FEniCS, part IV: Generating a model

M. M. Sussman

sussmanm@math.pitt.edu

Office Hours: 11:10AM-12:10PM, Thack 622

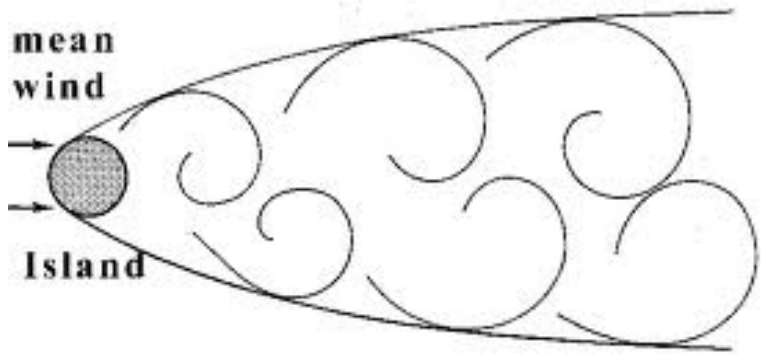
May 12 – June 19, 2014

Topics

Topics

von Kàrmàn vortex street

Fluid flow past an obstacle is not steady



Clouds

Discussed in FEniCS book

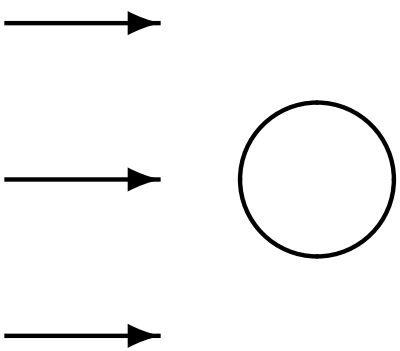
- Chapter 21 discusses Navier-Stokes equations with several problems and several solution methods.
- “Cylinder” problem
- Other sources of information, too

Background facts

- Self-generated transient behavior
- Simplest example: flow past a circular cylinder
- Plenty of experimental studies
- Reynolds numbers above a few hundred
- Strouhal number $\mathrm{St} = \frac{fL}{V}$ nearly 0.2

How to start?

- Choose a 2D channel
- At least 4 times wider than cylinder
- At least 4 times longer than wide
- Flow coming in on the left side
- Flow confined top and bottom
- Flow exiting on right



Navier-Stokes equations

- Navier-Stokes

$$\frac{\partial u}{\partial t} - \nu \nabla^2 u + (u \cdot \nabla) u + \nabla p = f$$

- Incompressibility

$$\nabla \cdot u = 0$$

- No time derivative of p!

Time discretization

- Use “lagged” backward Euler

$$\frac{u^{k+1} - u^k}{\Delta t} - \nu \nabla^2 u^{k+1} + (u^k \cdot \nabla) u^{k+1} + \nabla p^{k+1} = f^{k+1}$$

$$\nabla \cdot u^{k+1} = 0$$

- *Linear* at each time step
- Implicit
- Only first-order in Δt

“Fully implicit”

- Use backward Euler

$$\frac{u^{k+1} - u^k}{\Delta t} - \nu \nabla^2 u^{k+1} + (u^{k+1} \cdot \nabla) u^{k+1} + \nabla p^{k+1} = f^{k+1}$$
$$\nabla \cdot u^{k+1} = 0$$

- Nonlinear at each time step
- Need Jacobian for Newton solver
- Implicit
- Only first-order in Δt

Higher order methods in time

- Crank-Nicolson

$$\frac{u^{k+1}-u^k}{\Delta t}-\nu\left(\frac{\nabla^2 u^{k+1}+\nabla^2 u^k}{2}\right)$$
$$+\frac{(u^{k+1}\cdot\nabla)u^{k+1}+(u^k\cdot\nabla)u^k}{2}+\nabla p^k=\frac{f^{k+1}+f^k}{2}$$

- Along with

$$\nabla\cdot u^{k+1}=0$$

- p^{k+1} amounts to $p^{k+1/2}$
- Nonlinear at each time step
- Need Jacobian for Newton solver
- Implicit
- Second-order in Δt
- Other higher-order nonlinear methods are available

Boundary conditions

- No-slip on walls, including cylinder
- Inlet velocity?
- Exit condition?
- Pressure condition?

Inlet condition

- Can specify velocity
- Parabola
- Turbulent
- Constant
- Maybe specify pressure and not velocity

Exit condition

- Can specify exit velocity (consistent with mass conservation)
- Can take natural boundary condtion
- “Minimize upstream disturbance” condition
- “infinite” element

Pressure conditions

- No pressure condition is required, except eliminate constant
- Can eliminate constant in PETSc, but not FEniCS.
- Can specify pressure at exit or elsewhere
- Can penalize pressure by adding $\int \varepsilon pq\, dx$ to weak form

Initial condition

- Generate div-free init condition
  - Solve Stokes system, use as initial
- Start with zero, ramp boundary conditions up with time

Choosing time step

- Frequency comes from Strouhal number
- Need at least 10 steps per cycle
- Too large suppresses shedding
- Too large causes numerical instability
- Too small takes too long

Choosing Reynolds number (ν)

- Need Reynolds number > 150
- Too large causes numerical instability
- Too large can cause inability to solve nonlinear equation
- Too large requires tiny mesh

Weak form

For $(u, p) \in W = V \times P$

$$a((u, p), (v, q)) =$$

$$\int_\Omega \left(\frac{u \cdot v}{\Delta t} + (u^k \cdot \nabla u) \cdot v \, dx + \nu \nabla u \cdot \nabla v - p \nabla \cdot v + q \nabla \cdot u \right) dx$$

$$L(v, q) = \int_\Omega \frac{u^k \cdot v}{\Delta t} \, dx$$

Skew-symmetric convection

- If $\nabla \cdot u = 0$ with Dirichlet b.c.
- Convection term

$$C(w, u, v) = \int_\Omega (w \cdot \nabla u) v \, dx$$

- can be rewritten

$$C(w, u, v) = \frac{1}{2} \int_\Omega (w \cdot \nabla u) \cdot v) - (w \cdot \nabla v) \cdot u \, dx$$

Proof of skew-symmetric term

Since w is div-free,

$$(w \cdot \nabla u)v = (w_i \frac{\partial u_j}{\partial x_i})v_j = v_j \frac{\partial u_j w_i}{\partial x_i}$$

by the product rule and div-free w

$$(w \cdot \nabla v)u = (w_i \frac{\partial v_j}{\partial x_i})u_j = \frac{\partial u_j v_j w_i}{\partial x_i} - v_j \frac{\partial u_j w_i}{\partial x_i}$$

subtracting

$$(w \cdot \nabla u)v - (w \cdot \nabla v)u = \frac{\partial u_j v_j w_i}{\partial x_i} + 2v_j \frac{\partial u_j w_i}{\partial x_i}$$

Integrating and applying Dirichlet b.c.

$$\int_\Omega (w \cdot \nabla u)v - (w \cdot \nabla v)u \, dx = 2 \int_\Omega (w \cdot \nabla u) \cdot v \, dx$$

What good is it?

- Plugging $w = u$ and $v - u$ into the weak form yields an expression $E(u) = \int_\Omega fu\,dx$
- $\|u\|_2^2 \leq CE(u) \leq C\|f\|_2\|u\|_2$
- This yields a bound on $\|u\|_2$ for all time: stability
- Part of existence proof
- Warning: *not* Lyapunov-type stability

Topics

Example16: One implementation

```
from dolfin import *

# Constants related to the geometry
bmarg = 1.e-3 + DOLFIN_EPS
xmin = 0.0; xmax = 2.2
ymin = 0.0; ymax = 0.41
xcenter = 0.2; ycenter = 0.2; radius = 0.05

# generate coarse mesh, refine only near cylinder
domain = Rectangle(xmin,ymin,xmax,ymax) \
         - Circle(xcenter,ycenter,radius,12)
mesh = Mesh(domain, 22)
```

Example16: One implementation

```
from dolfin import *

# Constants related to the geometry
bmarg = 1.e-3 + DOLFIN_EPS
xmin = 0.0; xmax = 2.2
ymin = 0.0; ymax = 0.41
xcenter = 0.2; ycenter = 0.2; radius = 0.05

# generate coarse mesh, refine only near cylinder
domain = Rectangle(xmin,ymin,xmax,ymax) \
         - Circle(xcenter,ycenter,radius,12)
mesh = Mesh(domain, 22)

# refine mesh TWICE around the cylinder
for refinements in [0,1]:
    cell_markers = CellFunction("bool", mesh)
    cell_markers.set_all(False)
    for cell in cells(mesh):
        if (xcenter/2. < p[0] < (xmax - xcenter)/2.) and \
           (ycenter/4. < p[1] < ymax - ycenter/4.) :
            cell_markers[cell] = True
    mesh = refine(mesh, cell_markers)
plot(mesh)
```

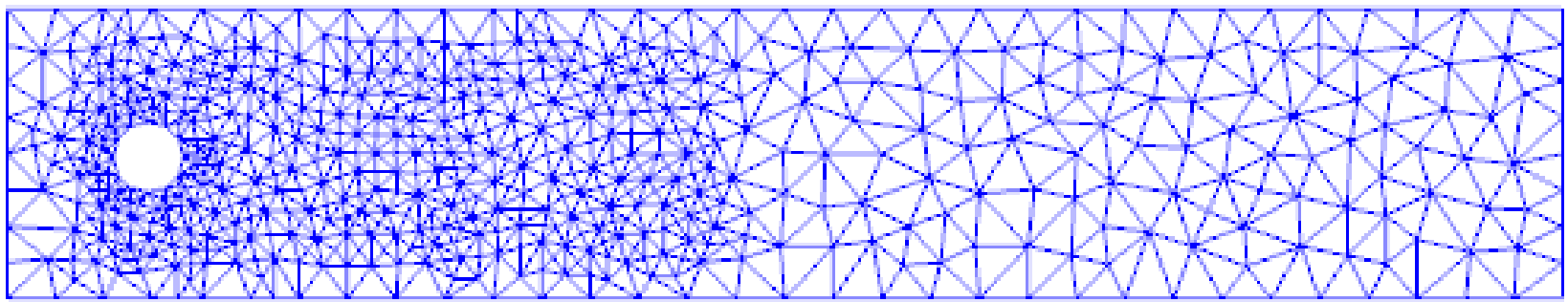
Example16: One implementation

```
from dolfin import *

# Constants related to the geometry
bmarg = 1.e-3 + DOLFIN_EPS
xmin = 0.0; xmax = 2.2
ymin = 0.0; ymax = 0.41
xcenter = 0.2; ycenter = 0.2; radius = 0.05

# generate coarse mesh, refine only near cylinder
domain = Rectangle(xmin,ymin,xmax,ymax) \
         - Circle(xcenter,ycenter,radius,12)
mesh = Mesh(domain, 22)

# refine mesh TWICE around the cylinder
for refinements in [0,1]:
    cell_markers = CellFunction("bool", mesh)
    cell_markers.set_all(False)
    for cell in cells(mesh):
        if (xcenter/2. < p[0] < (xmax - xcenter)/2.) and \
           (ycenter/4. < p[1] < ymax - ycenter/4.) :
            cell_markers[cell] = True
    mesh = refine(mesh, cell_markers)
plot(mesh)

# timestepping
dt = .0125
endTime = 10. - 1.e-5
```

Example16: One implementation

```
from dolfin import *

# Constants related to the geometry
bmarg = 1.e-3 + DOLFIN_EPS
xmin = 0.0; xmax = 2.2
ymin = 0.0; ymax = 0.41
xcenter = 0.2; ycenter = 0.2; radius = 0.05

# generate coarse mesh, refine only near cylinder
domain = Rectangle(xmin,ymin,xmax,ymax) \
         - Circle(xcenter,ycenter,radius,12)
mesh = Mesh(domain, 22)

# refine mesh TWICE around the cylinder
for refinements in [0,1]:
    cell_markers = CellFunction("bool", mesh)
    cell_markers.set_all(False)
    for cell in cells(mesh):
        if (xcenter/2. < p[0] < (xmax - xcenter)/2.) and \
           (ycenter/4. < p[1] < ymax - ycenter/4.) :
            cell_markers[cell] = True
    mesh = refine(mesh, cell_markers)
plot(mesh)

# timestepping
dt = .0125
endTime = 10. - 1.e-5

# kinematic viscosity
nu = .0005
```

Mesh



Mesh compromises

- Fine enough to see the solution
- Coarse enough to run quickly
- Default fineness on circle circumference too small to refine twice

Boundary conditions

```
# boundary conditions using a mesh function
boundaries = MeshFunction("size_t", mesh, mesh.topology().dim()-1)
```

Boundary conditions

```
# boundary conditions using a mesh function
boundaries = MeshFunction("size_t", mesh, mesh.topology().dim()-1)

# Inflow boundary
class InflowBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and x[0] < xmin + bmarg
```

Boundary conditions

```
# boundary conditions using a mesh function
boundaries = MeshFunction("size_t", mesh, mesh.topology().dim()-1)

# Inflow boundary
class InflowBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and x[0] < xmin + bmarg

# No-slip boundary
class NoslipBoundary(SubDomain):
    def inside(self, x, on_boundary):
        dx = x[0] - xcenter
        dy = x[1] - ycenter
        r = sqrt(dx*dx + dy*dy)
        return on_boundary and \
               (x[1] < ymin + bmarg or x[1] > ymax - bmarg or \
                r < radius + bmarg)
```

Boundary conditions

```
# boundary conditions using a mesh function
boundaries = MeshFunction("size_t", mesh, mesh.topology().dim()-1)

# Inflow boundary
class InflowBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and x[0] < xmin + bmarg

# No-slip boundary
class NoslipBoundary(SubDomain):
    def inside(self, x, on_boundary):
        dx = x[0] - xcenter
        dy = x[1] - ycenter
        r = sqrt(dx*dx + dy*dy)
        return on_boundary and \
               (x[1] < ymin + bmarg or x[1] > ymax - bmarg or \
                r < radius + bmarg)

# Outflow boundary
class OutflowBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and x[0] > xmax - bmarg
```

Define spaces, apply b.c.

```
# Define function spaces
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V * Q
```

Define spaces, apply b.c.

```
# Define function spaces
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V * Q

# no-slip velocity b.c.
noslipBoundary = NoslipBoundary()
g0 = Constant( (0.,0.) )
bc0 = DirichletBC(W.sub(0), g0, noslipBoundary)
```

Define spaces, apply b.c.

```
# Define function spaces
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V * Q

# no-slip velocity b.c.
noslipBoundary = NoslipBoundary()
g0 = Constant( (0.,0.) )
bc0 = DirichletBC(W.sub(0), g0, noslipBoundary)

# inlet velocity b.c.
inflowBoundary = InflowBoundary()
g1 = Expression( ("4.*Um*(x[1]*(ymax-x[1]))/(ymax*ymax)" , "0.0"),
bc1 = DirichletBC(W.sub(0), g1, inflowBoundary)
```

Define spaces, apply b.c.

```
# Define function spaces
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V * Q

# no-slip velocity b.c.
noslipBoundary = NoslipBoundary()
g0 = Constant( (0.,0.) )
bc0 = DirichletBC(W.sub(0), g0, noslipBoundary)

# inlet velocity b.c.
inflowBoundary = InflowBoundary()
g1 = Expression( ("4.*Um*(x[1]*(ymax-x[1]))/(ymax*ymax)" , "0.0"),
bc1 = DirichletBC(W.sub(0), g1, inflowBoundary)

# outflow pressure b.c.
outflowBoundary = OutflowBoundary()
g2 = Constant(0.)
bc2 = DirichletBC(W.sub(1), g2, outflowBoundary)
```

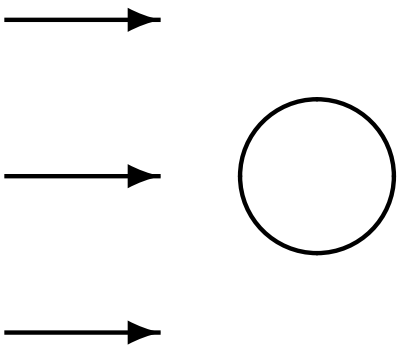
Define spaces, apply b.c.

```
# Define function spaces
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V * Q

# no-slip velocity b.c.
noslipBoundary = NoslipBoundary()
g0 = Constant( (0.,0.) )
bc0 = DirichletBC(W.sub(0), g0, noslipBoundary)

# inlet velocity b.c.
inflowBoundary = InflowBoundary()
g1 = Expression( ("4.*Um*(x[1]*(ymax-x[1]))/(ymax*ymax)" , "0.0"),
bc1 = DirichletBC(W.sub(0), g1, inflowBoundary)

# outflow pressure b.c.
outflowBoundary = OutflowBoundary()
g2 = Constant(0.)
bc2 = DirichletBC(W.sub(1), g2, outflowBoundary)

# outflow velocity b.c., same as inlet
bc3 = DirichletBC(W.sub(0), g1, outflowBoundary)
```

Define spaces, apply b.c.

```
# Define function spaces
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V * Q

# no-slip velocity b.c.
noslipBoundary = NoslipBoundary()
g0 = Constant( (0.,0.) )
bc0 = DirichletBC(W.sub(0), g0, noslipBoundary)

# inlet velocity b.c.
inflowBoundary = InflowBoundary()
g1 = Expression( ("4.*Um*(x[1]*(ymax-x[1]))/(ymax*ymax)" , "0.0"),
bc1 = DirichletBC(W.sub(0), g1, inflowBoundary)

# outflow pressure b.c.
outflowBoundary = OutflowBoundary()
g2 = Constant(0.)
bc2 = DirichletBC(W.sub(1), g2, outflowBoundary)

# outflow velocity b.c., same as inlet
bc3 = DirichletBC(W.sub(0), g1, outflowBoundary)

# collect b.c.
bcs = [bc0, bc1, bc2, bc3]
```

Outflow boundary



- Dirichlet velocity at outflow for simplicity
- Parabolic shape because Reynolds number is low
- Dirichlet pressure at whole outflow for convenience

Weak forms

```
# functions
(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)
w = Function(W)
u0 = Function(V)
uplot = Function(V)  # used for plotting in loop so get only 1 frame
```

Weak forms

```
# functions
(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)
w = Function(W)
u0 = Function(V)
uplot = Function(V)  # used for plotting in loop so get only 1 frame

# weak form Stokes equation
Stokes = (inner(grad(u), grad(v)) - div(v)*p + q*div(u))*dx
f = Constant((0., 0.))
LStokes = inner(f, v)*dx
```

Weak forms

```
# functions
(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)
w = Function(W)
u0 = Function(V)
uplot = Function(V)  # used for plotting in loop so get only 1 frame

# weak form Stokes equation
Stokes = (inner(grad(u), grad(v)) - div(v)*p + q*div(u))*dx
f = Constant((0., 0.))
LStokes = inner(f, v)*dx

# weak form NSE
LNSE = inner(u0,v)*dx
# skew-symmetric convection term
NSE = (inner(u,v) + dt*(.5*inner(grad(u)*u0,v) - .5*inner(grad(v)*u0,u)\
                      + nu*inner(grad(u),grad(v)) - div(v)*p) + q*div(u) )*dx
```

Weak forms

```
# functions
(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)
w = Function(W)
u0 = Function(V)
uplot = Function(V)  # used for plotting in loop so get only 1 frame

# weak form Stokes equation
Stokes = (inner(grad(u), grad(v)) - div(v)*p + q*div(u))*dx
f = Constant((0., 0.))
LStokes = inner(f, v)*dx

# weak form NSE
LNSE = inner(u0,v)*dx
# skew-symmetric convection term
NSE = (inner(u,v) + dt*(.5*inner(grad(u)*u0,v) - .5*inner(grad(v)*u0,u)\
                        + nu*inner(grad(u),grad(v)) - div(v)*p) + q*div(u) )*dx
```

Why use skew-symmetric convection?

Solve for initial velocity

```
# initial condition comes from Stokes equation
solve(Stokes == LStokes, w, bcs,solver_parameters=dict(linear_solver="lu"))
```

Solve for initial velocity

```
# initial condition comes from Stokes equation
solve(Stokes == LStokes, w, bcs,solver_parameters=dict(linear_solver="lu"))

# Split the mixed solution using deepcopy
(u, p) = w.split(True)
```

Solve for initial velocity

```
# initial condition comes from Stokes equation
solve(Stokes == LStokes, w, bcs,solver_parameters=dict(linear_solver="lu"))

# Split the mixed solution using deepcopy
(u, p) = w.split(True)

plot(u,title="Initial velocity")
plot(p,"Initial pressure")
```

Solve for initial velocity

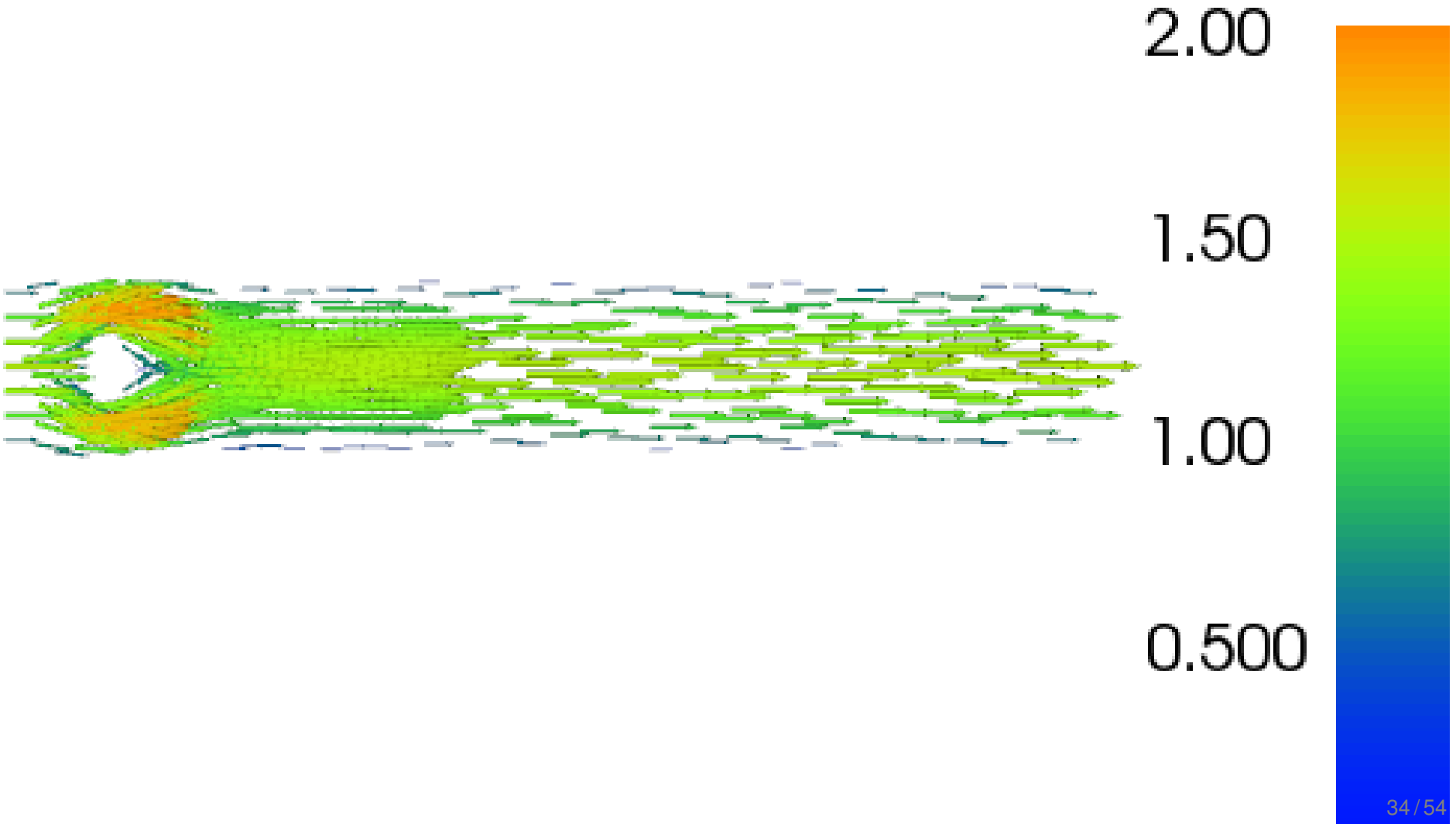
```
# initial condition comes from Stokes equation
solve(Stokes == LStokes, w, bcs,solver_parameters=dict(linear_solver="lu"))

# Split the mixed solution using deepcopy
(u, p) = w.split(True)

plot(u,title="Initial velocity")
plot(p,"Initial pressure")
```

`lu` is default, but I might want to change it.

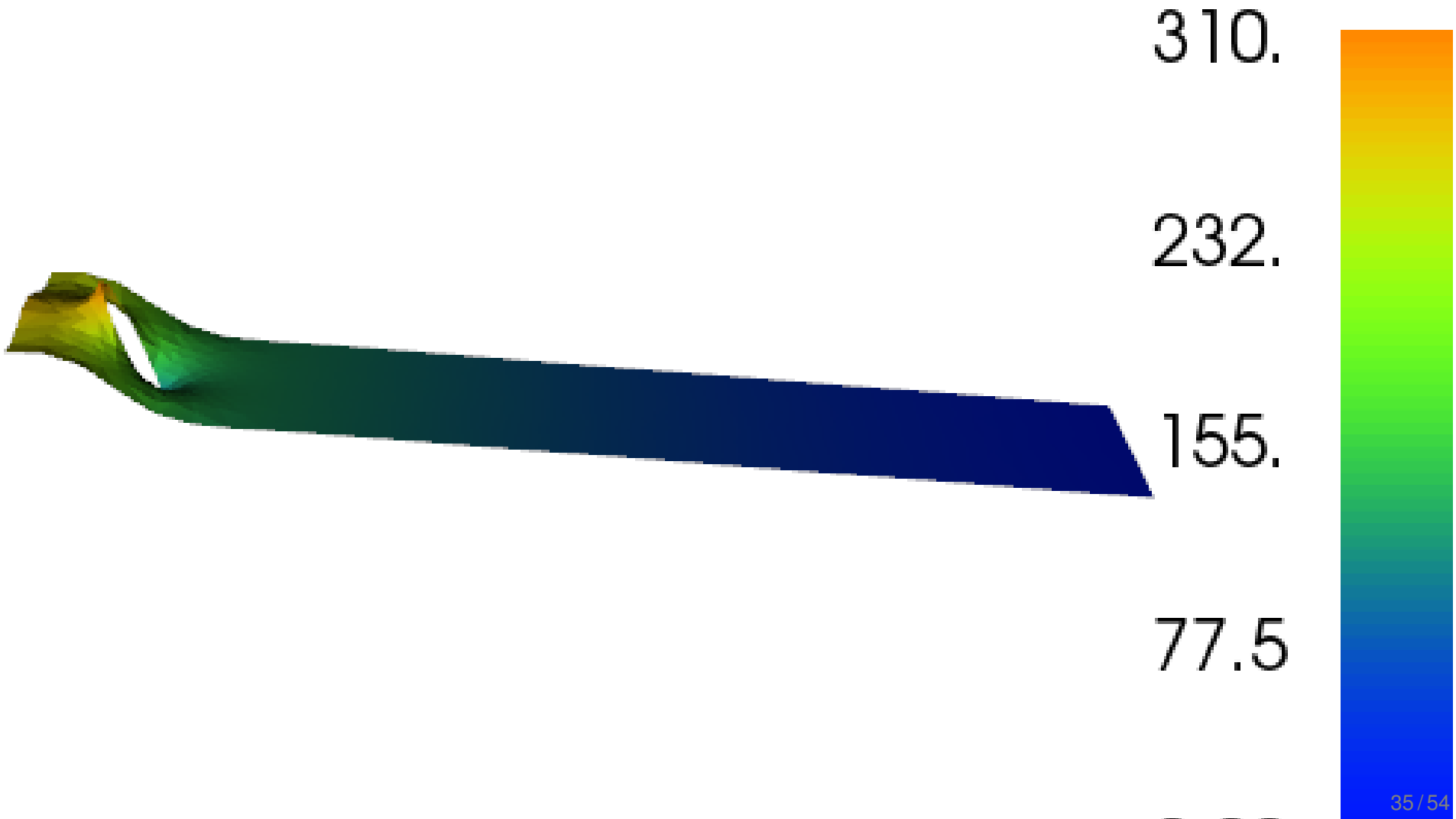
Debugging

- Examine this solution closely!
- Check that b.c. are satisfied
- Check solution reasonable
- Make sure no error messages were printed

Stokes velocity solution



Stokes pressure solution



Time stepping

```
while t < endTime * (1. + 1.e-10):
    u0.assign(u)
    t += dt
    step += 1
```

Time stepping

```
while t < endTime * (1. + 1.e-10):
    u0.assign(u)
    t += dt
    step += 1
    solve(NSE == LNSE, w, bcs, \
       solver_parameters=dict(linear_solver="lu"))
```

Time stepping

```
while t < endTime * (1. + 1.e-10):
    u0.assign(u)
    t += dt
    step += 1
    solve(NSE == LNSE, w, bcs, \
       solver_parameters=dict(linear_solver="lu"))

    # split vel, pres out of w (deepcopy)
    (u, p) = w.split(True)
```

Time stepping

```
while t < endTime * (1. + 1.e-10):
    u0.assign(u)
    t += dt
    step += 1
    solve(NSE == LNSE, w, bcs, \
       solver_parameters=dict(linear_solver="lu"))

    # split vel, pres out of w (deepcopy)
    (u, p) = w.split(True)

    # interactive plot
    """
    If plot(u), then get a new frame each time step.
    If use a single variable-as here-get only one frame for all
    time steps.
    """
    uplot.assign(u)
    plot(uplot,title="Velocity",rescale=False)
```

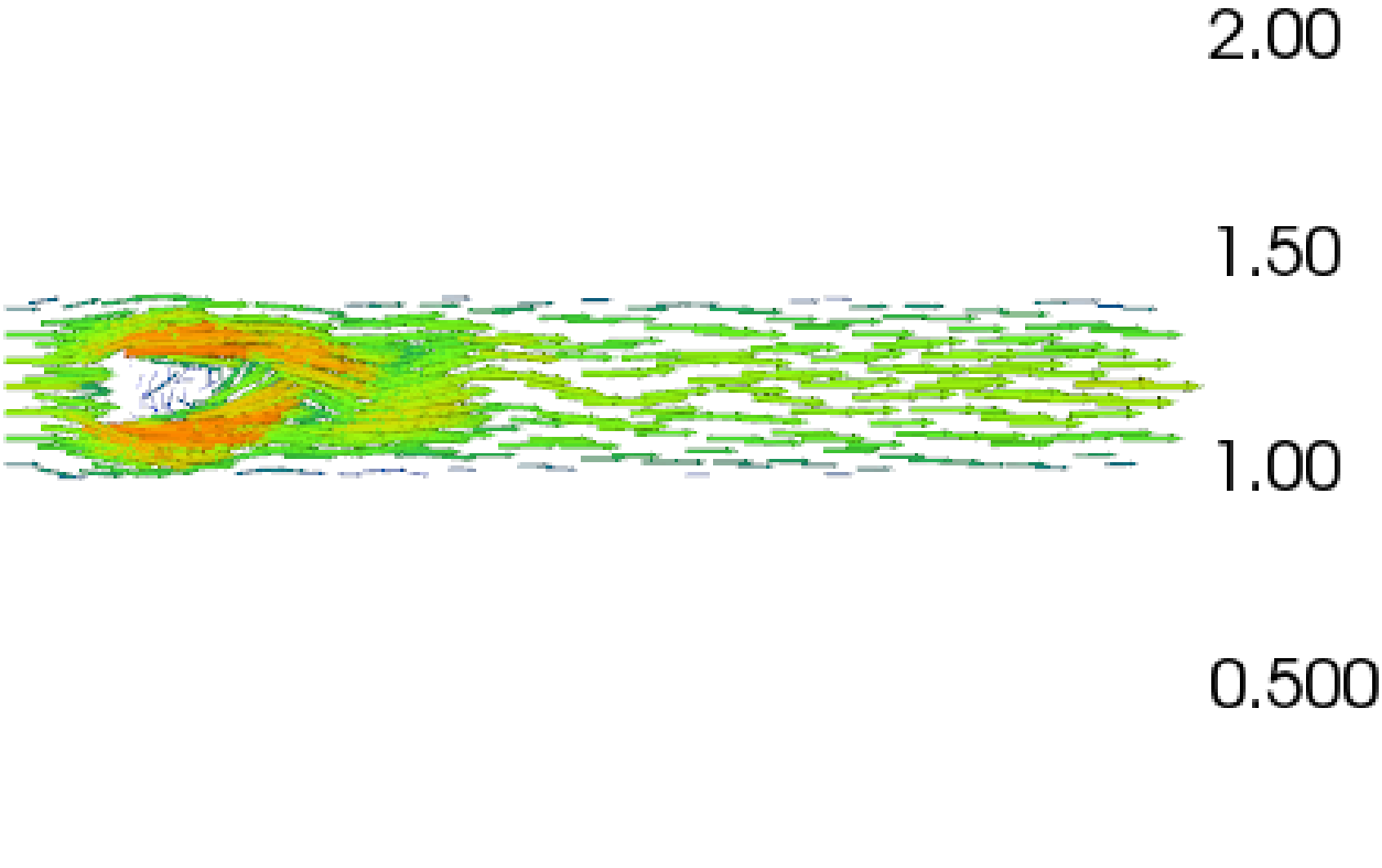
Time stepping

```
while t < endTime * (1. + 1.e-10):
    u0.assign(u)
    t += dt
    step += 1
    solve(NSE == LNSE, w, bcs, \
       solver_parameters=dict(linear_solver="lu"))

    # split vel, pres out of w (deepcopy)
    (u, p) = w.split(True)

    # interactive plot
    """
    If plot(u), then get a new frame each time step.
    If use a single variable-as here-get only one frame for all
    time steps.
    """
    uplot.assign(u)
    plot(uplot,title="Velocity",rescale=False)
```

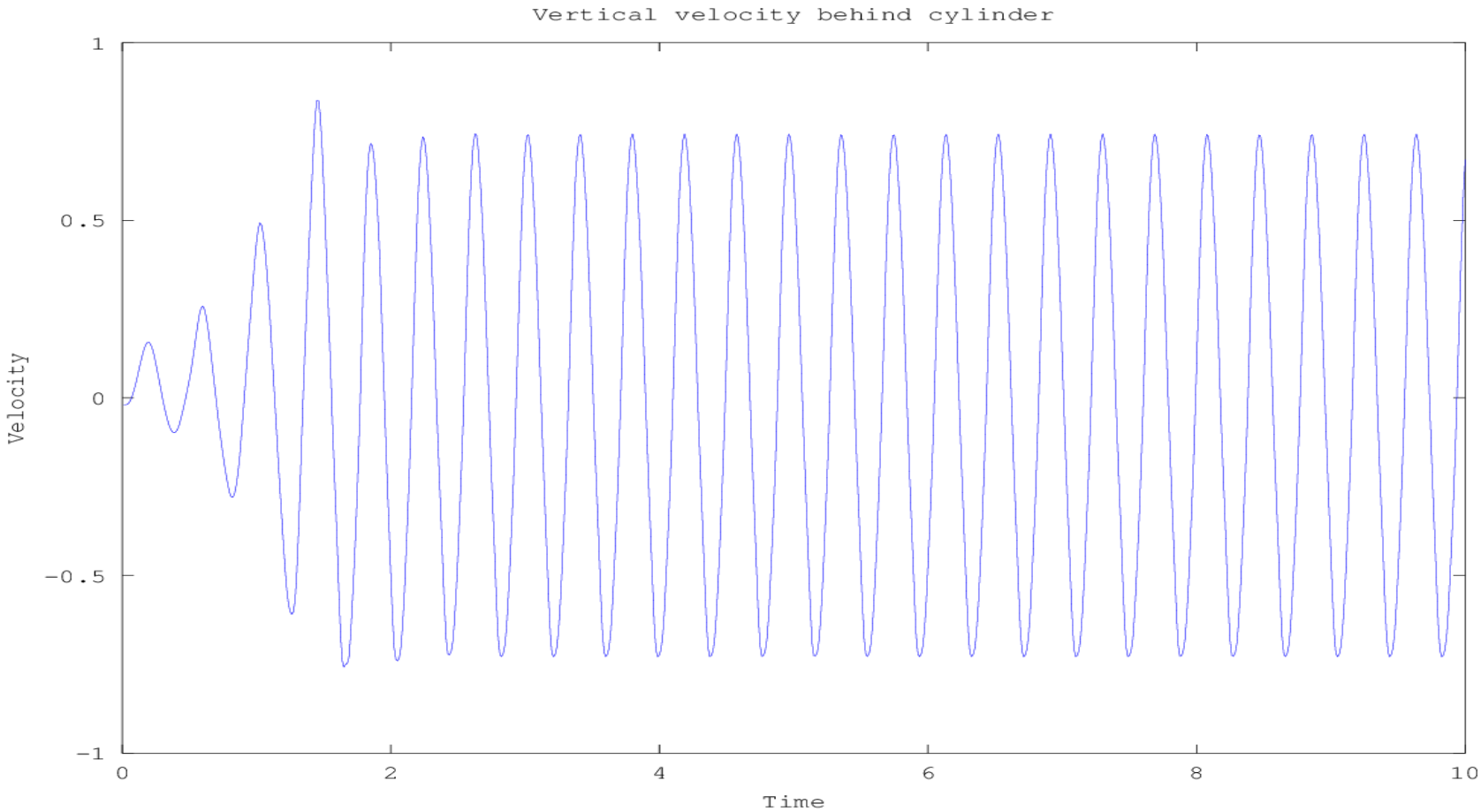
Time stepping

```
while t < endTime * (1. + 1.e-10):
    u0.assign(u)
    t += dt
    step += 1
    solve(NSE == LNSE, w, bcs, \
       solver_parameters=dict(linear_solver="lu"))

    # split vel, pres out of w (deepcopy)
    (u, p) = w.split(True)

    # interactive plot
    """
    If plot(u), then get a new frame each time step.
    If use a single variable-as here-get only one frame for all
    time steps.
    """
    uplot.assign(u)
    plot(uplot,title="Velocity",rescale=False)

    # print a little information each step
    print "t=%f, max abs u=%e, max p=%e, v(.5,0)=%e" \
      % (t, max(abs(u.vector().array())), max(p.vector().array()), \
           u([.35,0.2])[1] )
```

NSE velocity solution



Transient velocity point



Restarting

- Before starting transient

```
# startup if True, restarting if False
if True:
    step = 0
    t=0
else:
    # MUST set appropriate step number
    step = -1
    # generate name and read from file
    soln_file = File("restart" + "%05d"%step + ".xml")
    soln_file >> w
    # make new endTime, assuming constant dt in previous runs
    t = step*dt
    endTime += t
```

- At very end, write velocity to a file

```
soln_file = File("restart" + "%05d"%step + ".xml")
soln_file << w
```

Debugging restarts

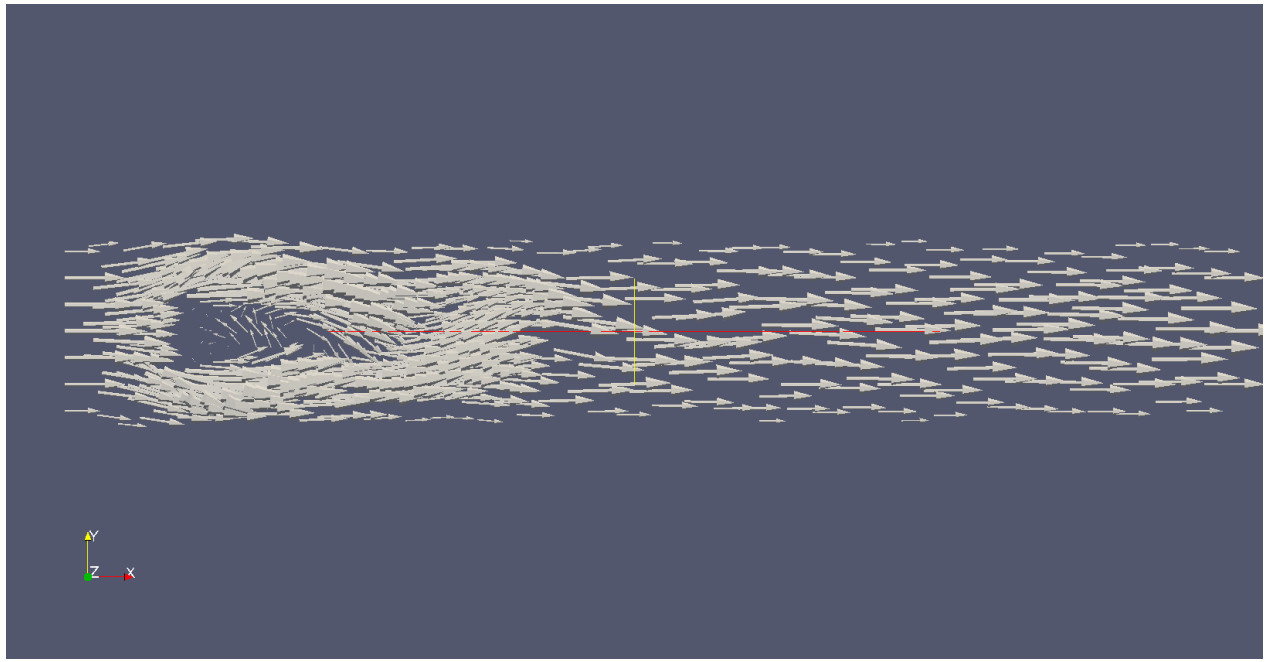
- If want to rely on restarting, *must* confirm it works!
- First, run for 2 steps, write file
- Second, run for 1 step, write file
- Third, run 1 step, restarting from second file, write file
- Compare first and third files.
- Use **`diff`**

What happens if ...

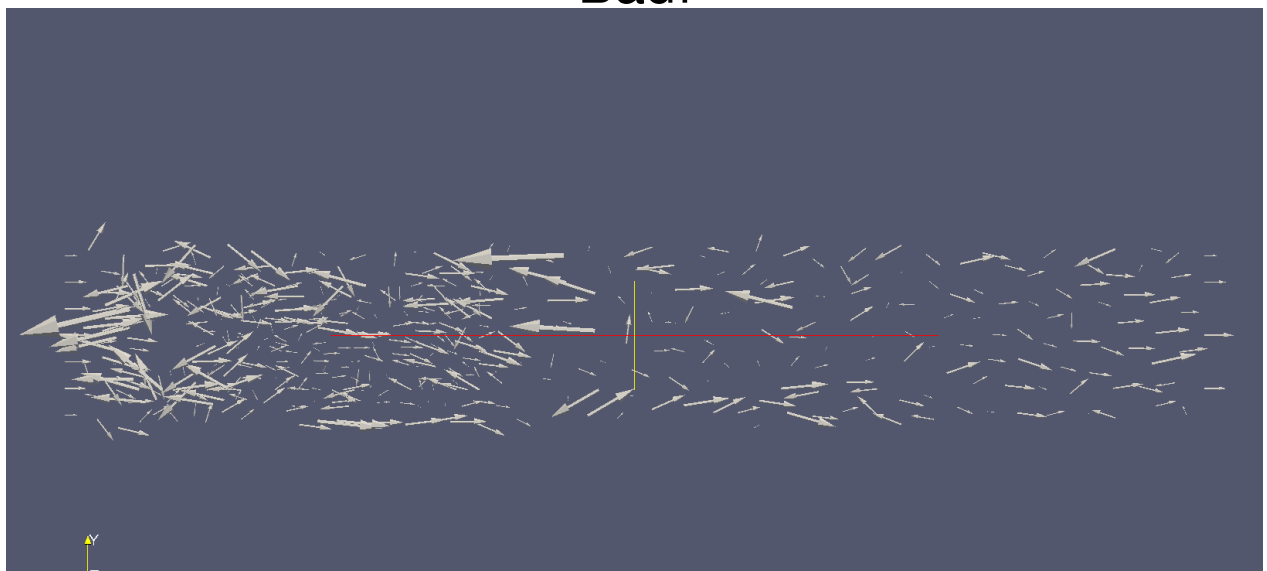
- Too large a timestep?
  - Solution blows up
  - Solution becomes steady when should not
- ν too large: solution becomes steady
- ν too small: solution goes nuts or blows up

Goes nuts?

Good!



Bad!



Another starting approach

- Instead of starting from Stokes, start from zero
- Initial conditions: By defining **`w = Function(W)`**, **`w`** starts out with all zeros.
- C or C++ syntax:

  `t<1.0 ? exprYES : exprNO`

- Expression for inlet/exit velocities

```
g1 = Expression( \
("t<1.0 ? sin(pi*t/2.0) * 4.*Um*(x[1]*(ymax-x[1]))/(ymax*ymax) :"\
                      +   "4.*Um*(x[1]*(ymax-x[1]))/(ymax*ymax)" \
                         , "0.0"), Um=1.5, ymax=ymax, t=t)
```

- Experiment to see if **`t=1.0`** is a good choice
- Inside the timestep loop

  `g1.t = t`

Exercise 20 (10 points)

Example 16 presents a solution of the Navier-Stokes equations, starting from a solution of the Stokes equations. Modify it so that it starts from zero velocity and pressure, ramping up the boundary velocities until they agree with the existing boundary velocities at time t=1.0.

Exercise 21 (25 points)

Poiseuille flow is an exact solution of the Navier-Stokes equations in a 2D channel. The velocity is parabolic in y, zero at the upper and lower boundaries, and independent of x. Pressure varies linearly with x and is uniform in y.

Start your work from **`example16.py`**.

1. Generate a roughly uniform mesh in the following way:

```
domain = Rectangle(xmin,ymin,xmax,ymax)
mesh = Mesh(domain, 30)
```

2. Reduce the problem end time **`endTime = 0.5 - 1.e-5`**
3. Increase the kinematic viscosity **`nu = 0.01`**
4. Verify that the solution at the end is visually identical to the initial solution (both velocity and pressure). Send me these plots.
5. Change the problem to use conventional convection term instead of skew-symmetric convection, and verify the final and intial solutions are visually identical. Send me these plots.
6. Change the exit *velocity* boundary condition to a Neumann condition. Verify that the final and intial solutions are visually identical. Send me these plots.

Exercise 21 cont'd

7. Return to skew-symmetric convection. Verify that the final solution *is not the same* as the initial solution. Send me these plots.
8. Revisit the derivation of the skew-symmetric form of convection and establish that there are additional terms to include in the weak form to properly model Poiseuille flow. Send me your work
9. Include your extra terms in your code and confirm visually that the initial and final flow agree. Send me these plots.
10. Compute the L^2 norm of the difference between the computed solution and the correct parabolic velocity at the exit. This value should be very small. What value did you get?
11. Please send me the code you used, commented so that I can find the changes you made.

Chorin and Temam's projection method

- Designed to be very efficiently implemented
- Matrices are constant in time
- Stability: Courant number$= u\Delta t/\Delta x < 1$
- A. J. Chorin, Numerical solution of the Navier-Stokes equations, *Math. Comp.,* **22**(1968) 745-762.
  `http://math.berkeley.edu/~chorin/chorin68.pdf`
- R. Temam, Sur l'approximation de la solution des équations de Navier-Stokes par la méthode des pas fractionnaires, *Arch. Rat. Mech. Anal.,* **32**(1969), 377-385

Timestepping algorithm

1. Compute $\tilde{u}$ by solving

$$\int_\Omega \left[\left(\frac{\tilde{u} - u^n}{\Delta t} + u^n \cdot \nabla u^n \right) v + \nu \nabla \tilde{u} \cdot \nabla v \right] dx = \int_\Omega f^{n+1} v \, dx$$

2. Compute p^{n+1} by solving

$$\int_\Omega \nabla p^{n+1} \nabla q \, dx = - \int_\Omega \frac{\nabla \cdot \tilde{u} q}{\Delta t}$$

3. and get u^{n+1} by solving

$$\int_\Omega u^{n+1} v \, dx = \int_\Omega \left(\tilde{u} - \Delta t \nabla p^{n+1} \right) v \, dx$$

4. In each of the above, boundary conditions are applied as necessary.

example17.py

```
...  same mesh as  example16.py  ...
# timestepping
dt = .00125  # 1/10 of example16.py
endTime = 3. - 1.e-5

# kinematic viscosity
nu = .001  # 2X larger than example16.py
...  same boundary conditions as  example16.py  ...
...  same Stokes solution as  example16.py  ...
# Timestepping method starts here
```

example17.py

```
...  same mesh as  example16.py  ...
# timestepping
dt = .00125  # 1/10 of example16.py
endTime = 3. - 1.e-5

# kinematic viscosity
nu = .001  # 2X larger than example16.py
...  same boundary conditions as  example16.py  ...
...  same Stokes solution as  example16.py  ...
# Timestepping method starts here
```

example17.py

```
... same mesh as example16.py ...
# timestepping
dt = .00125  # 1/10 of example16.py
endTime = 3. - 1.e-5

# kinematic viscosity
nu = .001  # 2X larger than example16.py
... same boundary conditions as example16.py ...
... same Stokes solution as example16.py ...
# Timestepping method starts here
step = 0
t=0

u0 = Function(V)
u0.assign(uinit)
utilde = Function(V)
u1 = Function(V)
p1 = Function(Q)

# Define test and trial functions
v = TestFunction(V)
q = TestFunction(Q)
u = TrialFunction(V)
p = TrialFunction(Q)
```

`example17.py` cont'd

```
# STEP 1 (u will be utilde)
F1 = (1./dt)*inner(v, u - u0)*dx + inner(v, grad(u0)*u0)*dx \
        + nu*inner(grad(u), grad(v))*dx - inner(f, v)*dx
```

example17.py cont'd

```
# STEP 1 (u will be utilde)
F1 = (1./dt)*inner(v, u - u0)*dx + inner(v, grad(u0)*u0)*dx \
        + nu*inner(grad(u), grad(v))*dx - inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)
```

example17.py cont'd

```
# STEP 1 (u will be utilde)
F1 = (1./dt)*inner(v, u - u0)*dx + inner(v, grad(u0)*u0)*dx \
        + nu*inner(grad(u), grad(v))*dx - inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# STEP 2 (p)
a2 = inner(grad(q), grad(p))*dx
L2 = -(1./dt)*q*div(utilde)*dx
```

example17.py cont'd

```
# STEP 1 (u will be utilde)
F1 = (1./dt)*inner(v, u - u0)*dx + inner(v, grad(u0)*u0)*dx \
        + nu*inner(grad(u), grad(v))*dx - inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# STEP 2 (p)
a2 = inner(grad(q), grad(p))*dx
L2 = -(1./dt)*q*div(utilde)*dx

# STEP 3 (Velocity update)
a3 = inner(v, u)*dx
L3 = inner(v, utilde)*dx - dt*inner(v, grad(p1))*dx
```

example17.py cont'd

```
# STEP 1 (u will be utilde)
F1 = (1./dt)*inner(v, u - u0)*dx + inner(v, grad(u0)*u0)*dx \
        + nu*inner(grad(u), grad(v))*dx - inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# STEP 2 (p)
a2 = inner(grad(q), grad(p))*dx
L2 = -(1./dt)*q*div(utilde)*dx

# STEP 3 (Velocity update)
a3 = inner(v, u)*dx
L3 = inner(v, utilde)*dx - dt*inner(v, grad(p1))*dx

# Assemble matrices
# cannot use symmetric b.c.!
A1 = assemble(a1)
bc0u.apply(A1)
bc1u.apply(A1)
bc3u.apply(A1)
```

example17.py cont'd

```
# STEP 1 (u will be utilde)
F1 = (1./dt)*inner(v, u - u0)*dx + inner(v, grad(u0)*u0)*dx \
        + nu*inner(grad(u), grad(v))*dx - inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# STEP 2 (p)
a2 = inner(grad(q), grad(p))*dx
L2 = -(1./dt)*q*div(utilde)*dx

# STEP 3 (Velocity update)
a3 = inner(v, u)*dx
L3 = inner(v, utilde)*dx - dt*inner(v, grad(p1))*dx

# Assemble matrices
# cannot use symmetric b.c.!
A1 = assemble(a1)
bc0u.apply(A1)
bc1u.apply(A1)
bc3u.apply(A1)

A2 = assemble(a2)
bc2p.apply(A2)

A3 = assemble(a3)
bc0u.apply(A3)
bc1u.apply(A3)
bc3u.apply(A3)
```

example17.py cont'd

```
# set up 3 quiet solvers for fixed matrices
solver1 = LUSolver(A1)
solver1.parameters["reuse_factorization"] = True
solver1.parameters["report"] = False

solver2 = LUSolver(A2)
solver2.parameters["reuse_factorization"] = True
solver2.parameters["report"] = False

solver3 = LUSolver(A3)
solver3.parameters["reuse_factorization"] = True
solver3.parameters["report"] = False

# timestepping
while t < endTime * (1. + 1.e-10):
    t += dt
    step += 1
```

example17.py cont'd

```
# STEP 1 solution: solve for utilde
b = assemble(L1)
bc0u.apply(b)
bc1u.apply(b)
bc3u.apply(b)
solver1.solve( utilde.vector(), b )
```

example17.py cont'd

```
# STEP 1 solution: solve for utilde
b = assemble(L1)
bc0u.apply(b)
bc1u.apply(b)
bc3u.apply(b)
solver1.solve( utilde.vector(), b )

# STEP 2 solution: solve for p1
b = assemble(L2)
bc2p.apply(b)
solver2.solve( p1.vector(), b )
```

example17.py cont'd

```
# STEP 1 solution: solve for utilde
b = assemble(L1)
bc0u.apply(b)
bc1u.apply(b)
bc3u.apply(b)
solver1.solve( utilde.vector(), b )

# STEP 2 solution: solve for p1
b = assemble(L2)
bc2p.apply(b)
solver2.solve( p1.vector(), b )

# STEP 3 solution: solve for u1
b = assemble(L3)
bc0u.apply(b); bc1u.apply(b); bc3u.apply(b)
solver3.solve( u1.vector(), b )
```

example17.py cont'd

```
# STEP 1 solution: solve for utilde
b = assemble(L1)
bc0u.apply(b)
bc1u.apply(b)
bc3u.apply(b)
solver1.solve( utilde.vector(), b )

# STEP 2 solution: solve for p1
b = assemble(L2)
bc2p.apply(b)
solver2.solve( p1.vector(), b )

# STEP 3 solution: solve for u1
b = assemble(L3)
bc0u.apply(b); bc1u.apply(b); bc3u.apply(b)
solver3.solve( u1.vector(), b )

# prepare for next time step
u0.assign(u1)
```

example17.py cont'd

```
# STEP 1 solution: solve for utilde
b = assemble(L1)
bc0u.apply(b)
bc1u.apply(b)
bc3u.apply(b)
solver1.solve( utilde.vector(), b )

# STEP 2 solution: solve for p1
b = assemble(L2)
bc2p.apply(b)
solver2.solve( p1.vector(), b )

# STEP 3 solution: solve for u1
b = assemble(L3)
bc0u.apply(b); bc1u.apply(b); bc3u.apply(b)
solver3.solve( u1.vector(), b )

# prepare for next time step
u0.assign(u1)

plot(u0,title="Velocity",rescale=False)

# print a little information each step
print "t=%f, max abs u=%e, max p=%e, v(.5,0)=%e" \
  % (t, max(abs(u1.vector().array())), \
     max(p1.vector().array()), u1([.35,0.2])[1] )
```

example17.py cont'd

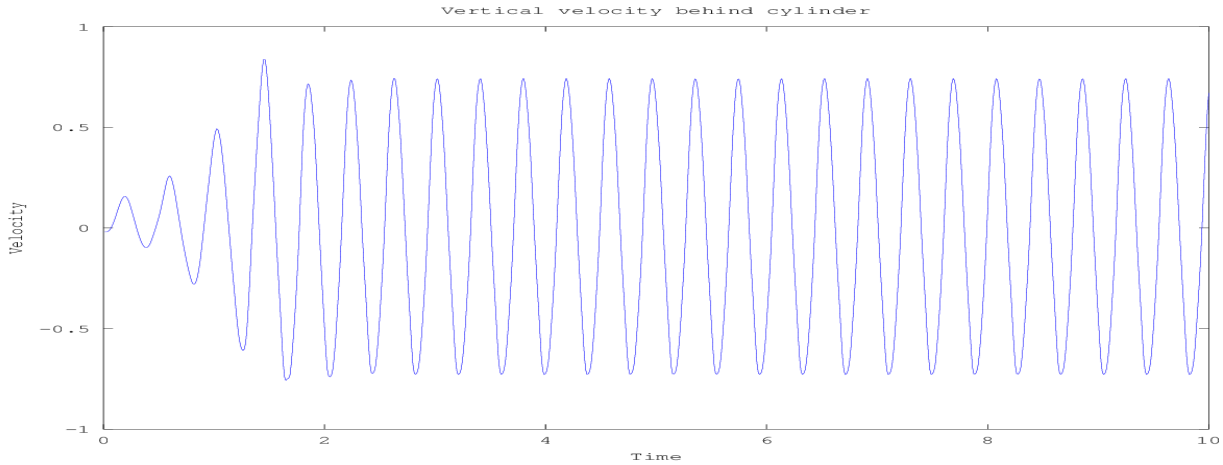
```
        # STEP 1 solution: solve for utilde
        b = assemble(L1)
        bc0u.apply(b)
        bc1u.apply(b)
        bc3u.apply(b)
        solver1.solve( utilde.vector(), b )

        # STEP 2 solution: solve for p1
        b = assemble(L2)
        bc2p.apply(b)
        solver2.solve( p1.vector(), b )

        # STEP 3 solution: solve for u1
        b = assemble(L3)
        bc0u.apply(b); bc1u.apply(b); bc3u.apply(b)
        solver3.solve( u1.vector(), b )

        # prepare for next time step
        u0.assign(u1)

        plot(u0,title="Velocity",rescale=False)

        # print a little information each step
        print "t=%f, max abs u=%e, max p=%e, v(.5,0)=%e" \
          % (t, max(abs(u1.vector().array())), \
             max(p1.vector().array()), u1([.35,0.2])[1] )

    umax = max(abs(u1.vector().array()))
    print "Worst possible Courant number=",dt*umax/mesh.hmin()
```

Remarks about the example

- **`example17.py`** runs in just about the same real time as **`example16.py`**
- Some problems would run faster, some slower
- Probably spends as much time plotting as solving
- Solution looks “slightly less viscous”
- Timestep plots look essentially similar

Vertical velocity plots

From **example16.py**



From **example17.py**