函数以启用对流边界条件。这将在x方向应用一个1D对流出口(目前没有办法改变这一点)。exitBC标志默认设置为false。在这种情况下，边界条件被设置为构建模拟时指定的u_BC向量的相应值。

```
using WaterLily

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

# y-periodic boundary conditions
Simulation((512,384), u_BC=(1,0), L=32; body, exitBC=true)
```

写入VTK文件

下面的示例演示了如何使用WriteVTK包和WaterLily vtkwriter函数将仿真数据写入。pvd文件。最简单的writer可以实例化

```
using WaterLily,WriteVTK

# make a sim
sim = make_sim(...)

# make a writer
writer = vtkwriter("simple_writer")

# write the data
write!(writer,sim)

# don't forget to close the file
close(writer)
```

这将把速度和压力场写入一个名为simmple_writer.pvd的文件。vtkwriter函数也可以接受一个自定义属性的字典来写入文件。例如，要将body (sdf)和λ⊠字段写入文件，可以使用以下代码:

使用WaterLily，WriteVTK

```
# make a writer with some attributes, need to output to CPU array to save file (|>
Array)
velocity(a::Simulation) = a.flow.u |> Array;
pressure(a::Simulation) = a.flow.p |> Array;
_body(a::Simulation) = (measure_sdf!(a.flow.σ, a.body, WaterLily.time(a));
                                     a.flow.σ |> Array;)
lamda(a::Simulation) = (@inside a.flow.σ[I] = WaterLily.λ₂(I, a.flow.u);
                        a.flow.σ |> Array;)

# this maps field names to values in the file
custom_attrib = Dict(
    "Velocity" => velocity,
    "Pressure" => pressure,
    "Body" => _body,
    "Lambda" => lamda
)

# make the writer
writer = vtkWriter("advanced_writer"; attrib=custom_attrib)
...
close(writer)
```

传递给attrib(自定义属性)的函数必须遵循与本例中所示相同的结构，即给定一个模拟，返回一个n维(标量或向量)字段。vtkwriter函数会自动将数据写入一个。pvd文件，该文件可以由Paraview读取。vtkwriter函数的原型是:

```
# prototype vtk writer function
custom_vtk_function(a::Simulation) = ... |> Array
```

…应该替换为生成你想写入文件的字段的代码。管道到(CPU)数组是必要的，以确保数据被写入到GPU模拟文件之前被写入到CPU。

从VTK文件重新启动

这种能力对于从先前的状态重新启动模拟非常有用。ReadVTK包用于从.pvd文件中读取仿真数据。这个。pvd**必须**是用vtkwriter函数写的，并且**必须**至少包含速度场和压力场。下面的示例演示了如何使用ReadVTK包和WaterLily vtkreader函数从.pvd文件重新启动模拟

```
using WaterLily,ReadVTK
sim = make_sim(...)
# restart the simulation
writer = restart_sim!(sim; fname="file_restart.pvd")
```

```
# this acctually append the data to the file used to restart
write!(writer, sim)

# don't forget to close the file
close(writer)
```

在内部，该函数读取.pvd文件中的最后一个文件，并使用它来设置仿真中的速度和压力场。sim_time也被设置为保存在。pvd文件中的最后一个值。该函数还返回一个vtkwriter，它将把新数据附加到用于重启模拟的文件中。**注意**，为了这次重启工作，将被填充的sim必须与保存到文件中的sim相同，即相同的大小，相同的主体等。

多线程和GPU后端

睡莲使用内核抽象。jl到多线程在CPU和运行在GPU后端。我们的ParCFD摘要中记录了实现方法和加速比。总之，一个宏睡莲。@loop几乎用于代码库中的每个循环，这使用KernelAbstractactions为每个后端生成优化代码。这种加速在大型模拟中更为明显，我们在英特尔酷睿i7-10750H x6处理器上测试了高达23x的速度，在NVIDIA GeForce GTX 1650 Ti GPU卡上测试了182x的速度。

注意，多线程需要用——threads参数*启动*Julia，请参阅手册的多线程部分。如果您正在运行具有多个线程的Julia，内核抽象将检测到这一点并自动执行多线程循环。正如上面的Taylor-Green-Vortex示例，在GPU上运行需要初始化GPU上的模拟内存，并且需要注意将数据移回CPU以进行可视化。另一个非平凡的例子参见《jelly fish》。

最后，内核抽象确实会为每个循环分配一些CPU，但除了sim_step!是完全不分配的。这也是加速比随着模拟规模增大而提高的原因之一。

发展目标

- 使用GeometryBasics沉浸在由3D网格定义的障碍中。多cpu /
- GPU模拟。
- 使用流体体积或水平集添加自由表面物理。添加外部势
- 流域边界条件。

如果您有其他建议或希望提供帮助，请在github上提出问题。