Generating high-order time stepping methods

Robert Kirby[1] Jorge Marchena-Menéndez[1] Patrick Farrell[2]

[1]Baylor University

[2]Oxford

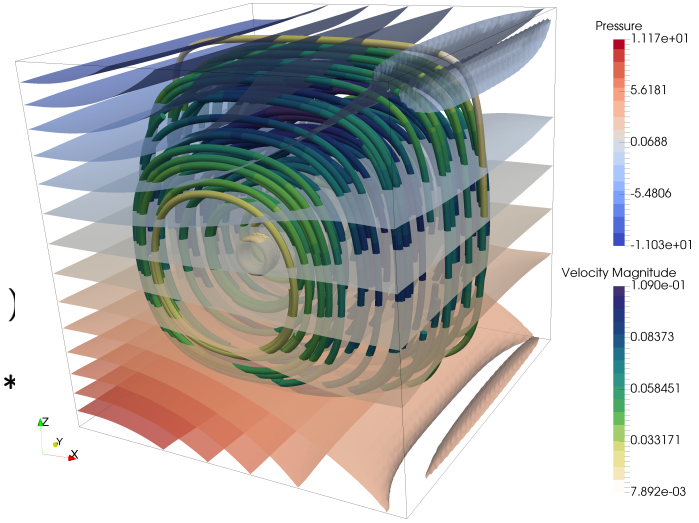
11-15 January 2021

Thanks to...

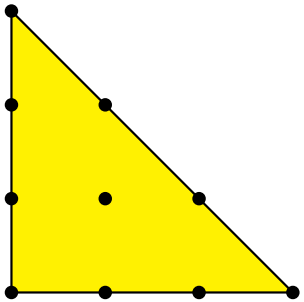
- NSF 1912653 (RCK, JMM)
- EPSRC EP/R029423/1, EP/V001493/1 (PF)

Firedrake success

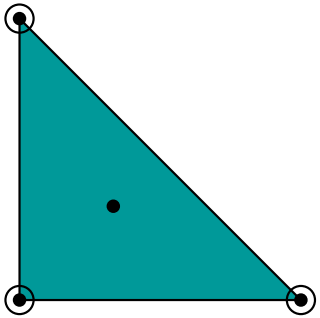
```
from firedrake import *
mesh = UnitCubeMesh(16, 16, 16)
V = VectorFunctionSpace(mesh, "Lagrange", 2)
W = FunctionSpace(mesh, "Lagrange", 1)
Z = V * W * W
upT = Function(Z);  u, p, T = split(upT)
v, q, S = TestFunctions(Z)
Ra = Constant(200.0); Pr = Constant(6.8)

g = Constant((0, 0, -1))
F = (
    inner(grad(u), grad(v))*dx + inner(dot(grad(u), u)
    - inner(p, div(v))*dx - Ra*Pr*inner(T*g, v)*dx
    + inner(div(u), q)*dx + inner(dot(grad(T), u), S)*
    + 1/Pr * inner(grad(T), grad(S))*dx
    )
nullspace = ...
bcs = ...
params = ...
solve(F == 0, upT, bcs=bcs, nullspace=nullspace,
    solver_parameters=params)
```



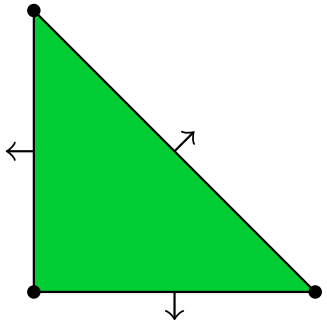
The fine art of spatial discretization



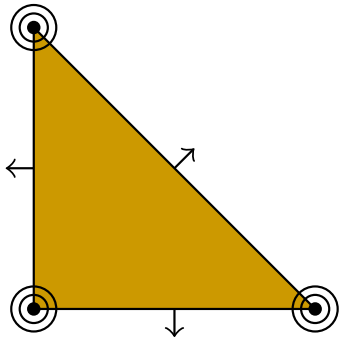
(a) Cubic Lagrange



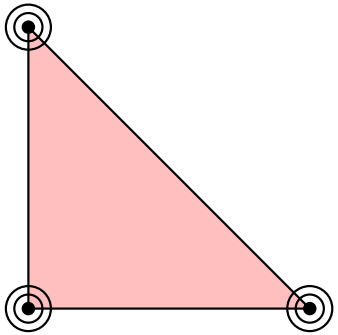
(b) Cubic Hermite



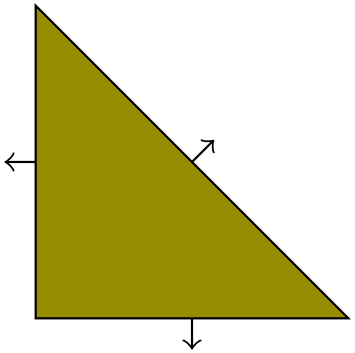
(c) Morley



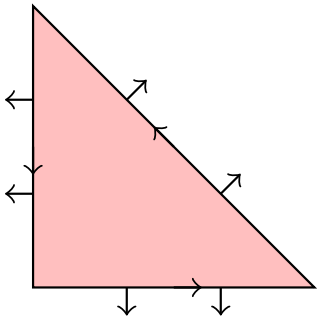
(d) Quintic Argyris



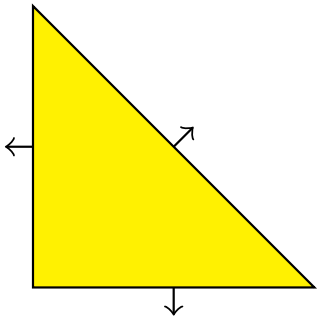
(e) Bell



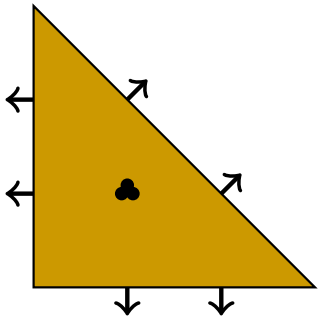
(a) Raviart-Thomas



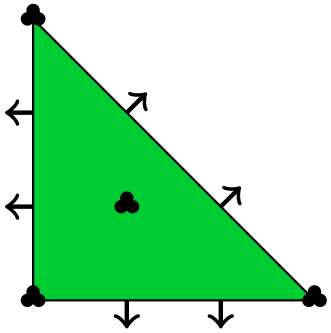
(b) Mardal-Tai-Winther



(c) HHJ



(d) AW^{nc}



(e) AW

But...time stepping!

$$\vec{u}_t - \Delta\vec{u} + \vec{u}\cdot\nabla\vec{u} + \nabla p = -\frac{\mathrm{Ra}}{\mathrm{Pr}}Tg\hat{\mathbf{z}}, \tag{1a}$$

$$\nabla\cdot\vec{u} = 0, \tag{1b}$$

$$\vec{T}_t - \mathrm{Pr}\Delta T + \vec{u}\cdot\nabla T = 0, \tag{1c}$$

But...time stepping!

$$\vec{u}_t - \Delta\vec{u} + \vec{u}\cdot\nabla\vec{u} + \nabla p = -\frac{\mathrm{Ra}}{\mathrm{Pr}} T g \hat{\mathbf{z}}, \tag{1a}$$

$$\nabla\cdot\vec{u} = 0, \tag{1b}$$

$$\vec{T}_t - \mathrm{Pr}\Delta T + \vec{u}\cdot\nabla T = 0, \tag{1c}$$

Why not?

PETScTS

Why not?

- Firedrake interface Works great (Ivan Yaschuk)

PETScTS

Why not?

- Firedrake interface Works great (Ivan Yaschuk)
- Many different methods...

PETScTS

Why not?

- Firedrake interface Works great (Ivan Yaschuk)
- Many different methods...
- Limited support for stable high-order implicit methods

Options

Multistep

Options

Multistep

- Simple to implement

Options

Multistep

- Simple to implement
- Use information from previous steps to gain efficiency

Options

Multistep

- Simple to implement
- Use information from previous steps to gain efficiency
- Dahlquist Barrier!

Options

Multistep

- Simple to implement
- Use information from previous steps to gain efficiency
- Dahlquist Barrier!

Runge-Kutta

Options

Multistep

- Simple to implement
- Use information from previous steps to gain efficiency
- Dahlquist Barrier!

Runge-Kutta

- Explicit or implicit

Options

Multistep

- Simple to implement
- Use information from previous steps to gain efficiency
- Dahlquist Barrier!

Runge-Kutta

- Explicit or implicit
- General order

Options

Multistep

- Simple to implement
- Use information from previous steps to gain efficiency
- Dahlquist Barrier!

Runge-Kutta

- Explicit or implicit
- General order
- Stability "options"

Options

Multistep

- Simple to implement
- Use information from previous steps to gain efficiency
- Dahlquist Barrier!

Runge-Kutta

- Explicit or implicit
- General order
- Stability "options"
- Common interface (Butcher tableaux)

Options

Multistep

- Simple to implement
- Use information from previous steps to gain efficiency
- Dahlquist Barrier!

Runge-Kutta

- Explicit or implicit
- General order
- Stability "options"
- Common interface (Butcher tableaux)
- Complexity/cost

Complexity and cost

Example: The heat equation

Math

$$(u_t, v) + (\nabla u, \nabla v) = 0$$

Example: The heat equation

Math

$$(u_t, v) + (\nabla u, \nabla v) = 0$$

(Almost) UFL

```
F = inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx
```

Runge-Kutta

Variational Form

$$(k_i, v_i) + \left(\nabla \left(u^n + \Delta t \sum_{j=1}^{s} a_{ij} k_j \right), \nabla v_i \right) - \left(f\left(t + c_i \Delta t, \cdot\right), v_i \right) = 0,$$

then

$$u^{n+1} = u^n + \Delta t \sum_{i=1}^{s} b_i k_i.$$

Runge-Kutta

Variational Form

$$(k_i, v_i) + \left(\nabla \left(u^n + \Delta t \sum_{j=1}^{s} a_{ij} k_j \right), \nabla v_i \right) - \left(f\left(t + c_i \Delta t, \cdot\right), v_i \right) = 0,$$

then

$$u^{n+1} = u^n + \Delta t \sum_{i=1}^{s} b_i k_i.$$

Block system

$$\left(\begin{bmatrix} M & 0 & 0 \\ 0 & M & 0 \\ 0 & 0 & M \end{bmatrix} + \Delta t \begin{bmatrix} a_{11}K & a_{12}K & a_{13}K \\ a_{21}K & a_{22}K & a_{23}K \\ a_{31}K & a_{32}K & a_{33}K \end{bmatrix} \right) \begin{bmatrix} \mathrm{k}_1 \\ \mathrm{k}_2 \\ \mathrm{k}_3 \end{bmatrix} = \begin{bmatrix} \mathrm{f}_1 \\ \mathrm{f}_2 \\ \mathrm{f}_3 \end{bmatrix},$$

From RK to UFL

LobattoIIIC(2)

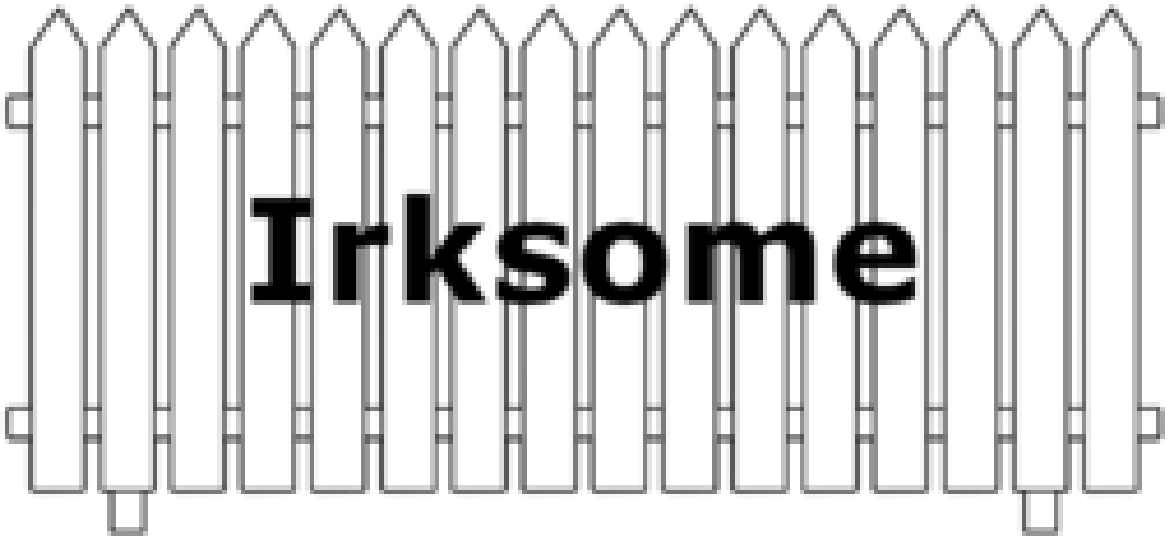
$$\begin{array}{c|cc} 0 & 1/2 & -1/2 \\ 1 & 1/2 & 1/2 \\ \hline & 1/2 & 1/2 \end{array}$$

UFL

```
Vbig = V * V
k = Function(Vbig)
k0, k1 = split(k)
v0, v1 = TestFunctions(Vbig)
u0 = u + Constant(0.5) * dt * k0 + Constant(-0.5) * dt * k1
u1 = u + Constant(0.5) * dt * k0 + Constant(0.5) * dt * k1

F = (inner(k0, v0) * dx + inner(k1, v1) * dx +
     inner(grad(u0), grad(v0)) * dx + inner(grad(u1), grad(v1)) * dx)
```

But solving the system at each time step?

“Labor in loneliness is irksome” – Mark Twain



Slaying the two-headed beast

Complexity

UFL manipulation:

Slaying the two-headed beast

Complexity

UFL manipulation:

- Semidiscrete UFL + Butcher tableau maps to fully discrete UFL

Slaying the two-headed beast

Complexity

UFL manipulation:

- Semidiscrete UFL + Butcher tableau maps to fully discrete UFL
- Boundary conditions set up

Slaying the two-headed beast

Complexity

UFL manipulation:

- Semidiscrete UFL + Butcher tableau maps to fully discrete UFL
- Boundary conditions set up
- Convenience class TimeStepper

Slaying the two-headed beast

Complexity

UFL manipulation:

- Semidiscrete UFL + Butcher tableau maps to fully discrete UFL
- Boundary conditions set up
- Convenience class TimeStepper
- Lots of Butcher tableaux available, easily extensible.

Slaying the two-headed beast

Complexity

UFL manipulation:

- Semidiscrete UFL + Butcher tableau maps to fully discrete UFL
- Boundary conditions set up
- Convenience class TimeStepper
- Lots of Butcher tableaux available, easily extensible.

Cost

Solve stage equations efficiently? Firedrake + PETSc

Slaying the two-headed beast

Complexity

UFL manipulation:

- Semidiscrete UFL + Butcher tableau maps to fully discrete UFL
- Boundary conditions set up
- Convenience class TimeStepper
- Lots of Butcher tableaux available, easily extensible.

Cost

Solve stage equations efficiently? Firedrake + PETSc

- FieldSplit segregates stages

Slaying the two-headed beast

Complexity

UFL manipulation:

- Semidiscrete UFL + Butcher tableau maps to fully discrete UFL
- Boundary conditions set up
- Convenience class TimeStepper
- Lots of Butcher tableaux available, easily extensible.

Cost

Solve stage equations efficiently? Firedrake + PETSc

- FieldSplit segregates stages
- PatchPC can couple stages locally for *monolithic* smoothers.

Slaying the two-headed beast

Complexity

UFL manipulation:

- Semidiscrete UFL + Butcher tableau maps to fully discrete UFL
- Boundary conditions set up
- Convenience class TimeStepper
- Lots of Butcher tableaux available, easily extensible.

Cost

Solve stage equations efficiently? Firedrake + PETSc

- FieldSplit segregates stages
- PatchPC can couple stages locally for *monolithic* smoothers.
- And you get the rest of PETSc!

Example: Heat equation

```
V = FunctionSpace(msh, "S", 3)
u = ... # set initial condition
v = TestFunction(V)
t = Constant(0)
dt = Constant(0.01)
F =  inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx - inner(rhs, v)*dx
bc = DirichletBC(V, 0, "on_boundary")
butcher_tableau = RadauIIA(2)
stepper = TimeStepper(F, butcher_tableau, t, dt, u, bcs=bc,
                      solver_parameters=...)
while float(t) < 1.0:
    stepper.advance()
    t.assign(float(t) + float(dt))
```

Solvers?

Mardal/Nilssen/Staff

$$P = \begin{bmatrix} M & 0 & 0 \\ 0 & M & 0 \\ 0 & 0 & M \end{bmatrix} + \Delta t \begin{bmatrix} a_{11}K & 0 & 0 \\ 0 & a_{22}K & 0 \\ 0 & 0 & a_{33}K \end{bmatrix}$$

See also Rana/Howle/Long/Meek/Milestone (2020)

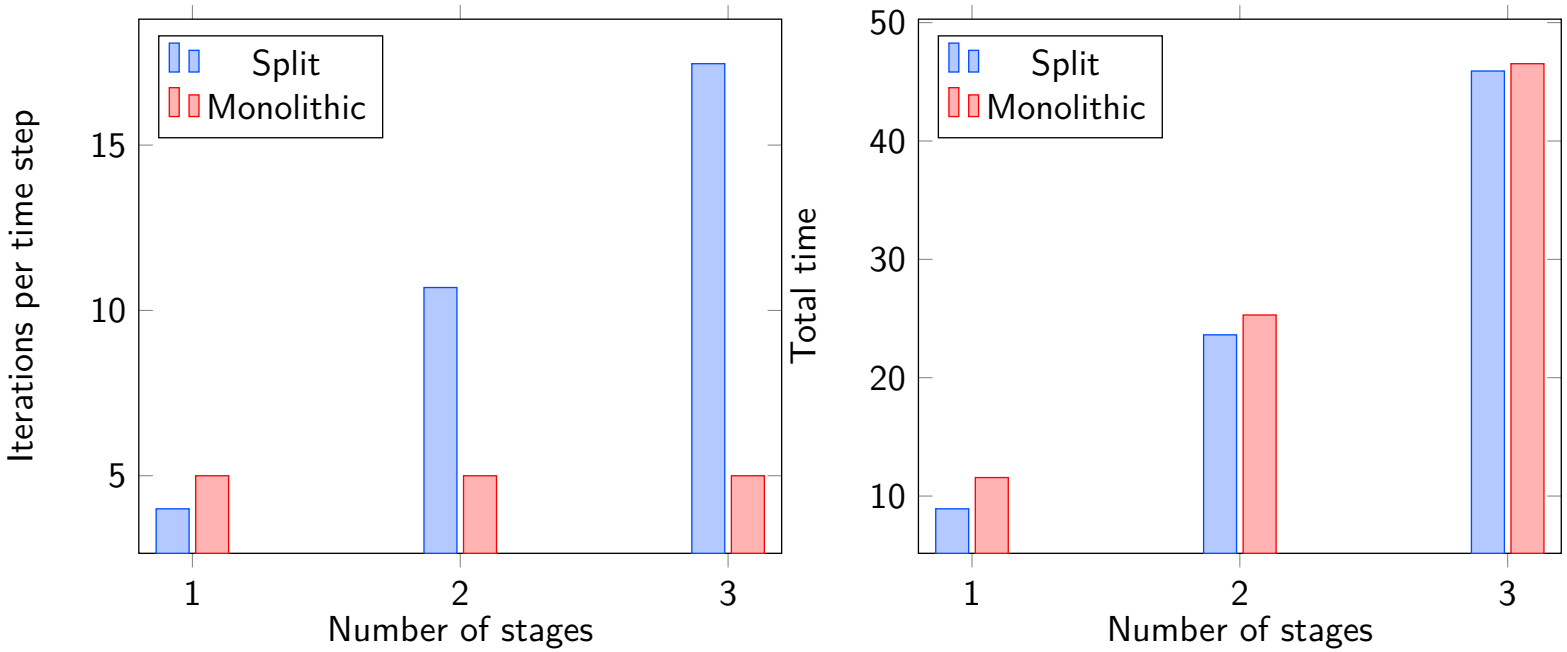
Solver parameters (Mardal/Nilssen/Staff)

```
params = {"snes_type": "ksponly",
  "ksp_type": "gmres",
  "pc_type": "fieldsplit",
  "pc_fieldsplit_type": "additive"
}

per_field = {"ksp_type": "preonly",
             "pc_type": "gamg"}

for s in range(butcher_tableau.num_stages):
    params["fieldsplit_%s" % (s,)] = per_field
```

Solver parameters: Stage-coupled multigrid

```
mgparams = {"mat_type": "aij",
            "snes_type": "ksponly",
            "ksp_type": "fgmres",
            "ksp_monitor_true_residual": None,
            "pc_type": "mg",
            "mg_levels_ksp_type": "chebyshev",
            "mg_levels_ksp_norm_type": "unpreconditioned",
            "mg_levels_pc_type": "python",
            "mg_levels_pc_python_type": "firedrake.PatchPC",
            "mg_levels_patch_pc_patch_save_operators": True,
            "mg_levels_patch_pc_patch_partition_of_unity": False,
            "mg_levels_patch_pc_patch_construct_type": "star",
            "mg_levels_patch_pc_patch_construct_dim": 0,
            "mg_levels_patch_pc_patch_sub_mat_type": "seqdense",
            "mg_levels_patch_pc_patch_dense_inverse": True,
            "mg_levels_patch_pc_patch_precompute_element_tensors": None,
            "mg_levels_patch_sub_ksp_type": "preonly",
            "mg_levels_patch_sub_pc_type": "lu",
            "mg_coarse_pc_type": "lu",
            "mg_coarse_pc_factor_mat_solver_type": "mumps"}
```

Results



(a) Average Krylov iterations per stage

(b) Total solver time

Figure: Solver performance for heat equation with Q^2 elements on a 128x128 mesh using RadauIIA time stepping with various numbers of stages. We compare the diagonal Mardal/Nillsen/Staff to a new monolotich multigrid approach.

Example: BBM

Benjamin-Bona-Mahoney

$$u_t + u_x + uu_x - u_{txx} = 0, \quad x \in \mathbb{R}, t \in \mathbb{R}^+$$

```
F = (inner(Dt(u), v) * dx + inner(u.dx(0), v) * dx
     + inner(u * u.dx(0), v) * dx
     + inner((Dt(u)).dx(0), v.dx(0)) * dx)
```

Example: BBM

Benjamin-Bona-Mahoney

$$u_t + u_x + uu_x - u_{txx} = 0, \quad x \in \mathbb{R}, t \in \mathbb{R}^+$$

Nonlinear Sobolev-type PDE

```
F = (inner(Dt(u), v) * dx + inner(u.dx(0), v) * dx
     + inner(u * u.dx(0), v) * dx
     + inner((Dt(u)).dx(0), v.dx(0)) * dx)
```

Example: BBM

Benjamin-Bona-Mahoney

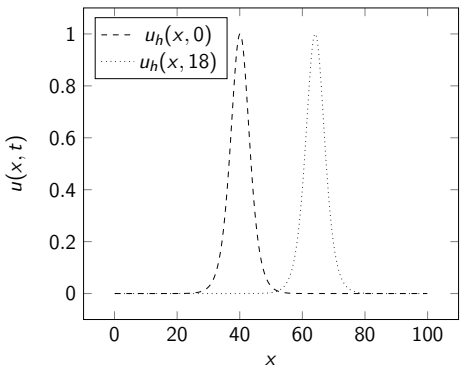
$$u_t + u_x + uu_x - u_{txx} = 0, \quad x \in \mathbb{R}, t \in \mathbb{R}^+$$

Nonlinear Sobolev-type PDE with three time-invariant quantities:

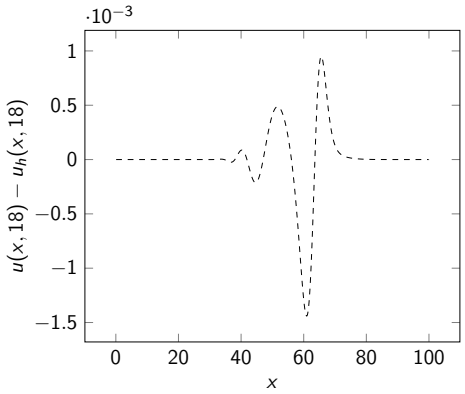
$$\begin{aligned} I_1 &= \int u \, dx, \\ I_2 &= \int u^2 + (u_x)^2 \, dx, \\ I_3 &= \int (u_x)^2 + \tfrac{1}{3} u^3 \, dx. \end{aligned}$$

```
F = (inner(Dt(u), v) * dx + inner(u.dx(0), v) * dx
     + inner(u * u.dx(0), v) * dx
     + inner((Dt(u)).dx(0), v.dx(0)) * dx)
```

BBM results



(a) Solution at $t = 0, 18$.



(b) Pointwise error, time $t = 18$.

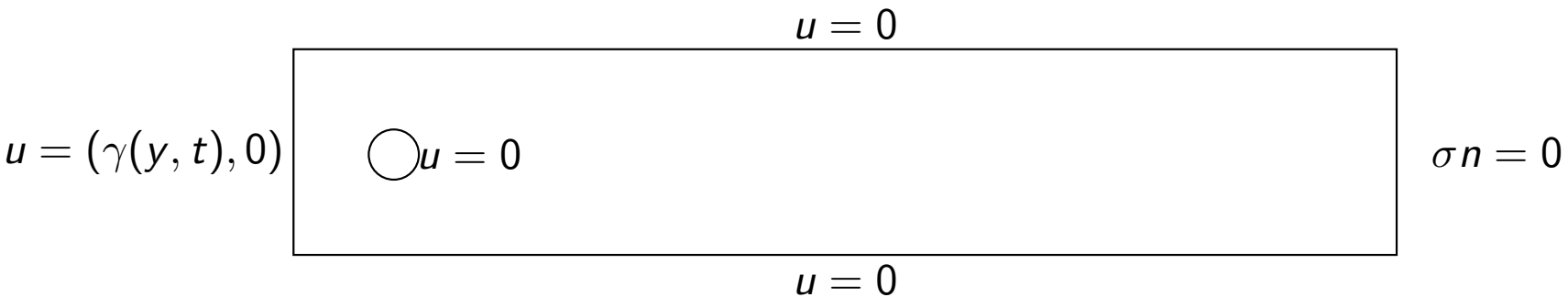
Figure: BBM solved with P^1 elements, GL(2) time stepping

Conserving invariants

Linear and quadratic invariants conserved exactly, cubics are close:

GL(1)				GL(2)			
t	$I_1(t)/I_1(0)$	$I_2(t)/I_2(0)$	$I_3(t)/I_3(0)$	t	$I_1(t)/I_1(0)$	$I_2(t)/I_2(0)$	$I_3(t)/I_3(0)$
0	1	1	1	0	1	1	1
6	1	1	0.9984516	6	1	1	0.9999996
12	1	1	0.9963326	12	1	1	0.9999992
18	1	1	0.9948969	18	1	1	0.9999990

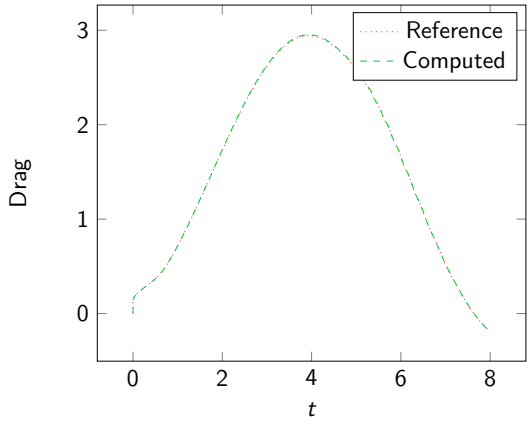
Navier-Stokes: Flow past cylinder



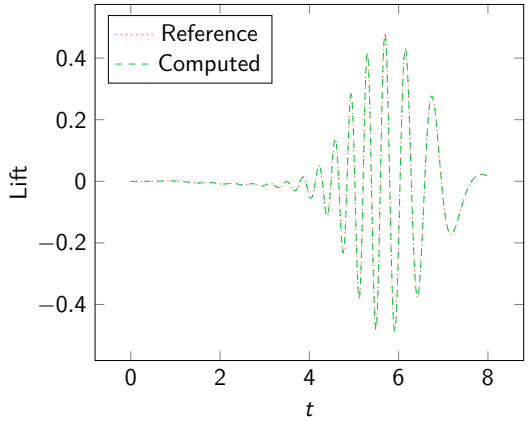
Method

- Us: Scott-Vogelius (P^4/P^3_{disc}) on curvilinear mesh with 7546 vertices, RadauIIA(2), $\Delta t = 1/800$
- Benchmark: Q^2/Q^0 on straight-edge quads, 133965 vertices, Crank-Nicolson, $\Delta t = 1/1600$.

Navier-Stokes: Flow past cylinder



(a) Drag



(b) Lift

Figure: Lift and drag calculations with RadauIIA(2), large time step

Conclusions

Lots to do!

Conclusions

Lots to do!

- Get "all" RK methods from UFL transformations

Conclusions

Lots to do!

- Get "all" RK methods from UFL transformations
- Rich solver possibilities

Conclusions

Lots to do!

- Get "all" RK methods from UFL transformations
- Rich solver possibilities
- Advanced use cases (NSE/MHD, Gross-Pitaevskii)

Conclusions

Lots to do!

- Get "all" RK methods from UFL transformations
- Rich solver possibilities
- Advanced use cases (NSE/MHD, Gross-Pitaevskii)
- IMEX/partitioned methods?