Irksome User Manual

Pablo Brubeck,
Patrick E. Farrell,
Robert C. Kirby,
Scott P. MacLachlan

First edition: 2024

CONTENTS

Irksome is a Python library that adds a temporal discretization layer on top of finite element discretizations provided by Firedrake. We provide a symbolic representation of time derivatives in UFL, allowing users to write weak forms of semidiscrete PDE. Irksome maps this and a Butcher tableau encoding a Runge-Kutta method into a fully discrete variational problem for the stage values. Irksome then leverages existing advanced solver technology in Firedrake and PETSc to allow for efficient computation of the Runge-Kutta stages. Convenience classes package the underlying lower-level manipulations and present users with a friendly high-level interface time stepping.

So, instead of manually coding UFL for backward Euler for the heat equation:

```
F = inner((unew - uold) / dt, v) * dx + inner(grad(unew), grad(v)) * dx
```

and rewriting this if you want a different time-stepping method, Irksome lets you write UFL for a semidiscrete form:

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

and maps this and a Butcher tableau for some Runge-Kutta method to UFL for a fully-discrete method. Hence, switching between RK methods is plug-and-play.

Irksome provides convenience classes to package the transformation of forms and boundary conditions and provide a method to advance by a time step. The underlying variational problem for the (possibly implicit!) Runge-Kutta stages composes fully with advanced Firedrake/PETSc solver technology, so you can use block preconditioners, multigrid with patch smoothers, and more – and in parallel, too!

CHAPTER

ONE

ACKNOWLEDGEMENTS

The Irksome team is deeply grateful to David Ham for setting up our initial documentation and Jack Betteridge for initializing the CI. Ivan Yashchuk set up the GitHub Actions. Lawrence Mitchell provided early assistance on some UFL manipulation and documentation.

We also acknowledge support from NSF 2410408.

CHAPTER

TWO

GETTING STARTED

Irksome requires Firedrake. Instructions for installing Firedrake can be found here. Once Firedrake is installed you can install Irksome by running:

```
$ pip install --src . --editable git+https://github.com/firedrakeproject/
↪Irksome.git#egg=Irksome
```

or, equivalently:

```
$ git clone https://github.com/firedrakeproject/Irksome.git
$ pip install --editable ./Irksome
```

CHAPTER THREE

TUTORIALS

After your installation works, please check out our demos.

The best place to start are with some simple heat and wave equations:

3.1 Solving the Heat Equation with Irksome

Let's start with the simple heat equation on $\Omega = [0, 10] \times [0, 10]$, with boundary Γ:

$$\begin{aligned} u_t - \Delta u &= f \\ u &= 0 \quad \text{on } \Gamma \end{aligned}$$

for some known function f. At each time t, the solution to this equation will be some function $u \in V$, for a suitable function space V.

We transform this into weak form by multiplying by an arbitrary test function $v \in V$ and integrating over Ω. We know have the variational problem of finding $u : [0, T] \to V$ such that

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

This demo implements an example used by Solin with a particular choice of f given below

As usual, we need to import firedrake:

```
from firedrake import *
```

We will also need to import certain items from irksome:

```
from irksome import GaussLegendre, Dt, MeshConstant, TimeStepper
```

We will create the Butcher tableau for the lowest-order Gauss-Legendre Runge-Kutta method, which is more commonly known as the implicit midpoint rule:

```
butcher_tableau = GaussLegendre(1)
ns = butcher_tableau.num_stages
```

Now we define the mesh and piecewise linear approximating space in standard Firedrake fashion:

```
N = 100
x0 = 0.0
x1 = 10.0
```

(continues on next page)

(continued from previous page)

```
y0 = 0.0
y1 = 10.0

msh = RectangleMesh(N, N, x1, y1)
V = FunctionSpace(msh, "CG", 1)
```

We define variables to store the time step and current time value:

```
MC = MeshConstant(msh)
dt = MC.Constant(10.0 / N)
t = MC.Constant(0.0)
```

This defines the right-hand side using the method of manufactured solutions:

```
x, y = SpatialCoordinate(msh)
S = Constant(2.0)
C = Constant(1000.0)
B = (x-Constant(x0))*(x-Constant(x1))*(y-Constant(y0))*(y-Constant(y1))/C
R = (x * x + y * y) ** 0.5
uexact = B * atan(t)*(pi / 2.0 - atan(S * (R - t)))
rhs = Dt(uexact) - div(grad(uexact))
```

We define the initial condition for the fully discrete problem, which will get overwritten at each time step:

```
u = Function(V)
u.interpolate(uexact)
```

Now, we will define the semidiscrete variational problem using standard UFL notation, augmented by the `Dt` operator from Irksome:

```
v = TestFunction(V)
F = inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx - inner(rhs, v)*dx
bc = DirichletBC(V, 0, "on_boundary")
```

Later demos will show how to use Firedrake's sophisticated interface to PETSc for efficient block solvers, but for now, we will solve the system with a direct method (Note: the matrix type needs to be explicitly set to aij if sparse direct factorization were used with a multi-stage method, as we wind up with a mixed problem that Firedrake will assemble into a PETSc MatNest otherwise):

```
luparams = {"mat_type": "aij",
            "ksp_type": "preonly",
            "pc_type": "lu"}
```

Most of Irksome's magic happens in the `TimeStepper`. It transforms our semidiscrete form *F* into a fully discrete form for the stage unknowns and sets up a variational problem to solve for the stages at each time step.:

```
stepper = TimeStepper(F, butcher_tableau, t, dt, u, bcs=bc,
                      solver_parameters=luparams)
```

This logic is pretty self-explanatory. We use the `TimeStepper`'s `advance` method, which solves the variational problem to compute the Runge-Kutta stage values and then updates the solution.:

```
while (float(t) < 1.0):
    if (float(t) + float(dt) > 1.0):
        dt.assign(1.0 - float(t))
    stepper.advance()
    print(float(t))
    t.assign(float(t) + float(dt))
```

Finally, we print out the relative L^2 error:

```
print()
print(norm(u-uexact)/norm(uexact))
```

3.2 Solving the Mixed form of the Heat Equation

This shows a different variational formulation of the heat equation. As before, we put $\Omega = [0, 10] \times [0, 10]$, with boundary Γ: but now we write the heat equation in mixed form:

$$\begin{aligned} \sigma + \nabla u &= 0 \\ u_t + \nabla \cdot \sigma &= f \end{aligned}$$

which gives rise to the weak form

$$\begin{aligned} (\sigma, v) - (u, \nabla \cdot v) &= 0 \\ (u_t, w) + (\nabla \cdot \sigma, w) &= (f, w) \end{aligned}$$

Here, σ is a vector-valued variable and will be discretized in a suitable $H(\mathrm{div})$ -conforming space. The variable u actually only requires L^2 regularity and will be discretized with discontinuous piecewise polynomials.

Note that this gives us a differential-algebraic system at the fully discrete level as there is no time derivative on σ.

Standard imports, although we're using a different RK scheme this time:

```
from firedrake import *
from irksome import LobattoIIIC, Dt, MeshConstant, TimeStepper

butcher_tableau = LobattoIIIC(2)
```

Build the mesh and approximating spaces:

```
N = 32
x0 = 0.0
x1 = 10.0
y0 = 0.0
y1 = 10.0
msh = RectangleMesh(N, N, x1, y1)
```

(continues on next page)

(continued from previous page)

```
V = FunctionSpace(msh, "RT", 2)
W = FunctionSpace(msh, "DG", 1)
Z = V * W
```

Create time and time-step variables:

```
MC = MeshConstant(msh)
dt = MC.Constant(10.0 / N)
t = MC.Constant(0.0)
```

As in the first heat demo, we build the RHS via the method of manufactured solutions:

```
x, y = SpatialCoordinate(msh)

S = Constant(2.0)
C = Constant(1000.0)

B = (x-Constant(x0))*(x-Constant(x1))*(y-Constant(y0))*(y-Constant(y1))/C
R = (x * x + y * y) ** 0.5

uexact = B * atan(t)*(pi / 2.0 - atan(S * (R - t)))
sigexact = -grad(uexact)

rhs = Dt(uexact) + div(sigexact)
```

Set up the initial condition:

```
sigu = project(as_vector([0, 0, uexact]), Z)
sigma, u = split(sigu)
```

And define the variational form:

```
v, w = TestFunctions(Z)

F = (inner(Dt(u), w) * dx + inner(div(sigma), w) * dx - inner(rhs, w) * dx
     + inner(sigma, v) * dx - inner(u, div(v)) * dx)
```

As before, we use a sparse direct method:

```
params = {"mat_type": "aij",
          "snes_type": "ksponly",
          "ksp_type": "preonly",
          "pc_type": "lu"}
```

We set the time stepper as before, except there are no strongly-enforced boundary conditiuons for the mixed method:

```
stepper = TimeStepper(F, butcher_tableau, t, dt, sigu,
                      solver_parameters=params)
```

And we advance the solution in time:

```
while (float(t) < 1.0):
    if (float(t) + float(dt) > 1.0):
        dt.assign(1.0 - float(t))
    stepper.advance()
    print(float(t))
    t.assign(float(t) + float(dt))
```

Finally, we check the accuracy of the solution:

```
sigma, u = sigu.subfunctions
print("U error      : ", errornorm(uexact, u) / norm(uexact))
print("Sig error    : ", errornorm(sigexact, sigma) / norm(sigexact))
print("Div Sig error: ",
      errornorm(sigexact, sigma, norm_type='Hdiv')
      / norm(sigexact, norm_type='Hdiv'))
```

3.3 Solving the Mixed Wave Equation: Energy conservation

Let Ω be the unit square with boundary Γ. We write the wave equation as a first-order system of PDE:

$$\begin{aligned} u_t + \nabla p &= 0 \\ p_t + \nabla \cdot u &= 0 \end{aligned}$$

together with homogeneous Dirichlet boundary conditions

$$p = 0 \quad \text{on } \Gamma$$

In this form, at each time, u is a vector-valued function in the Soboleve space $H(\mathrm{div})$ and *p* is a scalar-valued function. If we select appropriate test functions v and w, then we can arrive at the weak form

$$\begin{aligned} (u_t, v) - (p, \nabla \cdot v) &= 0 \\ (p_t, w) + (\nabla \cdot u, w) &= 0 \end{aligned}$$

Note that in mixed formulations, the Dirichlet boundary condition is weakly enforced via integration by parts rather than strongly in the definition of the approximating space.

In this example, we will use the next-to-lowest order Raviart-Thomas elements for the velocity variable u and discontinuous piecewise linear polynomials for the scalar variable p.

Here is some typical Firedrake boilerplate and the construction of a simple mesh and the approximating spaces:

```
from firedrake import *
from irksome import GaussLegendre, Dt, MeshConstant, TimeStepper

N = 10

msh = UnitSquareMesh(N, N)
V = FunctionSpace(msh, "RT", 2)
W = FunctionSpace(msh, "DG", 1)
Z = V*W
```

Now we can build the initial condition, which has zero velocity and a sinusoidal displacement:

```
x, y = SpatialCoordinate(msh)
up0 = project(as_vector([0, 0, sin(pi*x)*sin(pi*y)]), Z)
u0, p0 = split(up0)
```

We build the variational form in UFL:

```
v, w = TestFunctions(Z)
F = inner(Dt(u0), v)*dx + inner(div(u0), w) * dx + inner(Dt(p0), w)*dx -␣
↪inner(p0, div(v)) * dx
```

Energy conservation is an important principle of the wave equation, and we can test how well the spatial discretization conserves energy by creating a UFL expression and evaluating it at each time step:

```
E = 0.5 * (inner(u0, u0)*dx + inner(p0, p0)*dx)
```

The time and time step variables:

```
MC = MeshConstant(msh)
t = MC.Constant(0.0)
dt = MC.Constant(1.0/N)
```

The two-stage Gauss-Legendre method is, like all instances of that family, A-stable and symplectic. This gives us a fourth order method in time, although our spatial accuracy is of lower order. Feel free to experiment!:

```
butcher_tableau = GaussLegendre(2)
```

Like the heat equation demo, we are just using a direct method to solve the system at each time step:

```
params = {"mat_type": "aij",
          "snes_type": "ksponly",
          "ksp_type": "preonly",
          "pc_type": "lu"}

stepper = TimeStepper(F, butcher_tableau, t, dt, up0,
                      solver_parameters=params)
```

And, as with the heat equation, our time-stepping logic is quite simple. At easch time step, we print out the energy in the system:

```
print("Time    Energy")
print("==============")

while (float(t) < 1.0):
    if float(t) + float(dt) > 1.0:
        dt.assign(1.0 - float(t))

    stepper.advance()
```

(continues on next page)

(continued from previous page)

```
    t.assign(float(t) + float(dt))
    print("{0:1.1e} {1:5e}".format(float(t), assemble(E)))
```

If all is well with the world, the energy will be nearly identical (up to roundoff error) at each time step because the GL methods are symplectic and applied to a linear Hamiltonian system. As an exercise, the reader should edit this code to use other RK methods. In particular, *LobattoIIIC* and *BackwardEuler* or other Radau methods as well as the symplectic DIRK *QinZhang*.

To see a nonlinear problem in action, we have a simple lid-driven cavity example, too:

3.4 Solving the Navier-Stokes Equations with Irksome

Let's consider the lid-driven cavity problem on $\Omega = [0, 1] \times [0, 1]$, with boundary $\Gamma_T \cup \Gamma$, where Γ_T is the top of the domain, $0 \leq x \leq 1, y = 1$:

$$\begin{aligned} u_t + u \cdot \nabla u - \frac{1}{Re}\Delta u + \nabla p &= 0 \\ \nabla \cdot u &= 0 \\ u &= (0, 0) \quad \text{on } \Gamma \\ u &= (1, 0) \quad \text{on } \Gamma_T \end{aligned}$$

At each time t, the solution to this equation will be some functions $(u, p) \in V \times W$, for a suitable function spaces V, W.

We transform this into weak form by multiplying arbitrary test functions $v \in V$ and $w \in W$ and integrating over Ω. This gives the variational problem of finding $u : [0, T] \to V$ and $p : [0, T] \to W$ such that

$$\begin{aligned} (u_t, v) + (u \cdot \nabla u, v) + \frac{1}{Re}(\nabla u, \nabla v) - (p, \nabla \cdot v) &= 0 \\ (\nabla \cdot u, w) &= 0 \end{aligned}$$

As usual, we need to import firedrake:

```
from firedrake import *
```

We will also need to import certain items from irksome:

```
from irksome import RadauIIA, Dt, MeshConstant, TimeStepper
```

We will create the Butcher tableau for the two-stage RadauIIA Runge-Kutta method:

```
butcher_tableau = RadauIIA(2)
ns = butcher_tableau.num_stages
```

Now we define the mesh and Taylor-Hood approximating space in standard Firedrake fashion:

```
N = 32
msh = UnitSquareMesh(N, N)
V = VectorFunctionSpace(msh, "CG", 2)
W = FunctionSpace(msh, "CG", 1)
Z = V*W
```

We define variables to store the time step and current time value, as well as the Reynolds number:

```
MC = MeshConstant(msh)
dt = MC.Constant(1.0 / N)
t = MC.Constant(0.0)
Re = MC.Constant(10.0)
```

We define the solution over the product space, which will get overwritten at each time step:

```
up = Function(Z)
u, p = split(up)
```

Now, we will define the semidiscrete variational problem using standard UFL notation, augmented by the `Dt` operator from Irksome:

```
v, w = TestFunctions(Z)
F = (inner(Dt(u), v) * dx + inner(dot(u, grad(u)), v) * dx
     + 1/Re * inner(grad(u), grad(v)) * dx - inner(p, div(v)) * dx
     + inner(div(u), w) * dx)

bcs = [DirichletBC(Z.sub(0), as_vector([0, 0]), (1, 2, 3)),
       DirichletBC(Z.sub(0), as_vector([1, 0]), (4,))]
```

Later demos will show how to use Firedrake's sophisticated interface to PETSc for efficient solvers, but for now, we will solve the system with a direct method (Note: the matrix type needs to be explicitly set to aij if sparse direct factorization were used with a multi-stage method, as we wind up with a mixed problem that Firedrake will assemble into a PETSc MatNest otherwise):

```
luparams = {"mat_type": "aij",
            "snes_type": "newtonls",
            "snes_monitor": None,
            "ksp_type": "preonly",
            "pc_type": "lu",
            "pc_factor_mat_solver_type": "mumps",
            "snes_linesearch_type": "l2",
            "snes_force_iteration": 1,
            "snes_rtol": 1e-8,
            "snes_atol": 1e-8,
            }
```

Since this problem is ill-posed, we specify a nullspace vector to remove the possible constant "shift" in the pressure:

```
nsp = MixedVectorSpaceBasis(Z, [Z.sub(0), VectorSpaceBasis(constant=True,␣
↪comm=msh.comm)])
```

Most of Irksome's magic happens in the `TimeStepper`. It transforms our semidiscrete form *F* into a fully discrete form for the stage unknowns and sets up a variational problem to solve for the stages at each time step.:

```
stepper = TimeStepper(F, butcher_tableau, t, dt, up, bcs=bcs,
                      solver_parameters=luparams, nullspace=nsp)
```

This logic is pretty self-explanatory. We use the `TimeStepper`'s `advance` method, which solves the variational problem to compute the Runge-Kutta stage values and then updates the solution.:

```
for _ in range(N):
    print(f"Stepping from time {float(t)}")
    stepper.advance()
    t.assign(float(t) + float(dt))
```

Finally, we can visualize results of the simulation using Firedrake's plotting capabilities:

```
import matplotlib.pyplot as plt
from firedrake.pyplot import streamplot
u_, p_ = up.subfunctions
fig, axes = plt.subplots()
streamplot(u_, resolution=0.02, axes=axes)
axes.set_aspect("equal")
fig.savefig("demo_nse_streamlines.png")
```

Since those demos invariably rely on the non-scalable LU factorization, we have several demos showing how to work with Firedrake solver options to deploy more efficient methods:

3.5 Block diagonal preconditioners for the heat equation

This demo applies the method suggested in:

Mardal, Nilssen, Staff, "Order-optimal preconditioners for implicit Runge-Kutta schemes applied to parabolic PDEs", SISC 29(1): 361–375 (2007),

to our ongoing heat equation demonstration problem on $\Omega = [0, 10] \times [0, 10]$, with boundary Γ, giving rise to the weak form

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

A multi-stage RK method applied to the heat equation gives a block-structured system. The on-diagonal blocks are quite similar to what one obtains from a backward Euler discretization of the equation.

With a 2-stage method, we have

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

And the suggestion (analyzed rigorously) of Mardal, Nilssen, and Staff is to use a block diagonal preconditioner:

$$P = \begin{bmatrix} A_{11} & 0 \\ 0 & A_{22} \end{bmatrix}$$

This allows one to leverage an existing methodology for a low order method like backward Euler for the diagonal blocks. In our case, we will simply use an algebraic multigrid scheme, although one could certainly use geometric multigrid or some other technique.

Common set-up for the problem:

```
from firedrake import *  # noqa: F403
from irksome import LobattoIIIC, TimeStepper, Dt, MeshConstant

butcher_tableau = LobattoIIIC(3)

N = 64

x0 = 0.0
x1 = 10.0
y0 = 0.0
y1 = 10.0

msh = RectangleMesh(N, N, x1, y1)

MC = MeshConstant(msh)
dt = MC.Constant(10. / N)
t = MC.Constant(0.0)

V = FunctionSpace(msh, "CG", 1)
x, y = SpatialCoordinate(msh)

S = Constant(2.0)
C = Constant(1000.0)
B = (x-Constant(x0))*(x-Constant(x1))*(y-Constant(y0))*(y-Constant(y1))/C
R = (x * x + y * y) ** 0.5

uexact = B * atan(t)*(pi / 2.0 - atan(S * (R - t)))
rhs = Dt(uexact) - div(grad(uexact))
u = Function(V)
u.interpolate(uexact)
v = TestFunction(V)
F = inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx - inner(rhs, v)*dx

bc = DirichletBC(V, 0, "on_boundary")
```

Now, we define the solver parameters. PETSc-speak for taking the block diagonal is an “additive fieldsplit”:

```
params = {"mat_type": "aij",
          "snes_type": "ksponly",
          "ksp_type": "gmres",
          "ksp_monitor": None,
          "pc_type": "fieldsplit",   # block preconditioner
          "pc_fieldsplit_type": "additive"  # block diagaonal
          }
```

We also have to configure the (approximate) inverse of for each diagonal block. We’ll just apply a sweek of gamg (PETSC’s algebraic multigrid):

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

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

Note that we have used the same technique for each RK stage, which is probably typical. However, it is not necessary at all.

To test this preconditioning strategy, we'll create a time stepping object which will set up the variational problem for us:

```
stepper = TimeStepper(F, butcher_tableau, t, dt, u, bcs=bc,
                      solver_parameters=params)
```

But, since we're just testing the efficacy of the preconditioner, we'll solve the inside variational problem one time:

```
stepper.solver.solve()
```

3.6 Solving the heat equation with monolithic multigrid

This reprise of the heat equation demo uses a monolithic multigrid algorithm suggested by Patrick Farrell to perform time advancement.

We consider the heat equation on $\Omega = [0, 10] \times [0, 10]$, with boundary Γ: giving rise to the weak form

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

This demo implements an example used by Solin with a particular choice of f given below

We perform similar imports and setup as before:

```
from firedrake import *
from irksome import GaussLegendre, Dt, MeshConstant, TimeStepper
butcher_tableau = GaussLegendre(2)
```

However, we need to set up a mesh hierarchy to enable geometric multigrid within Firedrake:

```
N = 128
x0 = 0.0
x1 = 10.0
y0 = 0.0
y1 = 10.0

from math import log
coarseN = 8  # size of coarse grid
nrefs = log(N/coarseN, 2)
assert nrefs == int(nrefs)
nrefs = int(nrefs)
base = RectangleMesh(coarseN, coarseN, x1, y1)
```

(continues on next page)

(continued from previous page)

```
mh = MeshHierarchy(base, nrefs)
msh = mh[-1]
```

From here, setting up the function space, manufactured solution, etc, are just as for the regular heat equation demo:

```
V = FunctionSpace(msh, "CG", 1)

MC = MeshConstant(msh)
dt = MC.Constant(10.0 / N)
t = MC.Constant(0.0)

x, y = SpatialCoordinate(msh)
S = Constant(2.0)
C = Constant(1000.0)
B = (x-Constant(x0))*(x-Constant(x1))*(y-Constant(y0))*(y-Constant(y1))/C
R = (x * x + y * y) ** 0.5
uexact = B * atan(t)*(pi / 2.0 - atan(S * (R - t)))
rhs = Dt(uexact) - div(grad(uexact))

u = Function(V)
u.interpolate(uexact)
v = TestFunction(V)
F = inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx - inner(rhs, v)*dx
bc = DirichletBC(V, 0, "on_boundary")
```

And now for the solver parameters. Note that we are solving a block-wise system with all stages coupled together. This performs a monolithic multigrid with pointwise block Jacobi preconditioning:

```
mgparams = {"mat_type": "aij",
            "snes_type": "ksponly",
            "ksp_type": "gmres",
            "ksp_monitor_true_residual": None,
            "pc_type": "mg",
            "mg_levels": {
                "ksp_type": "chebyshev",
                "ksp_max_it": 1,
                "ksp_convergence_test": "skip",
                "pc_type": "python",
                "pc_python_type": "firedrake.PatchPC",
                "patch": {
                    "pc_patch": {
                        "save_operators": True,
                        "partition_of_unity": False,
                        "construct_type": "star",
                        "construct_dim": 0,
                        "sub_mat_type": "seqdense",
                        "dense_inverse": True,
                        "precompute_element_tensors": None},
```

(continues on next page)

(continued from previous page)

```
                    "sub": {
                        "ksp_type": "preonly",
                        "pc_type": "lu"}}},
            "mg_coarse": {
                "pc_type": "lu",
                "pc_factor_mat_solver_type": "mumps"}
            }
```

These solver parameters work just fine in the `TimeStepper`:

```
stepper = TimeStepper(F, butcher_tableau, t, dt, u, bcs=bc,
                      solver_parameters=mgparams)
```

And we can advance the solution in time in typical fashion:

```
while (float(t) < 1.0):
    if (float(t) + float(dt) > 1.0):
        dt.assign(1.0 - float(t))
    stepper.advance()
    print(float(t), flush=True)
    t.assign(float(t) + float(dt))
```

After the solve, we can retrieve some statistics about the solver:

```
steps, nonlinear_its, linear_its = stepper.solver_stats()

print("Total number of timesteps was %d" % (steps))
print("Average number of nonlinear iterations per timestep was %.2f" %␣
↪(nonlinear_its/steps))
print("Average number of linear iterations per timestep was %.2f" % (linear_
↪its/steps))
```

Finally, we print out the relative L^2 error:

```
print()
print(norm(u-uexact)/norm(uexact))
```

3.7 Solving the heat equation with monolithic multigrid and Discontinuous Galerkin-in-Time

This reprise of the heat equation demo uses a monolithic multigrid algorithm suggested by Patrick Farrell to perform time advancement, but now applies it to the Discontinuous Galerkin-in-Time discretization.

We consider the heat equation on $\Omega = [0, 10] \times [0, 10]$, with boundary Γ: giving rise to the weak form

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

This demo implements an example used by Solin with a particular choice of f given below

We perform similar imports and setup as before:

```
from firedrake import *
from irksome import DiscontinuousGalerkinTimeStepper, Dt, MeshConstant
```

However, we need to set up a mesh hierarchy to enable geometric multigrid within Firedrake:

```
N = 128
x0 = 0.0
x1 = 10.0
y0 = 0.0
y1 = 10.0

from math import log
coarseN = 8  # size of coarse grid
nrefs = log(N/coarseN, 2)
assert nrefs == int(nrefs)
nrefs = int(nrefs)
base = RectangleMesh(coarseN, coarseN, x1, y1)
mh = MeshHierarchy(base, nrefs)
msh = mh[-1]
```

From here, setting up the function space, manufactured solution, etc, are just as for the regular heat equation demo:

```
V = FunctionSpace(msh, "CG", 1)

MC = MeshConstant(msh)
dt = MC.Constant(10.0 / N)
t = MC.Constant(0.0)

x, y = SpatialCoordinate(msh)
S = Constant(2.0)
C = Constant(1000.0)
B = (x-Constant(x0))*(x-Constant(x1))*(y-Constant(y0))*(y-Constant(y1))/C
R = (x * x + y * y) ** 0.5
uexact = B * atan(t)*(pi / 2.0 - atan(S * (R - t)))
rhs = Dt(uexact) - div(grad(uexact))

u = Function(V)
u.interpolate(uexact)
v = TestFunction(V)
F = inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx - inner(rhs, v)*dx
bc = DirichletBC(V, 0, "on_boundary")
```

And now for the solver parameters. Note that we are solving a block-wise system with all stages coupled together. This performs a monolithic multigrid with pointwise block Jacobi preconditioning:

```
mgparams = {"mat_type": "aij",
            "snes_type": "ksponly",
            "ksp_type": "gmres",
            "ksp_monitor_true_residual": None,
```

(continues on next page)

(continued from previous page)

```
            "pc_type": "mg",
            "mg_levels": {
                "ksp_type": "chebyshev",
                "ksp_max_it": 1,
                "ksp_convergence_test": "skip",
                "pc_type": "python",
                "pc_python_type": "firedrake.PatchPC",
                "patch": {
                    "pc_patch": {
                        "save_operators": True,
                        "partition_of_unity": False,
                        "construct_type": "star",
                        "construct_dim": 0,
                        "sub_mat_type": "seqdense",
                        "dense_inverse": True,
                        "precompute_element_tensors": None},
                    "sub": {
                        "ksp_type": "preonly",
                        "pc_type": "lu"}}},
            "mg_coarse": {
                "pc_type": "lu",
                "pc_factor_mat_solver_type": "mumps"}
            }
```

These solver parameters work just fine in the *DiscontinuousGalerkinTimeStepper*:

```
stepper = DiscontinuousGalerkinTimeStepper(F, 2, t, dt, u, bcs=bc,
                                           solver_parameters=mgparams)
```

And we can advance the solution in time in typical fashion:

```
while (float(t) < 1.0):
    if (float(t) + float(dt) > 1.0):
        dt.assign(1.0 - float(t))
    stepper.advance()
    print(float(t), flush=True)
    t.assign(float(t) + float(dt))
```

After the solve, we can retrieve some statistics about the solver:

```
steps, nonlinear_its, linear_its = stepper.solver_stats()

print("Total number of timesteps was %d" % (steps))
print("Average number of nonlinear iterations per timestep was %.2f" %␣
↪(nonlinear_its/steps))
print("Average number of linear iterations per timestep was %.2f" % (linear_
↪its/steps))
```

Finally, we print out the relative L^2 error:

```
print()
print(norm(u-uexact)/norm(uexact))
```

3.8 Rana/Howle/et al preconditioning

This demo applies a method suggested in:

Rana, Howle, Long, Meek, Milestone "A new block preconditioner for implicit Runge-Kutta methods for parabolic PDE problems," SISC 2021

to our ongoing heat equation demonstration problem on $\Omega = [0, 10] \times [0, 10]$, with boundary Γ, giving rise to the weak form

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

A multi-stage RK method applied to the heat equation gives a block-structured system. The on-diagonal blocks are quite similar to what one obtains from a backward Euler discretization of the equation.

With a 2-stage method, we have

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

The suggestion in the paper is to approximate the Butcher matrix A with with a triangular approximation. For example, if A = LDU, then one could approximate A with LD or DU. This gives rise to a block triangular preconditioner

$$P = \begin{bmatrix} \tilde{A}_{11} & 0 \\ \tilde{A}_{21} & \tilde{A}_{22} \end{bmatrix}$$

This allows one to leverage an existing methodology for a low order method like backward Euler for the diagonal blocks. Empirical results suggest that this method is strongly stage-independent.

Common set-up for the problem:

```
from firedrake import *  # noqa: F403
from irksome import LobattoIIIC, TimeStepper, Dt, MeshConstant

butcher_tableau = LobattoIIIC(3)

N = 16

x0 = 0.0
x1 = 10.0
y0 = 0.0
y1 = 10.0

msh = RectangleMesh(N, N, x1, y1)

MC = MeshConstant(msh)
dt = MC.Constant(10. / N)
```

(continues on next page)

(continued from previous page)

```
t = MC.Constant(0.0)

V = FunctionSpace(msh, "CG", 1)
x, y = SpatialCoordinate(msh)

S = Constant(2.0)
C = Constant(1000.0)
B = (x-Constant(x0))*(x-Constant(x1))*(y-Constant(y0))*(y-Constant(y1))/C
R = (x * x + y * y) ** 0.5

uexact = B * atan(t)*(pi / 2.0 - atan(S * (R - t)))
rhs = Dt(uexact) - div(grad(uexact))
u = Function(V)
u.interpolate(uexact)

v = TestFunction(V)
F = inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx - inner(rhs, v) * dx

bc = DirichletBC(V, 0, "on_boundary")
```

Now, we define the solver parameters. We get at the Rana method through an Irksome-provided Python preconditioner. This method inherits from `firedrake.AuxiliaryOperatorPC` (it provides the Jacobian of the variational form with the approximate Butcher tableu substituted) and so provides the user with an 'aux' preconditioner to configure. Since the Rana technique gives us a block triangular matrix, a multiplicative field split exactly applies the preconditioner if its diagonal blocks are exactly inverted:

```
params = {"mat_type": "aij",
          "snes_type": "ksponly",
          "ksp_type": "gmres",
          "ksp_monitor": None,
          "pc_type": "python",
          "pc_python_type": "irksome.RanaLD",
          "aux": {
              "pc_type": "fieldsplit",
              "pc_fieldsplit_type": "multiplicative"
          }}
```

But they don't have to be. We'll approximate the inverse of each diagonal block with a sweep of gamg (PETSc's multigrid):

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

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

Note that we have used the same technique for each RK stage, which is probably typical. However, it is not necessary at all.

To test this preconditioning strategy, we'll create a time stepping object which will set up the variational problem for us. (Important note: The stepper puts some special information into a PETSc context for the variational problem it configures. This is vital for the Rana-type preconditioners to function. If you want to use the preconditioner outside of the `TimeStepper` then you will have some extra setup to do):

```
stepper = TimeStepper(F, butcher_tableau, t, dt, u, bcs=bc,
                      solver_parameters=params)
```

But, since we're just testing the efficacy of the preconditioner, we'll solve the inside variational problem one time:

```
stepper.solver.solve()
```

3.9 Accessing DIRK methods

Many practitioners favor diagonally implicit methods over fully implicit ones since the stages can be computed sequentially rather than concurrently. We support a range of DIRK methods and provide a convenient high-level interface that is very similar to other RK schemes. This demo is intended to show how to access DIRK methods seamlessly in Irksome.

This example uses the Qin Zhang symplectic DIRK to attack the mixed form of the wave equation. Let Ω be the unit square with boundary Γ. We write the wave equation as a first-order system of PDE:

$$\begin{aligned} u_t + \nabla p &= 0 \\ p_t + \nabla \cdot u &= 0 \end{aligned}$$

together with homogeneous Dirichlet boundary conditions

$$p = 0 \quad \text{on } \Gamma$$

In this form, at each time, u is a vector-valued function in the Sobolev space $H(\mathrm{div})$ and *p* is a scalar-valued function. If we select appropriate test functions v and w, then we can arrive at the weak form (see the mixed wave demos for more information):

$$\begin{aligned} (u_t, v) - (p, \nabla \cdot v) &= 0 \\ (p_t, w) + (\nabla \cdot u, w) &= 0 \end{aligned}$$

As in that case, we will use the next-to-lowest order Raviart-Thomas space for u and discontinuous piecewise linear elements for p.

As an example, we will use the two-stage A-stable and symplectic DIRK of Qin and Zhang, given by Butcher tableau:

$$\begin{array}{c|cc} 1/4 & 1/4 & 0 \\ 3/4 & 1/2 & 1/4 \\ \hline & 1/2 & 1/2 \end{array}$$

Imports from Firedrake and Irksome:

```
from firedrake import *
from irksome import QinZhang, Dt, MeshConstant, TimeStepper
```

We configure the discretization:

```
N = 10
msh = UnitSquareMesh(N, N)

MC = MeshConstant(msh)
t = MC.Constant(0.0)
dt = MC.Constant(1.0/N)

V = FunctionSpace(msh, "RT", 2)
W = FunctionSpace(msh, "DG", 1)
Z = V*W

v, w = TestFunctions(Z)

butcher_tableau = QinZhang()
```

And set up the initial condition and variational problem:

```
x, y = SpatialCoordinate(msh)
up0 = project(as_vector([0, 0, sin(pi*x)*sin(pi*y)]), Z)

u0, p0 = split(up0)

F = inner(Dt(u0), v)*dx + inner(div(u0), w) * dx + inner(Dt(p0), w)*dx -␣
↪inner(p0, div(v)) * dx
```

We will keep track of the energy to determine whether we're sufficiently accurate in solving the linear system:

```
E = 0.5 * (inner(u0, u0)*dx + inner(p0, p0)*dx)
```

When stepping with a DIRK, we only solve for one stage at a time. Although we could configure PETSc to try some iterative solver, here we will just use a direct method:

```
params = {"mat_type": "aij",
          "snes_type": "ksponly",
          "ksp_type": "preonly",
          "pc_type": "lu"}
```

Now, we just set the stage type to be "dirk" in Irksome and we're ready to advance in time:

```
stepper = TimeStepper(F, butcher_tableau, t, dt, up0,
                      stage_type="dirk",
                      solver_parameters=params)

print("Time    Energy")
print("==============")
while (float(t) < 1.0):
    if float(t) + float(dt) > 1.0:
        dt.assign(1.0 - float(t))

    stepper.advance()
```

(continues on next page)

(continued from previous page)

```
    print("{0:1.1e} {1:5e}".format(float(t), assemble(E)))

    t.assign(float(t) + float(dt))
```

If all is right in the universe, you should see that the energy remains constant.

We now have support for DIRKs:

3.10 Solving the Heat Equation with a DIRK in Irksome

Let's start with the simple heat equation on $\Omega = [0, 10] \times [0, 10]$, with boundary Γ:

$$\begin{aligned} u_t - \Delta u &= f \\ u &= 0 \quad \text{on } \Gamma \end{aligned}$$

for some known function f. At each time t, the solution to this equation will be some function $u \in V$, for a suitable function space V.

We transform this into weak form by multiplying by an arbitrary test function $v \in V$ and integrating over Ω. We know have the variational problem of finding $u : [0, T] \rightarrow V$ such that

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

This demo implements an example used by Solin with a particular choice of f given below

As usual, we need to import firedrake:

```
from firedrake import *
```

We will also need to import certain items from irksome:

```
from irksome import Alexander, Dt, MeshConstant, TimeStepper
```

We will create the Butcher tableau for a three-stage L-stable DIRK due to Alexander (SINUM 14(6): 1006-1021, 1977).:

```
butcher_tableau = Alexander()
ns = butcher_tableau.num_stages
```

Now we define the mesh and piecewise linear approximating space in standard Firedrake fashion:

```
N = 100
x0 = 0.0
x1 = 10.0
y0 = 0.0
y1 = 10.0

msh = RectangleMesh(N, N, x1, y1)
V = FunctionSpace(msh, "CG", 1)
```

We define variables to store the time step and current time value:

```
MC = MeshConstant(msh)
dt = MC.Constant(10.0 / N)
t = MC.Constant(0.0)
```

This defines the right-hand side using the method of manufactured solutions:

```
x, y = SpatialCoordinate(msh)
S = Constant(2.0)
C = Constant(1000.0)
B = (x-Constant(x0))*(x-Constant(x1))*(y-Constant(y0))*(y-Constant(y1))/C
R = (x * x + y * y) ** 0.5
uexact = B * atan(t)*(pi / 2.0 - atan(S * (R - t)))
rhs = Dt(uexact) - div(grad(uexact))
```

We define the initial condition for the fully discrete problem, which will get overwritten at each time step:

```
u = Function(V)
u.interpolate(uexact)
```

Now, we will define the semidiscrete variational problem using standard UFL notation, augmented by the `Dt` operator from Irksome:

```
v = TestFunction(V)
F = inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx - inner(rhs, v)*dx
bc = DirichletBC(V, 0, "on_boundary")
```

We'll just solve each stage with gamg + gmres:

```
params = {"mat_type": "aij",
          "ksp_type": "gmres",
          "pc_type": "gamg"}
```

Most of Irksome's magic happens in the `TimeStepper`. It transforms our semidiscrete form *F* into a fully discrete form for the stage unknowns and sets up a variational problem to solve for the stages at each time step.:

```
stepper = TimeStepper(F, butcher_tableau, t, dt, u, bcs=bc,
                      solver_parameters=params,
                      stage_type="dirk")
```

This logic is pretty self-explanatory. We use the `TimeStepper`'s `advance` method, which solves the variational problem to compute the Runge-Kutta stage values and then updates the solution.:

```
while (float(t) < 1.0):
    if (float(t) + float(dt) > 1.0):
        dt.assign(1.0 - float(t))
    stepper.advance()
    print(float(t))
    t.assign(float(t) + float(dt))
```

Now, check the relative L^2 error:

```
print()
print(norm(u-uexact)/norm(uexact))
```

And report the solver statistics:

```
num_steps, _, num_lin_its = stepper.solver_stats()

print(f"{num_steps} steps taken")
print(f"{num_lin_its} linear iterations taken")
print(f"That's {num_lin_its / num_steps} per time step")
print(f"And {num_lin_its / num_steps / 3} average per stage")
```

and for Galerkin-in-Time:

3.11 Solving the Mixed Wave Equation: Energy conservation, Multigrid, Galerkin-in-Time

Let Ω be the unit square with boundary Γ. We write the wave equation as a first-order system of PDE:

$$\begin{aligned} u_t + \nabla p &= 0 \\ p_t + \nabla \cdot u &= 0 \end{aligned}$$

together with homogeneous Dirichlet boundary conditions

$$p = 0 \quad \text{on } \Gamma$$

In this form, at each time, u is a vector-valued function in the Soboleve space $H(\mathrm{div})$ and *p* is a scalar-valued function. If we select appropriate test functions v and w, then we can arrive at the weak form

$$\begin{aligned} (u_t, v) - (p, \nabla \cdot v) &= 0 \\ (p_t, w) + (\nabla \cdot u, w) &= 0 \end{aligned}$$

Note that in mixed formulations, the Dirichlet boundary condition is weakly enforced via integration by parts rather than strongly in the definition of the approximating space.

In this example, we will use the next-to-lowest order Raviart-Thomas elements for the velocity variable u and discontinuous piecewise linear polynomials for the scalar variable p.

Here is some typical Firedrake boilerplate and the construction of a simple mesh and the approximating spaces. We are going to use a multigrid preconditioner for each timestep, so we create a MeshHierarchy as well:

```
from firedrake import *
from irksome import Dt, MeshConstant, GalerkinTimeStepper

N = 10

base = UnitSquareMesh(N, N)
mh = MeshHierarchy(base, 2)
```

(continues on next page)

(continued from previous page)

```
msh = mh[-1]
V = FunctionSpace(msh, "RT", 2)
W = FunctionSpace(msh, "DG", 1)
Z = V*W
```

Now we can build the initial condition, which has zero velocity and a sinusoidal displacement:

```
x, y = SpatialCoordinate(msh)
up0 = project(as_vector([0, 0, sin(pi*x)*sin(pi*y)]), Z)
u0, p0 = split(up0)
```

We build the variational form in UFL:

```
v, w = TestFunctions(Z)
F = inner(Dt(u0), v)*dx + inner(div(u0), w) * dx + inner(Dt(p0), w)*dx -␣
↪inner(p0, div(v)) * dx
```

Energy conservation is an important principle of the wave equation, and we can test how well the spatial discretization conserves energy by creating a UFL expression and evaluating it at each time step:

```
E = 0.5 * (inner(u0, u0)*dx + inner(p0, p0)*dx)
```

The time and time step variables:

```
MC = MeshConstant(msh)
t = MC.Constant(0.0)
dt = MC.Constant(1.0/N)
```

Here, we experiment with a multigrid preconditioner for the CG(2)-in-time discretization:

```
mgparams = {"mat_type": "aij",
            "snes_type": "ksponly",
            "ksp_type": "fgmres",
            "ksp_rtol": 1e-8,
            "pc_type": "mg",
            "mg_levels": {
                "ksp_type": "chebyshev",
                "ksp_max_it": 2,
                "ksp_convergence_test": "skip",
                "pc_type": "python",
                "pc_python_type": "firedrake.PatchPC",
                "patch": {
                    "pc_patch": {
                        "save_operators": True,
                        "partition_of_unity": False,
                        "construct_type": "star",
                        "construct_dim": 0,
                        "sub_mat_type": "seqdense",
                        "dense_inverse": True,
```

(continues on next page)

(continued from previous page)

```
                        "precompute_element_tensors": None},
                    "sub": {
                        "ksp_type": "preonly",
                        "pc_type": "lu"}}},
            "mg_coarse": {
                "pc_type": "lu",
                "pc_factor_mat_solver_type": "mumps"}
            }


stepper = GalerkinTimeStepper(F, 2, t, dt, up0,
                              solver_parameters=mgparams)
```

And, as with the heat equation, our time-stepping logic is quite simple. At easch time step, we print out the energy in the system:

```
print("Time    Energy")
print("==============")

while (float(t) < 1.0):
    if float(t) + float(dt) > 1.0:
        dt.assign(1.0 - float(t))

    stepper.advance()

    t.assign(float(t) + float(dt))
    print("{0:1.1e} {1:5e}".format(float(t), assemble(E)))
```

If all is well with the world, the energy will be nearly identical (up to roundoff error) at each time step because the Galerkin-in-time methods are symplectic and applied to a linear Hamiltonian system.

We can also confirm that the multigrid preconditioner is effective, by computing the average number of linear iterations per time-step:

```
(steps, nl_its, linear_its) = stepper.solver_stats()
print(f"The average number of multigrid iterations per time-step is {linear_
↪its/steps}.")
```

and for explicit schemes:

3.12 Solving the Mixed Wave Equation: Energy conservation and explicit RK

Let Ω be the unit square with boundary Γ. We write the wave equation as a first-order system of PDE:

$$\begin{aligned} u_t + \nabla p &= 0 \\ p_t + \nabla \cdot u &= 0 \end{aligned}$$

together with homogeneous Dirichlet boundary conditions

$$p = 0 \quad \text{on } \Gamma$$

In this form, at each time, u is a vector-valued function in the Sobolev space $H(\mathrm{div})$ and *p* is a scalar-valued function. If we select appropriate test functions v and w, then we can arrive at the weak form

$$\begin{aligned}(u_t, v) - (p, \nabla \cdot v) &= 0\\ (p_t, w) + (\nabla \cdot u, w) &= 0\end{aligned}$$

Note that in mixed formulations, the Dirichlet boundary condition is weakly enforced via integration by parts rather than strongly in the definition of the approximating space.

In this example, we will use the next-to-lowest order Raviart-Thomas elements for the velocity variable u and discontinuous piecewise linear polynomials for the scalar variable p.

Here is some typical Firedrake boilerplate and the construction of a simple mesh and the approximating spaces:

```
from firedrake import *
from irksome import Dt, MeshConstant, TimeStepper, PEPRK

N = 10

msh = UnitSquareMesh(N, N)
V = FunctionSpace(msh, "RT", 2)
W = FunctionSpace(msh, "DG", 1)
Z = V*W
```

Now we can build the initial condition, which has zero velocity and a sinusoidal displacement:

```
x, y = SpatialCoordinate(msh)
up0 = project(as_vector([0, 0, sin(pi*x)*sin(pi*y)]), Z)
u0, p0 = split(up0)
```

We build the variational form in UFL:

```
v, w = TestFunctions(Z)
F = inner(Dt(u0), v)*dx + inner(div(u0), w) * dx + inner(Dt(p0), w)*dx -␣
↪inner(p0, div(v)) * dx
```

Energy conservation is an important principle of the wave equation, and we can test how well the spatial discretization conserves energy by creating a UFL expression and evaluating it at each time step:

```
E = 0.5 * (inner(u0, u0)*dx + inner(p0, p0)*dx)
```

The time and time step variables:

```
MC = MeshConstant(msh)
t = MC.Constant(0.0)
dt = MC.Constant(0.2/N)
```

The PEP RK methods of de Leon, Ketcheson, and Ranoch offer explicit RK methods that preserve the energy up to a given order in step size. They have more stages than classical explicit methods but have much better energy conservation.:

```
butcher_tableau = PEPRK(4, 2, 5)
```

We'll use a simple iterative method to invert the mass matrix for each stage:

```
params = {"snes_type": "ksponly",
          "ksp_type": "cg",
          "pc_type": "icc"}

stepper = TimeStepper(F, butcher_tableau, t, dt, up0,
                      stage_type="explicit",
                      solver_parameters=params)
```

And, as with the heat equation, our time-stepping logic is quite simple. At each time step, we print out the energy in the system:

```
print("Time    Energy")
print("==============")

while (float(t) < 1.0):
    if float(t) + float(dt) > 1.0:
        dt.assign(1.0 - float(t))

    stepper.advance()

    t.assign(float(t) + float(dt))
    print("{0:1.1e} {1:5e}".format(float(t), assemble(E)))
```

If all is well with the world, the energy will be nearly identical (up to roundoff error) at each time step because the PEP methods conserve energy to quite high order. The reader can compare this to the mixed wave demo using Gauss-Legendre methods (which exactly conserve energy up to roundoff and solver tolerances.

and for Nystrom methods for second-order-in-time equations:

3.13 Solving the Wave Equation with Irksome

Let's start with the simple wave equation on $\Omega = [0,1] \times [0,1]$, with boundary Γ:

$$\begin{aligned} u_{tt} - \Delta u &= 0 \\ u &= 0 \quad \text{on } \Gamma \end{aligned}$$

At each time t, the solution to this equation will be some function $u \in V$, for a suitable function space V.

We transform this into weak form by multiplying by an arbitrary test function $v \in V$ and integrating over Ω. We know have the variational problem of finding $u : [0,T] \to V$ such that

$$(u_{tt}, v) + (\nabla u, \nabla v) = 0$$

As usual, we need to import firedrake:

```
from firedrake import *
```

We will also need to import certain items from irksome:

```
from irksome import GaussLegendre, Dt, MeshConstant,␣
↪StageDerivativeNystromTimeStepper
```

We will create the Butcher tableau for a Gauss-Legendre Runge-Kutta method, which generalizes the implicit midpoint rule:

```
ns = 2
butcher_tableau = GaussLegendre(ns)
```

Now we define the mesh and piecewise linear approximating space in standard Firedrake fashion:

```
N = 32

msh = UnitSquareMesh(N, N)
V = FunctionSpace(msh, "CG", 2)
```

We define variables to store the time step and current time value:

```
dt = Constant(2.0 / N)
t = Constant(0.0)
```

We define the initial condition for the fully discrete problem, which will get overwritten at each time step. For a second-order problem, we need an initial condition for the solution and its time derivative (in this case, taken to be zero):

```
x, y = SpatialCoordinate(msh)
uinit = sin(pi * x) * cos(pi * y)
u = Function(V)
u.interpolate(uinit)
ut = Function(V)
```

Now, we will define the semidiscrete variational problem using standard UFL notation, augmented by the `Dt` operator from Irksome. Here, the optional second argument indicates the number of derivatives, (defaulting to 1):

```
v = TestFunction(V)
F = inner(Dt(u, 2), v)*dx + inner(grad(u), grad(v))*dx
bc = DirichletBC(V, 0, "on_boundary")
```

Let's use simple solver options:

```
luparams = {"mat_type": "aij",
            "ksp_type": "preonly",
            "pc_type": "lu"}
```

Most of Irksome's magic happens in the *StageDerivativeNystromTimeStepper*. It takes our semidiscrete form F and the tableau and produces the variational form for computing the stage unknowns. Then, it sets up a variational problem to be solved for the stages at each time step.:

```
stepper = StageDerivativeNystromTimeStepper(
    F, butcher_tableau, t, dt, u, ut, bcs=bc,
    solver_parameters=luparams)
```

The system energy is an important quantity for the wave equation, and it should be conserved with our choice of time-stepping method:

```
E = 0.5 * (inner(ut, ut) * dx + inner(grad(u), grad(u)) * dx)
print(f"Initial energy: {assemble(E)}")
```

Then, we can loop over time steps, much like with 1st order systems:

```
while (float(t) < 1.0):
    if (float(t) + float(dt) > 1.0):
        dt.assign(1.0 - float(t))
    stepper.advance()
    t.assign(float(t) + float(dt))
    print(f"Time: {float(t)}, Energy: {assemble(E)}")
```

3.14 Solving the Wave Equation with Irksome and an explicit Nystrom method

Let's start with the simple wave equation on $\Omega = [0,1] \times [0,1]$, with boundary Γ:

$$\begin{aligned} u_{tt} - \Delta u &= 0 \\ u &= 0 \quad \text{on } \Gamma \end{aligned}$$

At each time t, the solution to this equation will be some function $u \in V$, for a suitable function space V.

We transform this into weak form by multiplying by an arbitrary test function $v \in V$ and integrating over Ω. We know have the variational problem of finding $u : [0,T] \to V$ such that

$$(u_{tt}, v) + (\nabla u, \nabla v) = 0$$

As usual, we need to import firedrake:

```
from firedrake import *
```

We will also need to import certain items from irksome:

```
from irksome import Dt, MeshConstant, StageDerivativeNystromTimeStepper,␣
↪ClassicNystrom4Tableau
```

Here, we will use the "classic" Nystrom method, a 4-stage explicit time-stepper:

```
nystrom_tableau = ClassicNystrom4Tableau()
```

Now we define the mesh and piecewise linear approximating space in standard Firedrake fashion:

```
N = 32

msh = UnitSquareMesh(N, N)
V = FunctionSpace(msh, "CG", 2)
```

We define variables to store the time step and current time value, noting that an explicit scheme requires a small timestep for stability:

```
dt = Constant(0.2 / N)
t = Constant(0.0)
```

We define the initial condition for the fully discrete problem, which will get overwritten at each time step. For a second-order problem, we need an initial condition for the solution and its time derivative (in this case, taken to be zero):

```
x, y = SpatialCoordinate(msh)
uinit = sin(pi * x) * cos(pi * y)
u = Function(V)
u.interpolate(uinit)
ut = Function(V)
```

Now, we will define the semidiscrete variational problem using standard UFL notation, augmented by the `Dt` operator from Irksome. Here, the optional second argument indicates the number of derivatives, (defaulting to 1):

```
v = TestFunction(V)
F = inner(Dt(u, 2), v)*dx + inner(grad(u), grad(v))*dx
bc = DirichletBC(V, 0, "on_boundary")
```

We're using an explicit scheme, so we only need to solve mass matrices in the update system. So, some good solver parameters here are to use a fieldsplit (so we only invert the diagonal blocks of the stage-coupled system) with incomplete Cholesky as a preconditioner for the mass matrices on the diagonal blocks:

```
mass_params = {"snes_type": "ksponly",
               "mat_type": "aij",
               "ksp_type": "preonly",
               "pc_type": "fieldsplit",
               "pc_fieldsplit_type": "multiplicative"}

per_field = {"ksp_type": "cg",
             "pc_type": "icc"}

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

Most of Irksome's magic happens in the *StageDerivativeNystromTimeStepper*. It takes our semidiscrete form *F* and the tableau and produces the variational form for computing the stage unknowns. Then, it sets up a variational problem to be solved for the stages at each time step. Here, we use *dDAE* style boundary conditions, which impose boundary conditions on the stages to match those of u_t on the boundary, consistent with the underlying partitioned scheme:

```
stepper = StageDerivativeNystromTimeStepper(
    F, nystrom_tableau, t, dt, u, ut, bcs=bc, bc_type="dDAE",
    solver_parameters=mass_params)
```

The system energy is an important quantity for the wave equation. It won't be conserved with our choice of time-stepping method, but serves as a good diagnostic:

```
E = 0.5 * (inner(ut, ut) * dx + inner(grad(u), grad(u)) * dx)
print(f"Initial energy: {assemble(E)}")
```

Then, we can loop over time steps, much like with 1st order systems:

```
while (float(t) < 1.0):
    if (float(t) + float(dt) > 1.0):
        dt.assign(1.0 - float(t))
    stepper.advance()
    t.assign(float(t) + float(dt))
    print(f"Time: {float(t)}, Energy: {assemble(E)}")
```

3.15 Solving the wave equation with monolithic multigrid

This reprise of the wave equation demo uses a monolithic multigrid algorithm to perform time advancement.

We consider the wave equation on $\Omega = [0, 1] \times [0, 1]$, with boundary Γ: giving rise to the weak form

$$(u_{tt}, v) + (\nabla u, \nabla v) = 0$$

We perform similar imports and setup as before:

```
from firedrake import *
from irksome import GaussLegendre, Dt, MeshConstant,␣
↪StageDerivativeNystromTimeStepper
butcher_tableau = GaussLegendre(2)
```

However, we need to set up a mesh hierarchy to enable geometric multigrid within Firedrake:

```
N = 4
nref = 3
base = UnitSquareMesh(N, N)
mh = MeshHierarchy(base, nref)
msh = mh[-1]
```

From here, setting up the function space, manufactured solution, etc, are just as for the regular wave equation demo:

```
V = FunctionSpace(msh, "CG", 2)

dt = Constant(2 / (N*2**nref))
t = Constant(0.0)
```

(continues on next page)

(continued from previous page)

```
x, y = SpatialCoordinate(msh)
uinit = sin(pi * x) * cos(pi * y)
u = Function(V)
u.interpolate(uinit)
ut = Function(V)

v = TestFunction(V)

F = inner(Dt(u, 2), v)*dx + inner(grad(u), grad(v))*dx
bc = DirichletBC(V, 0, "on_boundary")
```

And now for the solver parameters. Note that we are solving a block-wise system with all stages coupled together. This performs a monolithic multigrid with a stage-coupled vertex patch smoother:

```
mgparams = {"mat_type": "aij",
            "snes_type": "ksponly",
            "ksp_type": "gmres",
            "ksp_monitor_true_residual": None,
            "pc_type": "mg",
            "mg_levels": {
                "ksp_type": "chebyshev",
                "ksp_max_it": 1,
                "ksp_convergence_test": "skip",
                "pc_type": "python",
                "pc_python_type": "firedrake.ASMStarPC"},
            "mg_coarse": {
                "pc_type": "lu",
                "pc_factor_mat_solver_type": "mumps"}
            }
```

These solver parameters work just fine in the stepper.:

```
stepper = StageDerivativeNystromTimeStepper(
    F, butcher_tableau, t, dt, u, ut, bcs=bc,
    solver_parameters=mgparams)
```

The system energy is an important quantity for the wave equation, and it should be conserved with our choice of time-stepping method:

```
E = 0.5 * (inner(ut, ut) * dx + inner(grad(u), grad(u)) * dx)
print(f"Initial energy: {assemble(E)}")
```

And we can advance the solution in time in typical fashion:

```
while (float(t) < 1.0):
    if (float(t) + float(dt) > 1.0):
        dt.assign(1.0 - float(t))
    stepper.advance()
```

(continues on next page)

(continued from previous page)

```
    t.assign(float(t) + float(dt))
    print(f"Time: {float(t)}, Energy: {assemble(E)}")
```

3.16 Solving the telegraph equation with a stage-decoupled preconditioner

This demo solves a slightly more involved equation – the telegraph equation – and uses a stage-segregated preconditioner rather than direct method or monolithic multigrid.

We consider the telegraph equation on $\Omega = [0,1] \times [0,1]$, with boundary Γ: giving rise to the weak form

$$(u_{tt}, v) + (u_t, v) + (\nabla u, \nabla v) = 0$$

We perform similar imports and setup as before:

```
from firedrake import *
from irksome import GaussLegendre, Dt, MeshConstant,␣
↪StageDerivativeNystromTimeStepper
tableau = GaussLegendre(2)
```

We're going to use a stage-segregated preconditioner, and we have access to AMG for the fields. No need for a mesh hierarchy:

```
N = 32
msh = UnitSquareMesh(N, N)
```

From here, setting up the function space, manufactured solution, etc, are just as for the regular wave equation demo:

```
V = FunctionSpace(msh, "CG", 2)

dt = Constant(2 / N)
t = Constant(0.0)

x, y = SpatialCoordinate(msh)
uinit = sin(pi * x) * cos(pi * y)
u = Function(V)
u.interpolate(uinit)
ut = Function(V)

v = TestFunction(V)

F = inner(Dt(u, 2), v)*dx + inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx
bc = DirichletBC(V, 0, "on_boundary")
```

And now for the solver parameters. Note that we are solving a block-wise system with all stages coupled together. We will segregate those stages with the preconditioner from Clines/Howle/Long, and use hypre on the diagonal blocks:

```
params = {"mat_type": "aij",
          "snes_type": "ksponly",
          "ksp_type": "gmres",
          "ksp_monitor": None,
          "pc_type": "python",
          "pc_python_type": "irksome.ClinesLD",
          "aux": {
              "pc_type": "fieldsplit",
              "pc_fieldsplit_type": "multiplicative",
              "fieldsplit": {
                "ksp_type": "preonly",
                "pc_type": "hypre"
              }
          }}
```

These solver parameters work just fine in the stepper.:

```
stepper = StageDerivativeNystromTimeStepper(
    F, tableau, t, dt, u, ut, bcs=bc,
    solver_parameters=params)
```

The system energy is the same quantity as for the wave equation, but in the telegraph equation it decays exponentially over time instead of being conserved.:

```
E = 0.5 * (inner(ut, ut) * dx + inner(grad(u), grad(u)) * dx)
print(f"Initial energy: {assemble(E)}")
```

And we can advance the solution in time in typical fashion:

```
while (float(t) < 1.0):
    if (float(t) + float(dt) > 1.0):
        dt.assign(1.0 - float(t))
    stepper.advance()
    t.assign(float(t) + float(dt))
    print(f"Time: {float(t)}, Energy: {assemble(E)}")
```

and for bounds constraints:

3.17 Solving the Heat Equation with Bounds Constraints

In this demo we solve the simple heat equation with bounds constraints applied uniformly in time and space. Consider the simple heat equation on $\Omega = [0,1] \times [0,1]$, with boundary Γ:

$$\begin{aligned} u_t - \Delta u &= f \\ u &= g \quad \text{on } \Gamma \end{aligned}$$

for some known functions f and g. The solution will be some function $u \in V$, for a suitable function space V.

The weak form is found by multiplying by an arbitrary test function $v \in V$ and integrating over Ω. We then have the variational problem of finding $u : [0,T] \rightarrow V$ such that .. math:

```
(u_t, v) + (\nabla u, \nabla v) = (f, v)\quad \forall v \in V \textrm{ and }␣
↪t\in [0, T],
```

subject to the boundary condition $u = g$ on Γ. This demo uses particular choices of the functions f and g to be defined below.

The approach to bounds constraints below relies on the geometric properties of the Bernstein basis. In one dimension (on $[0, 1]$), the graph of the polynomial .. math:

```
p(x) = \sum_{i = 0}^n p_i b_i^n(x),
```

where $b_i^n(x)$ are Bernstein basis polynomials, lies in the convex-hull of the points .. math:

```
\left\{\left(\frac{i}{n}, p_i\right)\right\}_{i = 0}^n.
```

In particular, if the coefficients p_i lie in the interval $[m, M]$, then the output of $p(x)$ will also fall within this range. Similar results hold in higher dimensions. This property provides a straightforward approach to uniformly enforced bounds constraints in both space and time.

First, we must import firedrake and certain items from Irksome:

```
from firedrake import *
from irksome import Dt, MeshConstant, RadauIIA, TimeStepper,␣
↪BoundsConstrainedDirichletBC
```

Finally, numpy provides us with the upper bound of infinity:

```
import numpy as np
```

We first define the mesh and the necessary function space. We choose quadratic Bernstein polynomials to support bounds-constraints in space:

```
N = 32

msh = UnitSquareMesh(N, N)
V = FunctionSpace(msh, "Bernstein", 2)
```

In order to enforce bounds constraints in time, we must utilize a collocation method. In this demo, we will time-step using the L-stable, fully implicit, 2-stage RadauIIA Runge-Kutta method. The bounds will be passed as an argument to the `advance` method. We now define the Butcher Tableau and variables to store the time step, current time, and final time:

```
butcher_tableau = RadauIIA(2)

MC = MeshConstant(msh)
dt = MC.Constant(2 / N)
t = MC.Constant(0.0)
Tf = MC.Constant(1.0)
```

We will find an approximate solution at time $t = 1.0$ with and without enforcing a constraint on the lower bound. We will need the following pair of solver parameters:

```
lu_params = {
    "snes_type": "ksponly",
    "ksp_type": "preonly",
    "mat_type": "aij",
    "pc_type": "lu"
}

vi_params = {
    "snes_type": "vinewtonrsls",
    "snes_max_it": 300,
    "snes_atol": 1.e-8,
    "ksp_type": "preonly",
    "mat_type": "aij",
    "pc_type": "lu",
}
```

We now define the right-hand side using the method of manufactured solutions:

```
x, y = SpatialCoordinate(msh)

uexact = 0.5 * exp(-t) * (1 + (tanh((0.1 - sqrt((x - 0.5) ** 2 + (y - 0.5) **␣
↪2)) / 0.015)))

rhs = Dt(uexact) - div(grad(uexact))
```

Note that the exact solution is uniformly positive in space and time. Using a manufactured solution, one usually interpolates or projects the exact solution at time $t = 0$ onto the approximation space to obtain the initial condition. Interpolation does not work with the Bernstein basis, and there is no guarantee that an interpolant or projection would satisfy the bounds constraints. To guarantee that the initial condition satisfies the bounds constraints, we solve a variational inequality:

```
v = TestFunction(V)
u_init = Function(V)

G = inner(u_init - uexact, v) * dx

nlvp = NonlinearVariationalProblem(G, u_init)
nlvs = NonlinearVariationalSolver(nlvp, solver_parameters=vi_params)

lb = Function(V)
ub = Function(V)

ub.assign(np.inf)
lb.assign(0.0)

nlvs.solve(bounds=(lb, ub))

u = Function(V)
u.assign(u_init)
```

(continues on next page)

(continued from previous page)

```
u_c = Function(V)
u_c.assign(u_init)
```

u and u_c now hold a bounds-constrained approximation to the exact solution at $t = 0$. Note that *ub* = *None* is also supported and gets internally converted to what we have here.

We now construct semidiscrete variational problems for both the constrained and unconstrained approximations using UFL notation and the Dt operator from Irksome:

```
v = TestFunction(V)

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

v_c = TestFunction(V)

F_c = (inner(Dt(u_c), v_c) * dx + inner(grad(u_c), grad(v_c)) * dx -␣
↪inner(rhs, v_c) * dx)
```

We use exact boundary conditions in both cases. When g is the trace of a function defined over the whole domain, Firedrake creates its own version of the boundary condition by either interpolating or projecting that function onto the finite element space and computing the trace of the result. To ensure the internal boundary condition satisfies the bounds constraints, we will pass the bounds to the TimeStepper below.

```
bc = DirichletBC(V, uexact, "on_boundary")
```

For the unconstrained approximation, we configure the TimeStepper in a familiar way:

```
stepper = TimeStepper(F, butcher_tableau, t, dt, u, bcs=bc, solver_
↪parameters=lu_params)
```

We will enforce nonnegativity when finding the constrained approximation. We now set up the keyword database to configure an instance of TimeStepper for this task. We first specify, using the keyword stage_type, that we wish to use a stage-value formulation of the underlying collocation method. The keyword basis_type then allows us to change the basis of the collocation polynomial to the Bernstein basis. Having done this, we must specify a solver which is able to handle bounds constraints. In this example we solve a variational inequality using vinewtonrsls by passing vi_params as solver_parameters to the TimeStepper.

We set the bounds as follows (reusing those defined in the initial condition):

```
bounds = ('stage', lb, ub)
```

Internally, Firedrake will project the boundary condition expression into the entire space and match degrees of freedom on the boundary. This could introduce bounds violations. To ensure this does not happen, we can use a special kind of boundary condition that projects with bounds contraints.

```
bc = BoundsConstrainedDirichletBC(V, uexact, "on_boundary", (lb, ub), solver_
↪parameters=vi_params)
```

(continues on next page)

(continued from previous page)

```
kwargs_c = {"bounds": bounds,
            "stage_type": "value",
            "basis_type": 'Bernstein',
            "solver_parameters": vi_params
        }

stepper_c = TimeStepper(F_c, butcher_tableau, t, dt, u_c, bcs=bc, **kwargs_c)
```

Note that if one does not set the `basis_type` to Bernstein, the standard basis will be used. Solving for the Bernstein coefficients of the collocation polynomial we obtain uniform-in-time bounds constraints. If the standard basis is used, the bounds constraints are guaranteed at the Runge-Kutta stages and the discrete times, but not necessarily between them.

When using a stage-value formulation, passing `bounds` to the `TimeStepper` through the `advance` method will enforce the bounds constraints at the discrete stages and time levels (this results in uniformly enforced constraints when using the Bernstein basis).

We now advance both semidiscrete systems in the usual way. We add the bounds as an argument to the `advance` method for the constrained approximation.

In order to monitor our approximate solutions, we check the minimum value of each after every step in time. If an approximate solution violates the lower bound, we append a tuple to indicate the time and minimum value.

```
violations_for_unconstrained_method = []
violations_for_constrained_method = []

timestep = 0
while (float(t) < float(Tf)):

    if (float(t) + float(dt) > float(Tf)):
        dt.assign(float(Tf) - float(t))

    stepper.advance()
    stepper_c.advance()

    t.assign(float(t) + float(dt))
    timestep = timestep + 1

    min_value = min(u.dat.data)
    if min_value < 0:
        violations_for_unconstrained_method.append((float(t), timestep,␣
↪round(min_value, 3)))

    min_value_c = min(u_c.dat.data)
    if min_value_c < 0:
        violations_for_constrained_method.append((float(t), timestep,␣
↪round(min_value_c, 3)))

    print(float(t))
```

Finally, we print the relative L^2 error and the time and severity (if any) of constraint violations:

```
np.set_printoptions(legacy='1.25')

print()
print(f"Relative L^2 norm of the unconstrained solution: {norm(u - uexact) /␣
↪norm(uexact)}")
print(f"Relative L^2 norm of the constrained solution:   {norm(u_c - uexact) /
↪ norm(uexact)}")
print()
print("List of constraint violations in the form (time, time step, minimum␣
↪value) for each approximation:")
print()
print(f"Unconstrained solution: {violations_for_unconstrained_method}")
print()
print(f"Constrained solution: {violations_for_constrained_method}")
```

and for adaptive IRK methods:

3.18 Solving the Heat Equation with Irksome

Let's start with the simple heat equation on $\Omega = [0, 10] \times [0, 10]$, with boundary Γ:

$$\begin{aligned} u_t - \Delta u &= f \\ u &= 0 \quad \text{on } \Gamma \end{aligned}$$

for some known function f. At each time t, the solution to this equation will be some function $u \in V$, for a suitable function space V.

We transform this into weak form by multiplying by an arbitrary test function $v \in V$ and integrating over Ω. We know have the variational problem of finding $u : [0, T] \to V$ such that

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

This demo implements an example used by Solin with a particular choice of f given below

As usual, we need to import firedrake:

```
from firedrake import *
```

We will also need to import certain items from irksome:

```
from irksome import RadauIIA, Dt, MeshConstant, TimeStepper
```

We will create the Butcher tableau for the 3-stage RadauIIA Runge-Kutta method, which has an embedded scheme we can use for adaptivity:

```
butcher_tableau = RadauIIA(3)
ns = butcher_tableau.num_stages
```

Now we define the mesh and piecewise linear approximating space in standard Firedrake fashion:

```
N = 100
x0 = 0.0
x1 = 10.0
y0 = 0.0
y1 = 10.0

msh = RectangleMesh(N, N, x1, y1)
V = FunctionSpace(msh, "CG", 1)
```

We define variables to store the time step, current time value, and final time:

```
MC = MeshConstant(msh)
dt = MC.Constant(10.0 / N)
t = MC.Constant(0.0)
Tf = MC.Constant(1.0)
```

This defines the right-hand side using the method of manufactured solutions:

```
x, y = SpatialCoordinate(msh)
S = Constant(2.0)
C = Constant(1000.0)
B = (x-Constant(x0))*(x-Constant(x1))*(y-Constant(y0))*(y-Constant(y1))/C
R = (x * x + y * y) ** 0.5
uexact = B * atan(t)*(pi / 2.0 - atan(S * (R - t)))
rhs = Dt(uexact) - div(grad(uexact))
```

We define the initial condition for the fully discrete problem, which will get overwritten at each time step:

```
u = Function(V)
u.interpolate(uexact)
```

Now, we will define the semidiscrete variational problem using standard UFL notation, augmented by the `Dt` operator from Irksome:

```
v = TestFunction(V)
F = inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx - inner(rhs, v)*dx
bc = DirichletBC(V, 0, "on_boundary")
```

Later demos will show how to use Firedrake's sophisticated interface to PETSc for efficient block solvers, but for now, we will solve the system with a direct method (Note: the matrix type needs to be explicitly set to aij if sparse direct factorization were used with a multi-stage method, as we wind up with a mixed problem that Firedrake will assemble into a PETSc MatNest otherwise):

```
luparams = {"mat_type": "aij",
            "ksp_type": "preonly",
            "pc_type": "lu"}
```

Most of Irksome's magic happens in the `TimeStepper`. It transforms our semidiscrete form *F* into a fully discrete form for the stage unknowns and sets up a variational problem to solve for the stages at each time step.

In this demo, we use an adaptive timestepper, selected by sending a dictionary of adaptive parameters, which tells Irksome to also compute an error estimate at each timestep, and use that estimate to adjust the timestep size. The adaptation depends on some given parameters, including a tolerance *tol* for for the error at each step, and *KI* and *KP* that set the so-called integration and performance gain for the estimate.:

```
adapt_params = {"tol":1e-3, "KI":1/15, "KP":0.13}
stepper = TimeStepper(F, butcher_tableau, t, dt, u, bcs=bc,
                      stage_type="deriv", solver_parameters=luparams,
                      adaptive_parameters = adapt_params)
```

This logic is pretty self-explanatory. We use the `TimeStepper`'s `advance` method, which solves the variational problem to compute the Runge-Kutta stage values and then updates the solution.

Here, in contrast to the non-adaptive case, we get an estimate of the error at each step (that we do not use here) and a new adaptive timestep size at each step. We use these to control integrating to a fixed final time, *Tf*. This exposes the *dt_max* data for `TimeStepper`, which puts a hard limit on the timestep size in the adaptive case.:

```
while (float(t) < float(Tf)):
    stepper.dt_max = float(Tf)-float(t)
    (adapt_error, adapt_dt) = stepper.advance()
    print(float(t))
    t.assign(float(t) + float(adapt_dt))
```

Finally, we print out the relative L^2 error:

```
print()
print(norm(u-uexact)/norm(uexact))
```

Or check out two IMEX-type methods for the monodomain equations:

3.19 Solving monodomain equations with Fitzhugh-Nagumo reaction and an IMEX method

We're solving monodomain (reaction-diffusion) with a particular reaction term. The basic form of the equation is:

$$\chi \left(C_m u_t + I_{ion}(u) \right) = \nabla \cdot \sigma \nabla u$$

where u is the membrane potential, σ is the conductivity tensor, C_m is the specific capacitance of the cell membrane, and χ is the surface area to volume ratio. The term I_{ion} is current due to ionic flows through channels in the cell membranes, and may couple to a complicated reaction network. In our case, we take the relatively simple model due to Fitzhugh and Nagumo. Here, we have a separate concentration variable c satisfying the reaction equation:

$$c_t = \epsilon(u + \beta - \gamma c)$$

for certain positive parameters β and γ, and the current takes the form of:

$$I_{ion}(u, c) = \tfrac{1}{\epsilon} \left(u - \tfrac{u^3}{3} - c \right)$$

so that we have an overall system of two equations. One of them is linear but stiff/diffusive, and the other is nonstiff but nonlinear. This combination makes the system a good candidate for additive partitioning/IMEX-type methods.

We start with standard Firedrake/Irksome imports:

```
import copy

from firedrake import (And, Constant, File, Function, FunctionSpace,
                       RectangleMesh, SpatialCoordinate, TestFunctions,
                       as_matrix, conditional, dx, grad, inner, split)
from irksome import Dt, MeshConstant, RadauIIA, TimeStepper
```

And we set up the mesh and function space. Note this demo uses serendipity elements, but could just as easily use Lagrange on quads or triangles.:

```
mesh = RectangleMesh(20, 20, 70, 70, quadrilateral=True)
polyOrder = 2

V = FunctionSpace(mesh, "S", polyOrder)
Z = V * V

x, y = SpatialCoordinate(mesh)
MC = MeshConstant(mesh)
dt = MC.Constant(0.05)
t = MC.Constant(0.0)
```

Specify the physical constants and initial conditions:

```
eps = Constant(0.1)
beta = Constant(1.0)
gamma = Constant(0.5)

chi = Constant(1.0)
capacitance = Constant(1.0)

sigma1 = sigma2 = 1.0
sigma = as_matrix([[sigma1, 0.0], [0.0, sigma2]])


initial_potential = conditional(x < 3.5, Constant(2.0), Constant(-1.28791))
initial_cell = conditional(And(And(31 <= x, x < 39), And(0 <= y, y < 35)),
                           Constant(2.0), Constant(-0.5758))


uu = Function(Z)
vu, vc = TestFunctions(Z)
uu.sub(0).interpolate(initial_potential)
uu.sub(1).interpolate(initial_cell)

(u, c) = split(uu)
```

This sets up the Butcher tableau. All of our IMEX-type methods are based on a RadauIIA method for the implicit part. We use a two-stage method.:

```
butcher_tableau = RadauIIA(2)
ns = butcher_tableau.num_stages
```

To access an IMEX method, we need to separately specify the implicit and explicit parts of the operator. The part to be handled implicitly is taken to contain the time derivatives as well:

```
F1 = (inner(chi * capacitance * Dt(u), vu)*dx
      + inner(grad(u), sigma * grad(vu))*dx
      + inner(Dt(c), vc)*dx - inner(eps * u, vc)*dx
      - inner(beta * eps, vc)*dx + inner(gamma * eps * c, vc)*dx)
```

This is the part to be additively partitioned and handled explicitly:

```
F2 = inner((chi/eps) * (-u + (u**3 / 3) + c), vu)*dx
```

If we wanted to use a fully implicit method, we would just take F = F1 + F2. Now, set up solver parameters. For this problem, the Rana preconditioner applied with a multiplicative field split works very well.:

```
params = {"snes_type": "ksponly",
          "ksp_rtol": 1.e-8,
          "ksp_monitor": None,
          "mat_type": "matfree",
          "ksp_type": "fgmres",
          "pc_type": "python",
          "pc_python_type": "irksome.RanaLD",
          "aux": {
              "pc_type": "fieldsplit",
              "pc_fieldsplit_type": "multiplicative"}}
```

Each diagonal block to be solved is itself a block-coupled system, with a diffusive block and a mass matrix on the diagonal. Hence, we further fieldsplit each diagonal block, using algebraic multigrid for the diffusive block and incomplete Cholesky for the mass matrix.:

```
per_stage = {
    "ksp_type": "preonly",
    "pc_type": "fieldsplit",
    "pc_fieldsplit_type": "additive",
    "fieldsplit_0": {
        "ksp_type": "preonly",
        "pc_type": "gamg",
    },
    "fieldsplit_1": {
        "ksp_type": "preonly",
        "pc_type": "icc",
    }}
```

This bit of mess specifies how the overall system is split up. Each stage corresponds to a pair of fields (potential and concentration):

```
for s in range(ns):
    params["aux"][f"pc_fieldsplit_{s}_fields"] = f"{2*s},{2*s+1}"
    params["aux"][f"fieldsplit_{s}"] = per_stage
```

The partitioned IMEX methods also provide an “iterator” that is used both to start the method (filling in the stage values between the initial condition and first time step taken) and can also be used after a time step to improve the accuracy/stability of the solution. If the iterator “works”, applying it a large number of times will tend to produce the solution to a fully implicit RadauIIA method. In practice, we can sometimes get away with applying the iterator to a looser tolerance, but we otherwise use the same method as for the propagator.:

```
itparams = copy.deepcopy(params)
itparams["ksp_rtol"] = 1.e-4
```

Now, we access the IMEX method via the *TimeStepper* as with other methods. Note that we specify somewhat different kwargs, needing to specify the implicit and explicit parts separately as well as separate solver options for propagator and iterator.:

```
stepper = TimeStepper(F1, butcher_tableau, t, dt, uu,
                      stage_type="imex",
                      prop_solver_parameters=params,
                      it_solver_parameters=itparams,
                      Fexp=F2,
                      num_its_initial=5,
                      num_its_per_step=3)

uFinal, cFinal = uu.subfunctions
outfile1 = File("FHN_results/FHN_2d_u.pvd")
outfile2 = File("FHN_results/FHN_2d_c.pvd")
outfile1.write(uFinal, time=0)
outfile2.write(cFinal, time=0)

for j in range(12):
    print(f"{float(t)}")
    stepper.advance()
    t.assign(float(t) + float(dt))

    if (j % 5 == 0):
        outfile1.write(uFinal, time=j * float(dt))
        outfile2.write(cFinal, time=j * float(dt))

nsteps, nprop, nit, nnonlinprop, nlinprop, nnonlinit, nlinit = stepper.solver_
↪stats()
print(f"Time steps taken: {nsteps}")
print(f"  {nprop} propagator steps")
print(f"  {nit} iterator steps")
print(f"  {nnonlinprop} nonlinear steps in propagators")
print(f"  {nlinprop} linear steps in propagators")
print(f"  {nnonlinit} nonlinear steps in iterators")
print(f"  {nlinit} linear steps in iterators")
```

3.20 Solving monodomain equations with Fitzhugh-Nagumo reaction and a DIRK-IMEX method

We're solving monodomain (reaction-diffusion) with a particular reaction term. The basic form of the equation is:

$$\chi \left(C_m u_t + I_{ion}(u) \right) = \nabla \cdot \sigma \nabla u$$

where u is the membrane potential, σ is the conductivity tensor, C_m is the specific capacitance of the cell membrane, and χ is the surface area to volume ratio. The term I_{ion} is current due to ionic flows through channels in the cell membranes, and may couple to a complicated reaction network. In our case, we take the relatively simple model due to Fitzhugh and Nagumo. Here, we have a separate concentration variable c satisfying the reaction equation:

$$c_t = \epsilon(u + \beta - \gamma c)$$

for certain positive parameters β and γ, and the current takes the form of:

$$I_{ion}(u, c) = \frac{1}{\epsilon} \left(u - \frac{u^3}{3} - c \right)$$

so that we have an overall system of two equations. One of them is linear but stiff/diffusive, and the other is nonstiff but nonlinear. This combination makes the system a good candidate for IMEX-type methods.

We start with standard Firedrake/Irksome imports:

```
import copy

from firedrake import (And, Constant, File, Function, FunctionSpace,
                       RectangleMesh, SpatialCoordinate, TestFunctions,
                       as_matrix, conditional, dx, grad, inner, split)
from irksome import Dt, MeshConstant, TimeStepper, ARS_DIRK_IMEX
from irksome.labeling import explicit
```

And we set up the mesh and function space.:

```
mesh = RectangleMesh(20, 20, 70, 70, quadrilateral=True)
polyOrder = 2

V = FunctionSpace(mesh, "CG", 2)
Z = V * V

x, y = SpatialCoordinate(mesh)
MC = MeshConstant(mesh)
dt = MC.Constant(0.05)
t = MC.Constant(0.0)
```

Specify the physical constants and initial conditions:

```
eps = Constant(0.1)
beta = Constant(1.0)
gamma = Constant(0.5)
```

(continues on next page)

(continued from previous page)

```
chi = Constant(1.0)
capacitance = Constant(1.0)

sigma1 = sigma2 = 1.0
sigma = as_matrix([[sigma1, 0.0], [0.0, sigma2]])


initial_potential = conditional(x < 3.5, Constant(2.0), Constant(-1.28791))
initial_cell = conditional(And(And(31 <= x, x < 39), And(0 <= y, y < 35)),
                           Constant(2.0), Constant(-0.5758))


uu = Function(Z)
vu, vc = TestFunctions(Z)
uu.sub(0).interpolate(initial_potential)
uu.sub(1).interpolate(initial_cell)

(u, c) = split(uu)
```

This sets up the Butcher tableau. Here, we use the DIRK-IMEX methods proposed by Ascher, Ruuth, and Spiteri in their 1997 Applied Numerical Mathematics paper. For this case, We use a four-stage method.:

```
butcher_tableau = ARS_DIRK_IMEX(4, 4, 3)
ns = butcher_tableau.num_stages
```

To access an IMEX method, we need to separately specify the implicit and explicit parts of the operator. The part to be handled implicitly is taken to contain the time derivatives as well:

```
F1 = (inner(chi * capacitance * Dt(u), vu)*dx
      + inner(grad(u), sigma * grad(vu))*dx
      + inner(Dt(c), vc)*dx - inner(eps * u, vc)*dx
      - inner(beta * eps, vc)*dx + inner(gamma * eps * c, vc)*dx)
```

This is the part to be handled explicitly.:

```
F2 = inner((chi/eps) * (-u + (u**3 / 3) + c), vu)*dx
```

If we wanted to use a fully implicit method, we would just take F = F1 + F2. Instead, we use a label:

```
F = F1 + explicit(F2)
```

Now, set up solver parameters. Since we're using a DIRK-IMEX scheme, we can specify only parameters for each stage. We use an additive Schwarz (fieldsplit) method that applies AMG to the potential block and incomplete Cholesky to the cell block independently for each stage:

```
params = {"snes_type": "ksponly",
          "ksp_monitor": None,
```

(continues on next page)

(continued from previous page)

```
          "mat_type": "aij",
          "ksp_type": "fgmres",
          "pc_type": "fieldsplit",
          "pc_fieldsplit_type": "additive",
          "fieldsplit_0": {
              "ksp_type": "preonly",
              "pc_type": "gamg",
          },
          "fieldsplit_1": {
              "ksp_type": "preonly",
              "pc_type": "icc",
          }}
```

The DIRK-IMEX schemes also require a mass-matrix solver. Here, we just use an incomplete Cholesky preconditioner for CG on the coupled system, which works fine.:

```
mass_params = {"snes_type": "ksponly",
               "ksp_rtol": 1.e-8,
               "ksp_monitor": None,
               "mat_type": "aij",
               "ksp_type": "cg",
               "pc_type": "icc",
              }
```

Now, we access the IMEX method via the *TimeStepper* as with other methods. Note that we specify somewhat different kwargs, needing to specify the implicit and explicit parts separately as well as separate solver options for the implicit and mass solvers.:

```
stepper = TimeStepper(F1, butcher_tableau, t, dt, uu,
                      stage_type="dirkimex",
                      solver_parameters=params,
                      mass_parameters=mass_params,
                      Fexp=F2)

uFinal, cFinal = uu.subfunctions
outfile1 = File("FHN_results/FHN_2d_u.pvd")
outfile2 = File("FHN_results/FHN_2d_c.pvd")
outfile1.write(uFinal, time=0)
outfile2.write(cFinal, time=0)

for j in range(12):
    print(f"{float(t)}")
    stepper.advance()
    t.assign(float(t) + float(dt))

    if (j % 5 == 0):
        outfile1.write(uFinal, time=j * float(dt))
        outfile2.write(cFinal, time=j * float(dt))
```

We can print out some solver statistics here. We expect one implicit solve per stage per timestep, and that's what we see with the four-stage method. For this Butcher Tableau, we can avoid computing the final explicit stage (since it's coefficient in the next stage reconstruction is zero), so we see the same number of mass solves.:

```
nsteps, n_nonlin, n_lin, n_nonlin_mass, n_lin_mass = stepper.solver_stats()
print(f"Time steps taken: {nsteps}")
print(f"  {n_nonlin} nonlinear steps in implicit stage solves (should be
↪{nsteps*ns})")
print(f"  {n_lin} linear steps in implicit stage solves")
print(f"  {n_nonlin_mass} nonlinear steps in mass solves (should be
↪{nsteps*ns})")
print(f"  {n_lin_mass} linear steps in mass solves")
```

3.21 Advanced demos

There's also an example solving a Sobolev-type equation with symplectic RK methods:

3.21.1 Solving the Benjamin-Bona-Mahony equation

This demo solves the Benjamin-Bona-Mahony equation:

$$u_t + u_x + u u_x - u_{txx} = 0$$

typically posed on $\mathbb{R}$ or a bounded interval with periodic boundaries.

It is interesting because it is nonlinear ($u u_x$) and also a Sobolev-type equation, with spatial derivatives acting on a time derivative. We can obtain a weak form in the standard way:

$$(u_t, v) + (u_x, v) + (u u_x, v) + (u_{tx}, v_x) = 0$$

BBM is known to have a Hamiltonian structure, and there are three canonical polynomial invariants:

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

We are mainly interested in accuracy and in conserving these quantities reasonably well.

Firedrake imports:

```
from firedrake import *
from irksome import Dt, GaussLegendre, MeshConstant, TimeStepper
```

This function seems to be left out of UFL, but solitary wave solutions for BBM need it:

```
def sech(x):
    return 2 / (exp(x) + exp(-x))
```

Set up problem parameters, etc:

```
N = 1000
L = 100
h = L / N
msh = PeriodicIntervalMesh(N, L)

MC = MeshConstant(msh)
t = MC.Constant(0)
dt = MC.Constant(10*h)

x, = SpatialCoordinate(msh)
```

Here is the true solution, which is right-moving solitary wave (but not a soliton):

```
c = Constant(0.5)

center = 30.0
delta = -c * center

uexact = 3 * c**2 / (1-c**2) * sech(0.5 * (c * x - c * t / (1 - c ** 2) +␣
↪delta))**2
```

Create the approximating space and project true solution:

```
V = FunctionSpace(msh, "CG", 1)
u = project(uexact, V)
v = TestFunction(V)
```

The symplectic Gauss-Legendre methods are of interest here:

```
butcher_tableau = GaussLegendre(2)

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)
```

For a 1d problem, we don't worry much about efficient solvers.:

```
params = {"mat_type": "aij",
          "ksp_type": "preonly",
          "pc_type": "lu"}

stepper = TimeStepper(F, butcher_tableau, t, dt, u,
                      solver_parameters=params)
```

UFL for the mathematical invariants and containers to track them over time:

```
I1 = u * dx
I2 = (u**2 + (u.dx(0))**2) * dx
I3 = ((u.dx(0))**2 - u**3 / 3) * dx
```

(continues on next page)

(continued from previous page)

```
I1s = []
I2s = []
I3s = []
```

Time-stepping loop, keeping track of I values:

```
tfinal = 18.0
while (float(t) < tfinal):
    if float(t) + float(dt) > tfinal:
        dt.assign(tfinal - float(t))
    stepper.advance()

    I1s.append(assemble(I1))
    I2s.append(assemble(I2))
    I3s.append(assemble(I3))

    print('%.15f %.15f %.15f %.15f' % (float(t), I1s[-1], I2s[-1], I3s[-1]))
    t.assign(float(t) + float(dt))

print(errornorm(uexact, u) / norm(uexact))
```

and with a Galerkin-in-Time approach, in standard form or with auxiliary variables for alternate conservation properties:

3.21.2 Solving the Benjamin-Bona-Mahony equation with Galerkin-in-Time

This demo solves the Benjamin-Bona-Mahony equation:

$$u_t + u_x + uu_x - u_{txx} = 0$$

typically posed on $\mathbb{R}$ or a bounded interval with periodic boundaries.

It is interesting because it is nonlinear (uu_x) and also a Sobolev-type equation, with spatial derivatives acting on a time derivative. We can obtain a weak form in the standard way:

$$(u_t, v) + (u_x, v) + (uu_x, v) + (u_{tx}, v_x) = 0$$

BBM is known to have a Hamiltonian structure, and there are three canonical polynomial invariants:

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

We are mainly interested in accuracy and in conserving these quantities reasonably well.

Firedrake imports:

```
from firedrake import *
from irksome import Dt, GalerkinTimeStepper, MeshConstant
```

This function seems to be left out of UFL, but solitary wave solutions for BBM need it:

```
def sech(x):
    return 2 / (exp(x) + exp(-x))
```

Set up problem parameters, etc:

```
N = 1000
L = 100
h = L / N
msh = PeriodicIntervalMesh(N, L)

MC = MeshConstant(msh)
t = MC.Constant(0)
dt = MC.Constant(10*h)

x, = SpatialCoordinate(msh)
```

Here is the true solution, which is right-moving solitary wave (but not a soliton):

```
c = Constant(0.5)

center = 30.0
delta = -c * center

uexact = 3 * c**2 / (1-c**2) * sech(0.5 * (c * x - c * t / (1 - c ** 2) +␣
↪delta))**2
```

Create the approximating space and project true solution:

```
V = FunctionSpace(msh, "CG", 1)
u = project(uexact, V)
v = TestFunction(V)

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)
```

For a 1d problem, we don't worry much about efficient solvers.:

```
params = {"mat_type": "aij",
          "ksp_type": "preonly",
          "pc_type": "lu"}
```

The Galerkin-in-Time approach should have symplectic properties:

```
stepper = GalerkinTimeStepper(F, 2, t, dt, u,
                              solver_parameters=params)
```

UFL for the mathematical invariants and containers to track them over time:

```
I1 = u * dx
I2 = (u**2 + (u.dx(0))**2) * dx
I3 = ((u.dx(0))**2 - u**3 / 3) * dx

I1s = []
I2s = []
I3s = []
```

Time-stepping loop, keeping track of I values:

```
tfinal = 18.0
while (float(t) < tfinal):
    if float(t) + float(dt) > tfinal:
        dt.assign(tfinal - float(t))
    stepper.advance()

    I1s.append(assemble(I1))
    I2s.append(assemble(I2))
    I3s.append(assemble(I3))

    print('%.15f %.15f %.15f %.15f' % (float(t), I1s[-1], I2s[-1], I3s[-1]))
    t.assign(float(t) + float(dt))

print(errornorm(uexact, u) / norm(uexact))
```

3.21.3 Hamiltonian-structure-preserving implementation of the Benjamin-Bona-Mahoney equation

This demo solves the Benjamin-Bona-Mahony equation:

$$u_t - u_{txx} + u_x + uu_x = 0$$

posed on a bounded interval with periodic boundaries.

BBM is known to have a Hamiltonian structure, and there are several canonical polynomial invariants:

$$\begin{aligned} I_1 &= \int u \, \mathrm{d}x \\ I_2 &= \int u^2 + (u_x)^2 \, \mathrm{d}x \\ I_3 &= \int \frac{u^2}{2} + \frac{u^3}{6} \, \mathrm{d}x \end{aligned}$$

The BBM invariants are the total momentum I_1, the H^1-energy norm I_2, and the Hamiltonian I_3. The Hamiltonian variational formulation reads

$$\begin{aligned} (\partial_t u + \partial_x \tilde{w}_H, v)_{H^1} = 0 \\ (\tilde{w}_H, v_H)_{H^1} = \langle \frac{\delta I_3}{\delta u}, v_H \rangle \end{aligned}$$

For all test functions v, v_H in a suitable function space. The numerical scheme in this demo introduces the H^1-Riesz representative of the Fréchet derivative of the Hamiltonian $\frac{\delta I_3}{\delta u}$ as the auxiliary variable $\tilde{w}_H$.

Standard Gauss-Legendre and continuous Petrov-Galerkin (cPG) methods conserve the first two invariants exactly (up to roundoff and solver tolerances). They do quite well, but are inexact for the cubic one. Here, we consider the reformulation in Andrews and Farrell, "Enforcing conservation laws and dissipation inequalities numerically via auxiliary variables" (arXiv:2407.11904, to appear in SIAM J. Scientific Computing) that preserves the third invariant at the expense of the second. This method has an auxiliary variable in the system and requires a continuously differentiable spatial discretization (1D Hermite elements in this case). The time discretization puts the main unknown in a continuous space and the auxiliary variable in a discontinuous one. See equation (7.17) of Boris Andrews' thesis for the particular formulation.

Firedrake, Irksome, and other imports:

```
from firedrake import (Constant, Function, FunctionSpace,
    PeriodicIntervalMesh, SpatialCoordinate, TestFunction, TrialFunction,
    assemble, derivative, dx, errornorm, exp, grad, inner,
    interpolate, norm, plot, project, replace, solve, split
)

from irksome import Dt, GalerkinTimeStepper, TimeQuadratureLabel

import matplotlib.pyplot as plt
import numpy
```

Next, we define the domain and the exact solution

```
N = 8000
L = 100
h = L / N
msh = PeriodicIntervalMesh(N, L)
x, = SpatialCoordinate(msh)

t = Constant(0)
inv_dt = N // (10 * L)
tfinal = 18
Nt = tfinal * inv_dt
dt = Constant(tfinal / Nt)

c = Constant(0.5)
center = Constant(40.0)
delta = -c * center

def sech(x):
    return 2 / (exp(x) + exp(-x))

uexact = 3 * c**2 / (1-c**2) \
    * sech(0.5 * (c * x - c * t / (1 - c ** 2) + delta))**2
```

This sets up the function space for the unknown u and auxiliary variable $\tilde{w}_H$:

```
space_deg = 3
time_deg = 1
```

(continues on next page)

(continued from previous page)

```
V = FunctionSpace(msh, "Hermite", space_deg)
Z = V * V
```

We next define the BBM invariants. Again, the discrete formulation preserves I_1 and I_3 up to solver tolerances and roundoff errors, but I_2 is preserved up to a bounded oscillation

```
def h1inner(u, v):
    return inner(u, v) + inner(grad(u), grad(v))

def I1(u):
    return u * dx

def I2(u):
    return h1inner(u, u) * dx

def I3(u):
    return (u**2 / 2 + u**3 / 6) * dx
```

We project the initial condition on u, but we also need a consistent initial condition for the auxiliary variable. We need to find $\tilde{w}_H \in V$ such that

$$(\tilde{w}_H, v)_{H^1} = \langle \frac{\delta I_3}{\delta u}, v \rangle \text{ for all } v \in V$$

```
uwH = Function(Z)
u0, wH0 = uwH.subfunctions

v = TestFunction(V)
w = TrialFunction(V)
a = h1inner(w, v) * dx
dHdu = derivative(I3(u0), u0, v)

solve(a == h1inner(uexact, v)*dx, u0)
solve(a == dHdu, wH0)
```

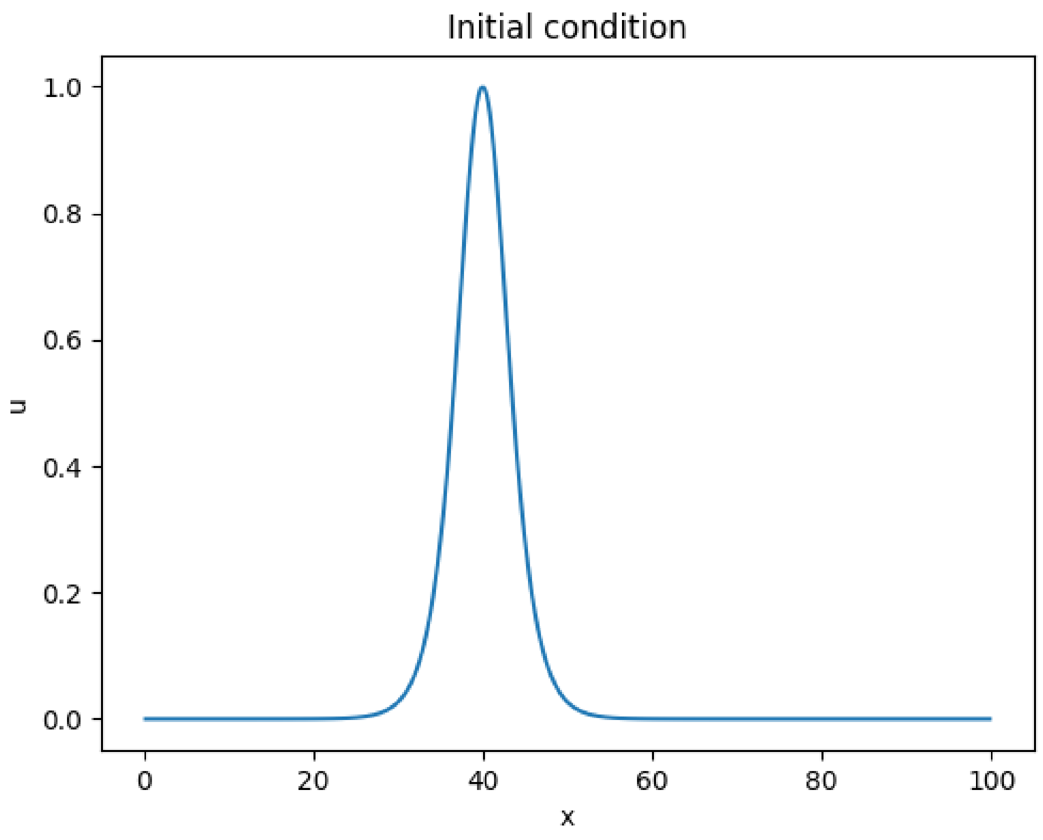
Visualize the initial condition:

```
fig, axes = plt.subplots(1)
plot(Function(FunctionSpace(msh, "CG", 1)).interpolate(u0), axes=axes)
axes.set_title("Initial condition")
axes.set_xlabel("x")
axes.set_ylabel("u")
plt.savefig("bbm_init.png")
```

Create time quadrature labels:

```
time_order_low = 2 * (time_deg - 1)
time_order_high = 3 * time_deg - 1
```

(continues on next page)



(continued from previous page)

```
Llow = TimeQuadratureLabel(time_order_low)
Lhigh = TimeQuadratureLabel(time_order_high)
```

This tags several of the terms with a low-order time integration scheme, but forces a higher-order method on the nonlinear term:

```
u, wH = split(uwH)
v, vH = split(TestFunction(Z))

Flow = h1inner(Dt(u) + wH.dx(0), v) * dx + h1inner(wH, vH) * dx
Fhigh = replace(dHdu, {u0: u})

F = Llow(Flow) - Lhigh(Fhigh(vH))
```

This sets up the cPG time stepper. There are two fields in the unknown, we indicate the second one is an auxiliary and hence to be discretized in the DG test space instead by passing the *aux_indices* keyword:

```
stepper = GalerkinTimeStepper(
    F, time_deg, t, dt, uwH, aux_indices=[1])
```

UFL expressions for the invariants, which we are going to track as we go through time steps:

```
times = [float(t)]
functionals = (I1(u), I2(u), I3(u))
invariants = [tuple(map(assemble, functionals))]
```

Do the time-stepping:

```
for _ in range(Nt):
    stepper.advance()

    invariants.append(tuple(map(assemble, functionals)))

    i1, i2, i3 = invariants[-1]
    t.assign(float(t) + float(dt))
    times.append(float(t))

    print(f'{float(t):.15f}, {i1:.15f}, {i2:.15f}, {i3:.15f}')
```

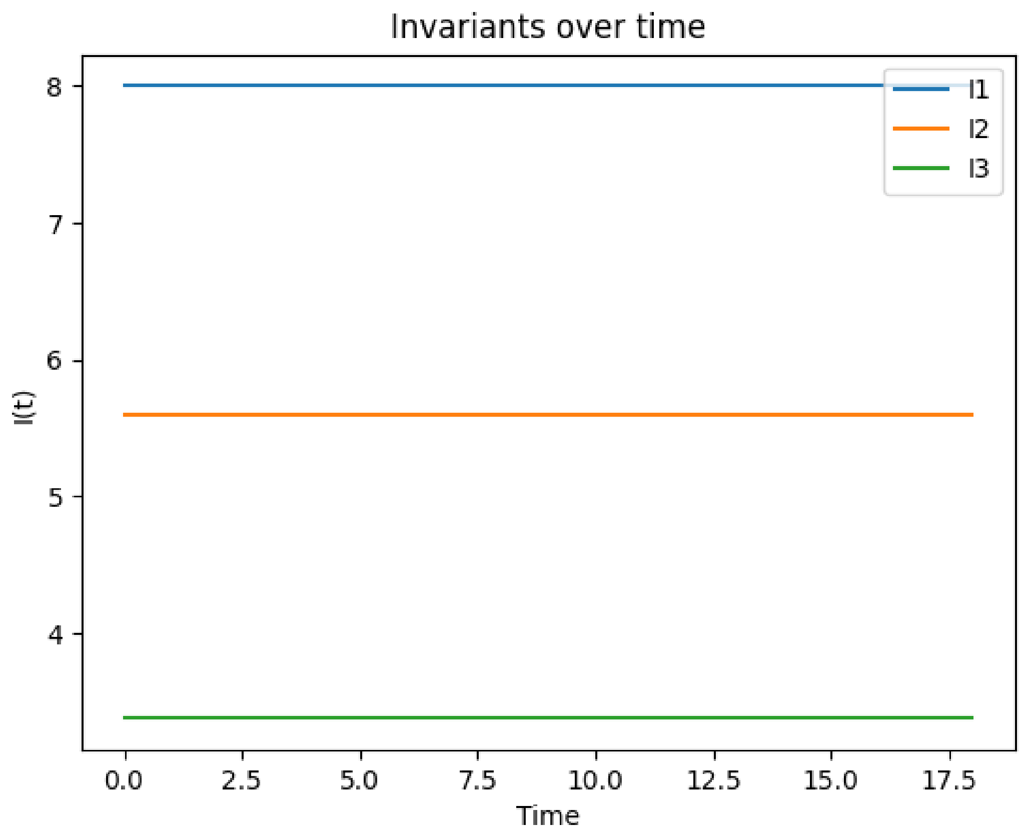
Visualize invariant preservation:

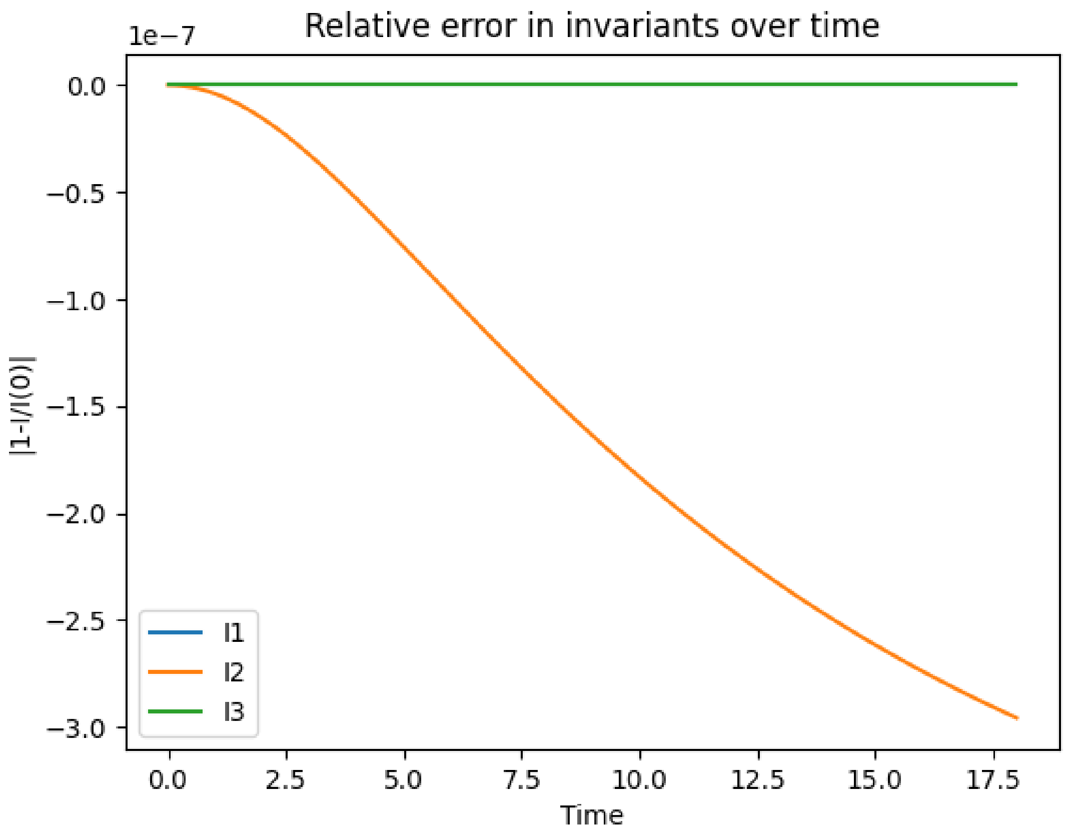
```
axes.clear()
invariants = numpy.array(invariants)

lbls = ("I1", "I2", "I3")

for i in (0, 1, 2):
    plt.plot(times, invariants[:, i], label=lbls[i])
axes.set_title("Invariants over time")
axes.set_xlabel("Time")
axes.set_ylabel("I(t)")
axes.legend()
plt.savefig("invariants.png")
axes.clear()

for i in (0, 1, 2):
    plt.plot(times, 1.0 - invariants[:, i]/invariants[0, i], label=lbls[i])
axes.set_title("Relative error in invariants over time")
axes.set_xlabel("Time")
axes.set_ylabel("|1-I/I(0)|")
axes.legend()
plt.savefig("invariant_errors.png")
```

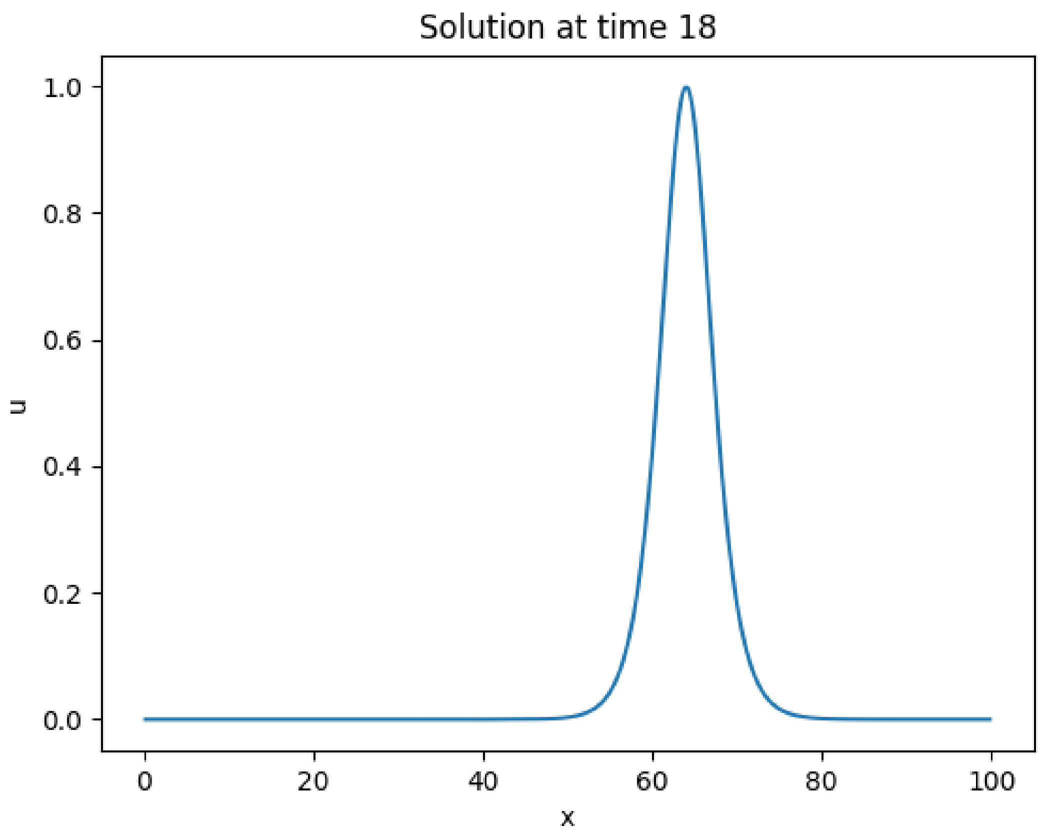
Visualize the solution at final time step:

```
axes.clear()
plot(Function(FunctionSpace(msh, "CG", 1)).interpolate(u0), axes=axes)
axes.set_title(f"Solution at time {tfinal}")
axes.set_xlabel("x")
axes.set_ylabel("u")
plt.savefig("bbm_final.png")
```

Finally, if you feel you must bypass the `TimeStepper` abstraction, we have some examples how to interact with Irksome at a slightly lower level:


Invariants over time
I(t)
Time
I1
I2
I3


Relative error in invariants over time
1e−7
|1-I/I(0)|
Time
I1
I2
I3



3.21.4 Not using the TimeStepper interface for the heat equation

Invariably, somebody will have a (possibly) compelling reason or at least urgent desire to avoid the top-level interface. This demo shows how one can do just that. It will be sparsely documented except for the critical bits and should only be read after perusing the more basic demos.

We're solving the same problem that is done in the heat equation demos.

Imports:

```
from firedrake import *

from irksome import GaussLegendre, getForm, Dt, MeshConstant
from irksome.tools import get_stage_space
```

Note that we imported *getForm* rather than `TimeStepper`. That's the lower-level function inside Irksome that manipulates UFL and boundary conditions.

Continuing:

```
butcher_tableau = GaussLegendre(1)
N = 64

x0 = 0.0
x1 = 10.0
y0 = 0.0
```

(continues on next page)

(continued from previous page)

```
y1 = 10.0

msh = RectangleMesh(N, N, x1, y1)
V = FunctionSpace(msh, "CG", 1)
x, y = SpatialCoordinate(msh)

MC = MeshConstant(msh)
dt = MC.Constant(10 / N)
t = MC.Constant(0.0)

S = Constant(2.0)
C = Constant(1000.0)

B = (x-Constant(x0))*(x-Constant(x1))*(y-Constant(y0))*(y-Constant(y1))/C
R = (x * x + y * y) ** 0.5
uexact = B * atan(t)*(pi / 2.0 - atan(S * (R - t)))

rhs = Dt(uexact) - div(grad(uexact))

u = Function(V)
u.interpolate(uexact)

v = TestFunction(V)
F = inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx - inner(rhs, v)*dx

bc = DirichletBC(V, 0, "on_boundary")
```

Get the function space for the stage-coupled problem and a function to hold the stages we're computing:

```
Vbig = get_stage_space(V, butcher_tableau.num_stages)
k = Function(Vbig)
```

Get the variational form and bcs for the stage-coupled variational problem:

```
Fnew, bcnew = getForm(F, butcher_tableau, t, dt, u, k, bcs=bc)
```

This returns several things:

- `Fnew` is the UFL variational form for the fully discrete method.
- `bcnew` is a list of new `DirichletBC` that need to be enforced on the variational problem for the stages

Solver parameters are just blunt-force LU. Other options are surely possible:

```
luparams = {"mat_type": "aij",
            "snes_type": "ksponly",
            "ksp_type": "preonly",
            "pc_type": "lu"}
```

We can set up a new nonlinear variational problem and create a solver for it in standard Firedrake fashion:

```
prob = NonlinearVariationalProblem(Fnew, k, bcs=bcnew)
solver = NonlinearVariationalSolver(prob, solver_parameters=luparams)
```

We'll need to split the stage variable so that we can update the solution after solving for the stages at each time step:

```
ks = k.subfunctions
```

And here is our time-stepping loop. Note that unlike in the higher-level interface examples, we have to manually update the solution:

```
while (float(t) < 1.0):
    if float(t) + float(dt) > 1.0:
        dt.assign(1.0 - float(t))
    solver.solve()

    for i in range(butcher_tableau.num_stages):
        u += float(dt) * butcher_tableau.b[i] * ks[i]

    t.assign(float(t) + float(dt))
    print(float(t))

print()
print(errornorm(uexact, u)/norm(uexact))
```

3.21.5 Time-dependent BCs and the low-level interface

This demo shows how to update inhomogeneous and time-dependent boundary conditions with the low-level interface. We are using a different manufactured solution than before, which is obvious from reading through the code.

Imports:

```
from firedrake import *

from irksome import GaussLegendre, getForm, Dt, MeshConstant
from irksome.tools import get_stage_space

butcher_tableau = GaussLegendre(2)
N = 64

msh = UnitSquareMesh(N, N)

MC = MeshConstant(msh)
dt = MC.Constant(10 / N)
t = MC.Constant(0.0)

V = FunctionSpace(msh, "CG", 1)
x, y = SpatialCoordinate(msh)
```

(continues on next page)

(continued from previous page)

```
uexact = exp(-t) * cos(pi * x) * sin(pi * y)
rhs = Dt(uexact) - div(grad(uexact))

u = Function(V)
u.interpolate(uexact)

v = TestFunction(V)
F = inner(Dt(u), v)*dx + inner(grad(u), grad(v))*dx - inner(rhs, v)*dx

bc = DirichletBC(V, uexact, "on_boundary")
```

Get the function space for the stage-coupled problem and a function to hold the stages we're computing:

```
Vbig = get_stage_space(V, butcher_tableau.num_stages)
k = Function(Vbig)
```

Get the variational form and bcs for the stage-coupled variational problem:

```
Fnew, bcnew = getForm(F, butcher_tableau, t, dt, u, k, bcs=bc)
```

Recall that *getForm* produces:

- `Fnew` is the UFL variational form for the fully discrete method.
- `bcnew` is a list of new `DirichletBC` that need to be enforced on the variational problem for the stages

We just use basic solver parameters and set up the variational problem and solver:

```
luparams = {"mat_type": "aij",
            "snes_type": "ksponly",
            "ksp_type": "preonly",
            "pc_type": "lu"}

prob = NonlinearVariationalProblem(Fnew, k, bcs=bcnew)
solver = NonlinearVariationalSolver(prob, solver_parameters=luparams)

ks = k.subfunctions
```

Now, our time-stepping loop shows how to manually update the per-stage boundary conditions at each time step:

```
while (float(t) < 1.0):
    if float(t) + float(dt) > 1.0:
        dt.assign(1.0 - float(t))

    solver.solve()

    for i in range(butcher_tableau.num_stages):
        u += float(dt) * butcher_tableau.b[i] * ks[i]

```

(continues on next page)

(continued from previous page)

```
    t.assign(float(t) + float(dt))
    print(float(t))

print()
print(errornorm(uexact, u)/norm(uexact))
```

3.21.6 Mixed problems with the low-level interface

Updating the solution at each time step is more complicated when we have problems on mixed function spaces. This demo peels back the `TimeStepper` abstraction in the mixed heat equation demo. In this case, the Dirichlet boundary conditions are weakly enforced. However, mixed problems do not change the way strongly-enforced BC are handled, just how the solution is updated.

Imports:

```
from irksome.tools import get_stage_space
from firedrake import *
from irksome import LobattoIIIC, Dt, getForm, MeshConstant

butcher_tableau = LobattoIIIC(2)
```

Build the mesh and approximating spaces:

```
N = 32
x0 = 0.0
x1 = 10.0
y0 = 0.0
y1 = 10.0
msh = RectangleMesh(N, N, x1, y1)

V = FunctionSpace(msh, "RT", 2)
W = FunctionSpace(msh, "DG", 1)
Z = V * W

MC = MeshConstant(msh)
dt = MC.Constant(10.0 / N)
t = MC.Constant(0.0)

x, y = SpatialCoordinate(msh)

S = Constant(2.0)
C = Constant(1000.0)

B = (x-Constant(x0))*(x-Constant(x1))*(y-Constant(y0))*(y-Constant(y1))/C
R = (x * x + y * y) ** 0.5

uexact = B * atan(t)*(pi / 2.0 - atan(S * (R - t)))
sigexact = -grad(uexact)
```

(continues on next page)

(continued from previous page)

```
rhs = Dt(uexact) + div(sigexact)

sigu = project(as_vector([0, 0, uexact]), Z)
sigma, u = split(sigu)

v, w = TestFunctions(Z)

F = (inner(Dt(u), w) * dx + inner(div(sigma), w) * dx - inner(rhs, w) * dx
     + inner(sigma, v) * dx - inner(u, div(v)) * dx)
```

Get the function space for the stage-coupled problem:

```
Vbig = get_stage_space(Z, butcher_tableau.num_stages)
k = Function(Vbig)
```

Get the form and new boundary conditions (which are dropped since we have weak Dirichlet)):

```
Fnew, _ = getForm(F, butcher_tableau, t, dt, sigu, k)
```

We set up the variational problem and solver using a sparse direct method:

```
params = {"mat_type": "aij",
          "snes_type": "ksponly",
          "ksp_type": "preonly",
          "pc_type": "lu"}

prob = NonlinearVariationalProblem(Fnew, k)
solver = NonlinearVariationalSolver(prob, solver_parameters=params)
```

Advancing the solution in time is a bit more complicated now:

```
num_fields = len(Z)
b = butcher_tableau.b

while (float(t) < 1.0):
    if (float(t) + float(dt) > 1.0):
        dt.assign(1.0 - float(t))

    solver.solve()

    for s in range(butcher_tableau.num_stages):
        for i in range(num_fields):
            sigu.dat.data[i][:] += float(dt) * b[s] * k.dat.data[num_fields *␣
↪s + i][:]

    print(float(t))
    t.assign(float(t) + float(dt))
```

Finally, we check the accuracy of the solution:

```
sigma, u = sigu.subfunctions
print("U error      : ", errornorm(uexact, u) / norm(uexact))
print("Sig error    : ", errornorm(sigexact, sigma) / norm(sigexact))
print("Div Sig error: ",
      errornorm(sigexact, sigma, norm_type='Hdiv')
      / norm(sigexact, norm_type='Hdiv'))
```

CHAPTER

FOUR

API DOCUMENTATION

There is also an alphabetical index, and a search engine.

4.1 irksome package

4.1.1 Submodules

4.1.2 irksome.ButcherTableaux module

class `irksome.ButcherTableaux.`**`Alexander`**

Bases: *`ButcherTableau`*

Third-order, diagonally implicit, 3-stage, L-stable scheme from Diagonally Implicit Runge-Kutta Methods for Stiff O.D.E.'s, R. Alexander, SINUM 14(6): 1006-1021, 1977.

class `irksome.ButcherTableaux.`**`BackwardEuler`**

Bases: *`RadauIIA`*

The rock-solid first-order implicit method.

class `irksome.ButcherTableaux.`**`ButcherTableau`**(*A*, *b*, *btilde*, *c*, *order*, *embedded_order*, *gamma0*)

Bases: `object`

Top-level class representing a Butcher tableau encoding
a Runge-Kutta method. It has members

Parameters

- **`A`** – a 2d array containing the Butcher matrix
- **`b`** – a 1d array giving weights assigned to each stage when computing the solution at time n+1.
- **`btilde`** – If present, a 1d array giving weights for an embedded lower-order method (used in estimating temporal truncation error.)
- **`c`** – a 1d array containing weights at which time-dependent terms are evaluated.
- **`order`** – the (integer) formal order of accuracy of the method
- **`embedded_order`** – If present, the (integer) formal order of accuracy of the embedded method

- **gamma0** – If present, the weight on the explicit term in the embedded lower-order method

property is_diagonally_implicit

property is_explicit

property is_fully_implicit

property is_implicit

property is_stiffly_accurate

Determines whether the method is stiffly accurate.

property num_stages

Return the number of stages the method has.

class irksome.ButcherTableaux.**CollocationButcherTableau**(*L*, *order*)

Bases: *ButcherTableau*

When an RK method is based on collocation with point sets present in FIAT, we have a general formula for producing the Butcher tableau.

Parameters

- **L** – a one-dimensional class FIAT.FiniteElement of Lagrange type – the degrees of freedom must all be point evaluation.
- **order** – the order of the resulting RK method.

class irksome.ButcherTableaux.**GaussLegendre**(*num_stages*)

Bases: *CollocationButcherTableau*

Collocation method based on the Gauss-Legendre points. The order of accuracy is 2 * *num_stages*. GL methods are A-stable, B-stable, and symplectic.

Parameters

num_stages – The number of stages (1 or greater)

class irksome.ButcherTableaux.**LobattoIIIA**(*num_stages*)

Bases: *CollocationButcherTableau*

Collocation method based on the Gauss-Lobatto points. The order of accuracy is 2 * *num_stages* - 2. LobattoIIIA methods are A-stable but not B- or L-stable.

Parameters

num_stages – The number of stages (2 or greater)

class irksome.ButcherTableaux.**LobattoIIIC**(*num_stages*)

Bases: *ButcherTableau*

Discontinuous collocation method based on the Lobatto points. The order of accuracy is 2 * *num_stages* - 2. LobattoIIIC methods are A-, L-, algebraically, and B- stable.

Parameters

num_stages – The number of stages (2 or greater)

class irksome.ButcherTableaux.**PareschiRusso**(*x*)

Bases: *ButcherTableau*

Second order, diagonally implicit, 2-stage. A-stable if x >= 1/4 and L-stable iff x = 1 plus/minus 1/sqrt(2).

class irksome.ButcherTableaux.**QinZhang**

Bases: *PareschiRusso*

Symplectic Pareschi-Russo DIRK

class irksome.ButcherTableaux.**RadauIIA**(*num_stages*, *variant='embed_Radau5'*)

Bases: *CollocationButcherTableau*

Collocation method based on the Gauss-Radau points. The order of accuracy is 2 * *num_stages* - 1. RadauIIA methods are algebraically (hence B-) stable.

Parameters

- **num_stages** – The number of stages (2 or greater)
- **variant** – Indicate whether to use the Radau5 style of embedded scheme (with *variant* = *"embed_Radau5"*) or the simple collocation type (with *variant* = *"embed_colloc"*)

4.1.3 irksome.ars_dirk_imex_tableaux module

class irksome.ars_dirk_imex_tableaux.**ARS_DIRK_IMEX**(*ns_imp*, *ns_exp*, *order*)

Bases: *DIRK_IMEX*

Class to generate IMEX tableaux based on Ascher, Ruuth, and Spiteri (ARS). It has members

Parameters

- **ns_imp** – number of implicit stages
- **ns_exp** – number of explicit stages
- **order** – the (integer) former order of accuracy of the method

4.1.4 irksome.base_time_stepper module

class irksome.base_time_stepper.**BaseTimeStepper**(*F*, *t*, *dt*, *u0*, *bcs=None*, *appctx=None*, *nullspace=None*)

Bases: object

Base class for various time steppers. This is mainly to give code reuse stashing objects that are common to all the time steppers. It's a developer-level class.

abstractmethod advance()

abstractmethod solver_stats()

class irksome.base_time_stepper.**StageCoupledTimeStepper**(*F*, *t*, *dt*, *u0*, *num_stages*, *bcs=None*, *solver_parameters=None*, *appctx=None*, *nullspace=None*, *transpose_nullspace=None*, *near_nullspace=None*, *splitting=None*, *bc_type=None*, *butcher_tableau=None*, *bounds=None*, ***kwargs*)

Bases: `BaseTimeStepper`

This developer-level class provides common features used by various methods requiring stage-coupled variational problems to compute the stages (e.g. fully implicit RK, Galerkin-in-time)

Parameters

- **F** – A `ufl.Form` instance describing the semi-discrete problem F(t, u; v) == 0, where *u* is the unknown `firedrake.Function and `v` is the :class:firedrake.TestFunction`.
- **t** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time.
- **dt** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time step. The user may adjust this value between time steps.
- **u0** – A `firedrake.Function` containing the current state of the problem to be solved.
- **num_stages** – The number of stages to solve for. It could be the number of RK stages or relate to the polynomial degree (Galerkin)
- **bcs** – An iterable of `firedrake.DirichletBC` or *firedrake.EquationBC* containing the strongly-enforced boundary conditions. Irksome will manipulate these to obtain boundary conditions for each stage of the RK method. Support for *firedrake.EquationBC* is limited to the stage derivative formulation with DAE style BCs.
- **solver_parameters** – An optional `dict` of solver parameters that will be used in solving the algebraic problem associated with each time step.
- **appctx** – An optional `dict` containing application context. This gets included with particular things that Irksome will pass into the nonlinear solver so that, say, user-defined preconditioners have access to it.
- **nullspace** – A list of tuples of the form (index, VSB) where index is an index into the function space associated with *u* and VSB is a :class: *firedrake.VectorSpaceBasis* instance to be passed to a *firedrake.MixedVectorSpaceBasis* over the larger space associated with the Runge-Kutta method
- **splitting** – An optional kwarg (not used by all superclasses)

- **bc_type** – An optional kwarg (not used by all superclasses)
- **butcher_tableau** – A `ButcherTableau` instance giving the Runge-Kutta method to be used for time marching.
- **bounds** – An optional kwarg used in certain bounds-constrained methods.

advance()

Advances the system from time *t* to time *t + dt*. Note: overwrites the value *u0*.

abstractmethod get_form_and_bcs(*stages*, *tableau=None*, *F=None*)

get_stage_bounds(*bounds=None*)

get_stages()

solver_stats()

4.1.5 irksome.bcs module

irksome.bcs.**BCStageData**(*bc*, *gcur*, *u0*, *stages*, *i*)

class irksome.bcs.**BoundsConstrainedDirichletBC**(*V*, *g*, *sub_domain*, *bounds*, *solver_parameters=None*)

Bases: `DirichletBC`

A DirichletBC with bounds-constrained data.

property function_arg

The value of this boundary condition.

reconstruct(*V=None*, *g=None*, *sub_domain=None*)

irksome.bcs.**EmbeddedBCData**(*bc*, *butcher_tableau*, *t*, *dt*, *u0*, *stages*)

irksome.bcs.**bc2space**(*bc*, *V*)

irksome.bcs.**extract_bcs**(*bcs*)

Return an iterable of boundary conditions on the residual form

irksome.bcs.**get_sub**(*u*, *indices*)

irksome.bcs.**stage2spaces4bc**(*bc*, *V*, *Vbig*, *i*)

used to figure out how to apply Dirichlet BC to each stage

4.1.6 irksome.deriv module

irksome.deriv.**Dt**(*f*, *order=1*)

Short-hand function to produce a *`TimeDerivative`* of a given order.

class irksome.deriv.**TimeDerivative**(*f*)

Bases: `Derivative`

UFL node representing a time derivative of some quantity/field. Note: Currently form compilers do not understand how to process these nodes. Instead, Irksome pre-processes forms containing *TimeDerivative* nodes.

Initalise.

property ufl_free_indices

property ufl_index_dimensions

property ufl_shape

class irksome.deriv.**TimeDerivativeRuleDispatcher**(*t=None*, *timedep_coeffs=None*, ***kwargs*)

Bases: `DAGTraverser`

Mapping rules to splat out time derivatives so that replacement should work on more complex problems.

Initialise.

process(*o*)

process(*o:* TimeDerivative)

process(*o: BaseForm*)

process(*o: Expr*)

Process node by type.

Args:
o: *Expr* to start DAG traversal from. **kwargs: keyword arguments for the `process` singledispatchmethod.

Returns:
Processed *Expr*.

time_derivative(*o*)

class irksome.deriv.**TimeDerivativeRuleset**(*t=None*, *timedep_coeffs=None*)

Bases: `GenericDerivativeRuleset`

Apply AD rules to time derivative expressions.

Initialise.

constant(*o*)

process(*o*)

process(*o: ConstantValue*)

process(*o: SpatialCoordinate*)

process(*o: Coefficient*)

process(*o:* TimeDerivative, *f*)

process(*o: Variable*, **operands*)

process(*o: ReferenceValue*, **operands*)

process(*o: ReferenceGrad*, **operands*)

process(*o: Indexed*, **operands*)

process(*o: Grad*, **operands*)

process(*o: Div*, **operands*)

process(*o: Derivative*, **operands*)

process(*o: Curl*, **operands*)

process(*o: Conj*, **operands*)

Process o.

Args:
o: *Expr* to be processed.

Returns:
Processed object.

terminal(*o*)

terminal_modifier(*o*, **operands*)

time_derivative(*o*, *f*)

irksome.deriv.**apply_time_derivatives**(*expression*, *t=None*, *timedep_coeffs=None*)

irksome.deriv.**expand_time_derivatives**(*expression*, *t=None*, *timedep_coeffs=None*)

4.1.7 irksome.dirk_imex_tableaux module

class irksome.dirk_imex_tableaux.**DIRK_IMEX**(*A: ndarray*, *b: ndarray*, *c: ndarray*, *A_hat: ndarray*, *b_hat: ndarray*, *c_hat: ndarray*, *order: int = None*)

Bases: *ButcherTableau*

Top-level class representing a pair of Butcher tableau encoding an implicit-explicit additive Runge-Kutta method. Since the explicit Butcher matrix is strictly lower triangular, only the lower-left (ns - 1)x(ns - 1) block is given. However, the full b_hat and c_hat are given. It has members

Parameters

- **A** – a 2d array containing the implicit Butcher matrix
- **b** – a 1d array giving weights assigned to each implicit stage when computing the solution at time n+1.
- **c** – a 1d array containing weights at which time-dependent implicit terms are evaluated.
- **A_hat** – a 2d array containing the explicit Butcher matrix (lower-left block only)
- **b_hat** – a 1d array giving weights assigned to each explicit stage when computing the solution at time n+1.
- **c_hat** – a 1d array containing weights at which time-dependent explicit terms are evaluated.
- **order** – the (integer) formal order of accuracy of the method

4.1.8 irksome.dirk_stepper module

class irksome.dirk_stepper.**DIRKTimeStepper**(*F*, *butcher_tableau*, *t*, *dt*, *u0*, *bcs=None*, *solver_parameters=None*, *appctx=None*, *nullspace=None*, *transpose_nullspace=None*, *near_nullspace=None*, ***kwargs*)

Bases: `object`

Front-end class for advancing a time-dependent PDE via a diagonally-implicit Runge-Kutta method formulated in terms of stage derivatives.

`advance`()

`get_form_and_bcs`(*stages*, *tableau=None*, *F=None*)

`solver_stats`()

`update_bc_constants`(*i*, *c*)

`irksome.dirk_stepper.`**`getFormDIRK`**(*F*, *ks*, *butch*, *t*, *dt*, *u0*, *bcs=None*)

4.1.9 irksome.discontinuous_galerkin_stepper module

class `irksome.discontinuous_galerkin_stepper.`**`DiscontinuousGalerkinTimeStepper`**(*F*, *order*, *t*, *dt*, *u0*, *bcs=None*, *basis_type=Nor*, *quadrature=None*, ***kwargs*)

Bases: *`StageCoupledTimeStepper`*

Front-end class for advancing a time-dependent PDE via a Discontinuous Galerkin in time method

Parameters

- **F** – A `ufl.Form` instance describing the semi-discrete problem F(t, u; v) == 0, where *u* is the unknown `firedrake.Function and `v` is the :class:firedrake.TestFunction`.
- **order** – an integer indicating the order of the DG space to use (with order == 0 corresponding to DG(0)-in-time)
- **t** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time.
- **dt** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time step. The user may adjust this value between time steps.
- **u0** – A `firedrake.Function` containing the current state of the problem to be solved.
- **bcs** – An iterable of `firedrake.DirichletBC` containing the strongly-enforced boundary conditions. Irksome will manipulate these to obtain boundary conditions for each stage of the method.

- **basis_type** – A string indicating the finite element family (either *'Lagrange'* or *'Bernstein'*) or the Lagrange variant for the test/trial spaces. Defaults to equispaced Lagrange elements.
- **quadrature** – A `FIAT.QuadratureRule` indicating the quadrature to be used in time, defaulting to GL with order+1 points
- **solver_parameters** – A `dict` of solver parameters that will be used in solving the algebraic problem associated with each time step.
- **appctx** – An optional `dict` containing application context. This gets included with particular things that Irksome will pass into the nonlinear solver so that, say, user-defined preconditioners have access to it.
- **nullspace** – A list of tuples of the form (index, VSB) where index is an index into the function space associated with *u* and VSB is a :class: *firedrake.VectorSpaceBasis* instance to be passed to a *firedrake.MixedVectorSpaceBasis* over the larger space associated with the Runge-Kutta method

get_form_and_bcs(*stages*, *basis_type=None*, *order=None*, *quadrature=None*, *F=None*)

`irksome.discontinuous_galerkin_stepper.`**getElement**(*basis_type*, *order*)

`irksome.discontinuous_galerkin_stepper.`**getFormDiscGalerkin**(*F*, *L*, *Q*, *t*, *dt*, *u0*, *stages*, *bcs=None*)

Given a time-dependent variational form, trial and test spaces, and a quadrature rule, produce UFL for the Discontinuous Galerkin-in-Time method.

Parameters

- **F** – UFL form for the semidiscrete ODE/DAE
- **L** – A `FIAT.FiniteElement` for the test and trial functions in time
- **Q** – A `FIAT.QuadratureRule` for the time integration
- **t** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time.
- **dt** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time step. The user may adjust this value between time steps.
- **u0** – a `Function` referring to the state of the PDE system at time *t*
- **stages** – a `Function` representing the stages to be solved for.
- **bcs** – optionally, a `DirichletBC` object (or iterable thereof) containing (possibly time-dependent) boundary conditions imposed on the system.
- **nullspace** – A list of tuples of the form (index, VSB) where index is an index into the function space associated with *u* and VSB is a :class: *firedrake.VectorSpaceBasis* instance to be passed to a *firedrake.MixedVectorSpaceBasis* over the larger space associated with the Runge-Kutta method

On output, we return a tuple consisting of four parts:

- Fnew, the `Form` corresponding to the DG-in-Time discretized problem
- *bcnew*, a list of `firedrake.DirichletBC` objects to be posed on the Galerkin-in-time solution,

4.1.10 irksome.explicit_stepper module

class `irksome.explicit_stepper.`**`ExplicitTimeStepper`**(*F*, *butcher_tableau*, *t*, *dt*, *u0*, *bcs=None*, *solver_parameters=None*, *appctx=None*)

Bases: *DIRKTimeStepper*

4.1.11 irksome.galerkin_stepper module

class `irksome.galerkin_stepper.`**`GalerkinTimeStepper`**(*F*, *order*, *t*, *dt*, *u0*, *bcs=None*, *basis_type=None*, *quadrature=None*, *aux_indices=None*, ***kwargs*)

Bases: *StageCoupledTimeStepper*

Front-end class for advancing a time-dependent PDE via a Galerkin in time method

Parameters

- **F** – A `ufl.Form` instance describing the semi-discrete problem F(t, u; v) == 0, where *u* is the unknown `firedrake.Function and `v` is the :class:firedrake.TestFunction`.
- **order** – an integer indicating the order of the DG space to use (with order == 1 corresponding to CG(1)-in-time for the trial space)
- **t** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time.
- **dt** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time step. The user may adjust this value between time steps.
- **u0** – A `firedrake.Function` containing the current state of the problem to be solved.
- **bcs** – An iterable of `firedrake.DirichletBC` containing the strongly-enforced boundary conditions. Irksome will manipulate these to obtain boundary conditions for each stage of the method.
- **basis_type** – A string indicating the finite element family (either *'Lagrange'* or *'Bernstein'*) or the Lagrange variant for the test/trial spaces. Defaults to equispaced Lagrange elements.
- **quadrature** – A `FIAT.QuadratureRule` indicating the quadrature to be used in time, defaulting to GL with order points
- **aux_indices** – a list of field indices to be discretized in the test space rather than trial space.
- **solver_parameters** – A `dict` of solver parameters that will be used in solving the algebraic problem associated with each time step.

- **appctx** – An optional `dict` containing application context. This gets included with particular things that Irksome will pass into the nonlinear solver so that, say, user-defined preconditioners have access to it.
- **nullspace** – A list of tuples of the form (index, VSB) where index is an index into the function space associated with *u* and VSB is a :class: *firedrake.VectorSpaceBasis* instance to be passed to a *firedrake.MixedVectorSpaceBasis* over the larger space associated with the Runge-Kutta method
- **aux_indices** – a list of field indices to be discretized in the test space rather than trial space.

get_form_and_bcs(*stages*, *basis_type=None*, *order=None*, *quadrature=None*, *aux_indices=None*, *F=None*)

set_initial_guess()

irksome.galerkin_stepper.**getElements**(*basis_type*, *order*)

irksome.galerkin_stepper.**getFormGalerkin**(*F*, *L_trial*, *L_test*, *Qdefault*, *t*, *dt*, *u0*, *stages*, *bcs=None*, *aux_indices=None*)

Given a time-dependent variational form, trial and test spaces, and a quadrature rule, produce UFL for the Galerkin-in-Time method.

Parameters

- **F** – UFL form for the semidiscrete ODE/DAE
- **L_trial** – A `FIAT.FiniteElement` for the trial functions in time
- **L_test** – A `FIAT.FinteElement` for the test functions in time
- **Qdefault** – A `FIAT.QuadratureRule` for the time integration. This rule will be used for all terms in the semidiscrete variational form that aren't specifically tagged with another quadrature rule.
- **t** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time.
- **dt** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time step. The user may adjust this value between time steps.
- **u0** – a `Function` referring to the state of the PDE system at time *t*
- **stages** – a `Function` representing the stages to be solved for.
- **bcs** – optionally, a `DirichletBC` object (or iterable thereof) containing (possibly time-dependent) boundary conditions imposed on the system.
- **aux_indices** – a list of field indices to be discretized in the test space rather than trial space.

On output, we return a tuple consisting of four parts:

- Fnew, the `Form` corresponding to the Galerkin-in-Time discretized problem

- *bcnew*, a list of `firedrake.DirichletBC` objects to be posed on the Galerkin-in-time solution,

`irksome.galerkin_stepper.`**`getTermGalerkin`**(*F*, *L_trial*, *L_test*, *Q*, *t*, *dt*, *u0*, *stages*, *test*, *aux_indices*)

4.1.12 irksome.imex module

`class` `irksome.imex.`**`DIRKIMEXMethod`**(*F*, *F_explicit*, *butcher_tableau*, *t*, *dt*, *u0*, *bcs=None*, *solver_parameters=None*, *mass_parameters=None*, *appctx=None*, *nullspace=None*)

Bases: `object`

Front-end class for advancing a time-dependent PDE via a diagonally-implicit Runge-Kutta IMEX method formulated in terms of stage derivatives. This implementation assumes a weak form written as F + F_explicit = 0, where both F and F_explicit are UFL Forms, with terms in F to be handled implicitly and those in F_explicit to be handled explicitly

`advance`()

`solver_stats`()

`class` `irksome.imex.`**`RadauIIAIMEXMethod`**(*F*, *Fexp*, *butcher_tableau*, *t*, *dt*, *u0*, *bcs=None*, *it_solver_parameters=None*, *prop_solver_parameters=None*, *splitting=<function AI>*, *appctx=None*, *nullspace=None*, *num_its_initial=0*, *num_its_per_step=0*)

Bases: `object`

Class for advancing a time-dependent PDE via a polynomial IMEX/RadauIIA method. This requires one to split the PDE into an implicit and explicit part. The class sets up two methods – *advance* and *iterate*. The former is used to move the solution forward in time, while the latter is used both to start the method (filling up the initial stage values) and can be used at each time step to increase the accuracy/stability. In the limit as the iterator is applied many times per time step, one expects convergence to the solution that would have been obtained from fully-implicit RadauIIA method.

Parameters

- **`F`** – A `ufl.Form` instance describing the implicit part of the semi-discrete problem F(t, u; v) == 0, where *u* is the unknown `firedrake.Function` `and` `` `v `` is the :class:firedrake.TestFunction\`.
- **`Fexp`** – A `ufl.Form` instance describing the part of the PDE that is explicitly split off.
- **`butcher_tableau`** – A `ButcherTableau` instance giving the Runge-Kutta method to be used for time marching. Only RadauIIA is allowed here (but it can be any number of stages).
- **`t`** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time.

- **dt** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time step. The user may adjust this value between time steps.
- **u0** – A `firedrake.Function` containing the current state of the problem to be solved.
- **bcs** – An iterable of `firedrake.DirichletBC` containing the strongly-enforced boundary conditions. Irksome will manipulate these to obtain boundary conditions for each stage of the RK method.
- **it_solver_parameters** – A `dict` of solver parameters that will be used in solving the algebraic problem associated with the iterator.
- **prop_solver_parameters** – A `dict` of solver parameters that will be used in solving the algebraic problem associated with the propagator.
- **splitting** – A callable used to factor the Butcher matrix, currently, only AI is supported.
- **appctx** – An optional `dict` containing application context.
- **nullspace** – An optional null space object.

advance()

get_form_and_bcs(*stages*, *tableau=None*, *F=None*)

iterate()

Called 1 or more times to set up the initial state of the system before time-stepping. Can also be called after each call to *advance*

propagate()

Moves the solution forward in time, to be followed by 0 or more calls to *iterate*.

solver_stats()

irksome.imex.**getFormExplicit**(*Fexp*, *butch*, *u0*, *UU*, *t*, *dt*, *splitting=None*)

Processes the explicitly split-off part for a RadauIIA-IMEX method. Returns the forms for both the iterator and propagator, which really just differ by which constants are in them.

irksome.imex.**getFormsDIRKIMEX**(*F*, *Fexp*, *ks*, *khats*, *butch*, *t*, *dt*, *u0*, *bcs=None*)

irksome.imex.**riia_explicit_coeffs**(*k*)

Computes the coefficients needed for the explicit part of a RadauIIA-IMEX method.

4.1.13 irksome.labeling module

class irksome.labeling.**TimeQuadratureLabel**(**args*)

Bases: `Label`

If the constructor gets one argument, it's an integer for the order of the quadrature rule. If there are two arguments, assume they are the points and weights.

Parameters

label
The name of the label.

value
The value for the label to take. Can be any type (subject to the validator). Defaults to True.

validator
Function to check the validity of any value later passed to the label. Defaults to None.

`default_value`

`label`

`validator`

`value`

class `irksome.labeling.`**`TimeQuadratureRule`**(*x*, *w*)

Bases: `object`

`get_points`()

`get_weights`()

`irksome.labeling.`**`split_explicit`**(*F*)

`irksome.labeling.`**`split_quadrature`**(*F*, *Qdefault=None*)

4.1.14 irksome.manipulation module

Manipulation of expressions containing *`TimeDerivative`* terms.

These can be used to do some basic checking of the suitability of a `Form` for use in Irksome (via *`check_integrals`*), and splitting out terms in the `Form` that contain a time derivative from those that don't (via *`extract_terms`*).

class `irksome.manipulation.`**`SplitTimeForm`**(*time: Form*, *remainder: Form*)

Bases: `NamedTuple`

A container for a form split into time terms and a remainder.

Create new instance of SplitTimeForm(time, remainder)

`remainder: Form`
Alias for field number 1

`time: Form`
Alias for field number 0

`irksome.manipulation.`**`check_integrals`**(*integrals: List[Integral]*, *expect_time_derivative: bool = True*) → List[Integral]

Check a list of integrals for linearity in the time derivative.

Parameters

- **`integrals`** – list of integrals.

- **expect_time_derivative** – Are we expecting to see a time derivative?

Raises

ValueError – if we are expecting a time derivative and don't see one, or time derivatives are applied nonlinearly, to more than one coefficient, or more than first order.

irksome.manipulation.**extract_terms**(*form: Form*) → *SplitTimeForm*

Extract terms from a Form.

This splits a form (a sum of integrals) into those integrals which do contain a *TimeDerivative* and those that don't.

Parameters

form – The form to split.

Returns

a *SplitTimeForm* tuple.

Raises

ValueError – if the form does not apply anything other than first-order time derivatives to a single coefficient.

4.1.15 irksome.nystrom_dirk_stepper module

class irksome.nystrom_dirk_stepper.**DIRKNystromTimeStepper**(*F*, *tableau*, *t*, *dt*, *u0*, *ut0*, *bcs=None*, *solver_parameters=None*, *appctx=None*, *nullspace=None*, *transpose_nullspace=None*, *near_nullspace=None*, *bc_type=None*, ***kwargs*)

Bases: object

Front-end class for advancing a second-order time-dependent PDE via a diagonally-implicit Runge-Kutta-Nystrom method formulated in terms of stage derivatives.

advance()

solver_stats()

update_bc_constants(*i*, *c*)

class irksome.nystrom_dirk_stepper.**ExplicitNystromTimeStepper**(*F*, *tableau*, *t*, *dt*, *u0*, *ut0*, *bcs=None*, *solver_parameters=None*, *appctx=None*, *nullspace=None*, *transpose_nullspace=None*, *near_nullspace=None*, *bc_type=None*, ***kwargs*)

Bases: *DIRKNystromTimeStepper*

Front-end class for advancing a second-order time-dependent PDE via an explicit Runge-Kutta-Nystrom method formulated in terms of stage derivatives.

`irksome.nystrom_dirk_stepper.`**`getFormDIRKNystrom`**`(`*F*, *ks*, *tableau*, *t*, *dt*, *u0*, *ut0*, *bcs=None*, *bc_type=None*`)`

4.1.16 irksome.nystrom_stepper module

`class` `irksome.nystrom_stepper.`**`ClassicNystrom4Tableau`**

Bases: *`NystromTableau`*

`class` `irksome.nystrom_stepper.`**`NystromTableau`**`(`*A*, *b*, *c*, *Abar*, *bbar*, *order*`)`

Bases: `object`

`property is_diagonally_implicit`

`property is_explicit`

`property is_fully_implicit`

`property is_implicit`

`property num_stages`

`class` `irksome.nystrom_stepper.`**`StageDerivativeNystromTimeStepper`**`(`*F*, *tableau*, *t*, *dt*, *u0*, *ut0*, *bcs=None*, *bc_type='DAE'*, ***kwargs*`)`

Bases: *`StageCoupledTimeStepper`*

`get_form_and_bcs``(`*stages*, *tableau=None*, *F=None*`)`

`irksome.nystrom_stepper.`**`butcher_to_nystrom`**`(`*butch*`)`

`irksome.nystrom_stepper.`**`getFormNystrom`**`(`*F*, *tableau*, *t*, *dt*, *u0*, *ut0*, *stages*, *bcs=None*, *bc_type=None*`)`

4.1.17 irksome.pc module

`class` `irksome.pc.`**`ClinesBase`**

Bases: *`NystromAuxiliaryOperatorPC`*

Base class for methods out of Clines/Howle/Long.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize`
- `update`
- `apply`
- `applyTranspose`

class irksome.pc.**ClinesLD**

Bases: *ClinesBase*

Implements Clines-type preconditioner using Atilde = LD where A=LDU.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize`
- `update`
- `apply`
- `applyTranspose`

getAtildes(*A*, *Abar*)

Derived classes produce a typically structured approximation to A and Abar.

class irksome.pc.**IRKAuxiliaryOperatorPC**

Bases: `AuxiliaryOperatorPC`

Base class that inherits from Firedrake's AuxiliaryOperatorPC class and provides the preconditioning bilinear form associated with an auxiliary Form and/or approximate Butcher matrix (which are provided by subclasses).

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize`
- `update`
- `apply`
- `applyTranspose`

form(*pc*, *test*, *trial*)

Implements the interface for AuxiliaryOperatorPC.

getAtilde(*A*)

Derived classes produce a typically structured approximation to A.

getNewForm(*pc*, *u0*, *test*)

Derived classes can optionally provide an auxiliary Form.

class irksome.pc.**NystromAuxiliaryOperatorPC**

Bases: `AuxiliaryOperatorPC`

Base class that inherits from Firedrake's AuxiliaryOperatorPC class and provides the preconditioning bilinear form associated with an auxiliary Form and/or approximate Nystrom matrices (which are provided by subclasses).

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize`
- `update`

- `apply`
- `applyTranspose`

`form`(*pc*, *test*, *trial*)
: Implements the interface for AuxiliaryOperatorPC.

`getAtildes`(*A*, *Abar*)
: Derived classes produce a typically structured approximation to A and Abar.

`getNewForm`(*pc*, *u0*, *ut0*, *test*)
: Derived classes can optionally provide an auxiliary Form.

class `irksome.pc.`**`RanaBase`**

Bases: *`IRKAuxiliaryOperatorPC`*

Base class for methods out of Rana, Howle, Long, Meek, & Milestone.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize`
- `update`
- `apply`
- `applyTranspose`

class `irksome.pc.`**`RanaDU`**

Bases: *`RanaBase`*

Implements Rana-type preconditioner using Atilde = DU where A=LDU.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize`
- `update`
- `apply`
- `applyTranspose`

`getAtilde`(*A*)
: Derived classes produce a typically structured approximation to A.

class `irksome.pc.`**`RanaLD`**

Bases: *`RanaBase`*

Implements Rana-type preconditioner using Atilde = LD where A=LDU.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize`
- `update`
- `apply`

- `applyTranspose`

getAtilde(*A*)
Derived classes produce a typically structured approximation to A.

`irksome.pc.`**ldu**(*A*)

4.1.18 irksome.pep_explicit_rk module

class `irksome.pep_explicit_rk.`**PEPRK**(*ns*, *order*, *peporder*)
Bases: *ButcherTableau*

4.1.19 irksome.sspk_tableau module

class `irksome.sspk_tableau.`**SSPButcherTableau**(*order*, *ns*)
Bases: *ButcherTableau*

Class used to generate tableau for strong stability preserving (SSP) schemes. It has members

Parameters

- **ns** – number of stages
- **order** – the (integer) formal order of accuracy of the method

class `irksome.sspk_tableau.`**SSPK_DIRK_IMEX**(*ssp_order*, *ns_imp*, *ns_exp*, *order*)
Bases: *DIRK_IMEX*

Class to generate IMEX tableaux based on Pareschi and Russo. It has members

Parameters

- **ssp_order** – order of ssp scheme
- **ns_imp** – number of implicit stages
- **ns_exp** – number of explicit stages
- **order** – the (integer) formal order of accuracy of the method

4.1.20 irksome.stage_derivative module

class `irksome.stage_derivative.`**AdaptiveTimeStepper**(*F*, *butcher_tableau*, *t*, *dt*, *u0*, *bcs=None*, *appctx=None*, *solver_parameters=None*, *bc_type='DAE'*, *splitting=<function AI>*, *nullspace=None*, *tol=0.001*, *dtmin=1e-15*, *dtmax=1.0*, *KI=0.06666666666666667*, *KP=0.13*, *max_reject=10*, *onscale_factor=1.2*, *safety_factor=0.9*, *gamma0_params=None*)
Bases: *StageDerivativeTimeStepper*

Front-end class for advancing a time-dependent PDE via an adaptive Runge-Kutta method.

Parameters

- **F** – A `ufl.Form` instance describing the semi-discrete problem F(t, u; v) == 0, where *u* is the unknown `firedrake.Function and `v` is the :class:firedrake.TestFunction`.
- **butcher_tableau** – A `ButcherTableau` instance giving the Runge-Kutta method to be used for time marching.
- **t** – A `firedrake.Constant` instance that always contains the time value at the beginning of a time step
- **dt** – A `firedrake.Constant` containing the size of the current time step. The user may adjust this value between time steps; however, note that the adaptive time step controls may adjust this before the step is taken.
- **u0** – A `firedrake.Function` containing the current state of the problem to be solved.
- **tol** – The temporal truncation error tolerance
- **dtmin** – Minimal acceptable time step. An exception is raised if the step size drops below this threshhold.
- **dtmax** – Maximal acceptable time step, imposed as a hard cap; this can be adjusted externally once the time-stepper is instantiated, by modifying *stepper.dt_max*
- **KI** – Integration gain for step-size controller. Should be less than 1/p, where p is the expected order of the scheme. Larger values lead to faster (attempted) increases in time-step size when steps are accepted. See Gustafsson, Lundh, and Soderlind, BIT 1988.
- **KP** – Proportional gain for step-size controller. Controls dependence on ratio of (error estimate)/(step size) in determining new time-step size when steps are accepted. See Gustafsson, Lundh, and Soderlind, BIT 1988.
- **max_reject** – Maximum number of rejected timesteps in a row that does not lead to a failure
- **onscale_factor** – Allowable tolerance in determining initial timestep to be “on scale”
- **safety_factor** – Safety factor used when shrinking timestep if a proposed step is rejected
- **gamma0_params** – Solver parameters for mass matrix solve when using an embedded scheme with explicit first stage
- **bcs** – An iterable of `firedrake.DirichletBC` or `EquationBC` containing the strongly-enforced boundary conditions. Irksome will manipulate these to obtain boundary conditions for each stage of the RK method.
- **solver_parameters** – A `dict` of solver parameters that will be used in solving the algebraic problem associated with each time step.

- **nullspace** – A list of tuples of the form (index, VSB) where index is an index into the function space associated with *u* and VSB is a :class: *firedrake.VectorSpaceBasis* instance to be passed to a *firedrake.MixedVectorSpaceBasis* over the larger space associated with the Runge-Kutta method

advance()

Attempts to advances the system from time *t* to time *t* + *dt*. If the error threshhold is exceeded, will adaptively decrease the time step until the step is accepted. Also predicts new time step once the step is accepted. Note: overwrites the value *u0*.

class `irksome.stage_derivative.`**StageDerivativeTimeStepper**(*F*, *butcher_tableau*, *t*, *dt*, *u0*, *bcs=None*, *solver_parameters=None*, *splitting=<function AI>*, *appctx=None*, *bc_type='DAE'*, **kwargs)

Bases: *StageCoupledTimeStepper*

Front-end class for advancing a time-dependent PDE via a Runge-Kutta method formulated in terms of stage derivatives.

Parameters

- **F** – A `ufl.Form` instance describing the semi-discrete problem F(t, u; v) == 0, where *u* is the unknown `firedrake.Function and `v` is the :class:firedrake.TestFunction`.
- **butcher_tableau** – A `ButcherTableau` instance giving the Runge-Kutta method to be used for time marching.
- **t** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time.
- **dt** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time step. The user may adjust this value between time steps.
- **u0** – A `firedrake.Function` containing the current state of the problem to be solved.
- **bcs** – An iterable of `firedrake.DirichletBC` or `EquationBC` containing the strongly-enforced boundary conditions. Irksome will manipulate these to obtain boundary conditions for each stage of the RK method.
- **bc_type** – How to manipulate the strongly-enforced boundary conditions to derive the stage boundary conditions. Should be a string, either "DAE", which implements BCs as constraints in the style of a differential-algebraic equation, or "ODE", which takes the time derivative of the boundary data and evaluates this for the stage values. Support for *firedrake.EquationBC* in *bcs* is limited to DAE style BCs.
- **solver_parameters** – A `dict` of solver parameters that will be used in solving the algebraic problem associated with each time step.
- **splitting** – An callable used to factor the Butcher matrix

- **appctx** – An optional `dict` containing application context. This gets included with particular things that Irksome will pass into the nonlinear solver so that, say, user-defined preconditioners have access to it.
- **nullspace** – A list of tuples of the form (index, VSB) where index is an index into the function space associated with *u* and VSB is a :class: *firedrake.VectorSpaceBasis* instance to be passed to a *firedrake.MixedVectorSpaceBasis* over the larger space associated with the Runge-Kutta method

get_form_and_bcs(*stages*, *tableau=None*, *F=None*)

irksome.stage_derivative.**getForm**(*F*, *butch*, *t*, *dt*, *u0*, *stages*, *bcs=None*, *bc_type=None*, *splitting=<function AI>*)

Given a time-dependent variational form and a `ButcherTableau`, produce UFL for the s-stage RK method.

Parameters

- **F** – UFL form for the semidiscrete ODE/DAE
- **butch** – the `ButcherTableau` for the RK method being used to advance in time.
- **t** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time.
- **dt** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time step. The user may adjust this value between time steps.
- **splitting** – a callable that maps the (floating point) Butcher matrix a to a pair of matrices *A1, A2* such that *butch.A = A1 A2*. This is used to vary between the classical RK formulation and Butcher's reformulation that leads to a denser mass matrix with block-diagonal stiffness. Some choices of function will assume that *butch.A* is invertible.
- **u0** – a `Function` referring to the state of the PDE system at time *t*
- **stages** – a `Function` representing the stages to be solved for.
- **bcs** – optionally, a `DirichletBC` or `EquationBC` object (or iterable thereof) containing (possibly time-dependent) boundary conditions imposed on the system.
- **bc_type** – How to manipulate the strongly-enforced boundary conditions to derive the stage boundary conditions. Should be a string, either "DAE", which implements BCs as constraints in the style of a differential-algebraic equation, or "ODE", which takes the time derivative of the boundary data and evaluates this for the stage values. Support for *firedrake.EquationBC* in *bcs* is limited to DAE style BCs.

On output, we return a tuple consisting of four parts:

- Fnew, the `Form`
- *bcnew*, a list of `firedrake.DirichletBC` or `EquationBC` objects to be posed on the stages,

4.1.21 irksome.stage_value module

`class irksome.stage_value.`**`StageValueTimeStepper`**`(`*F*, *butcher_tableau*, *t*, *dt*, *u0*, *bcs=None*, *solver_parameters=None*, *update_solver_parameters=None*, *splitting=<function AI>*, *basis_type=None*, *appctx=None*, *bounds=None*, *use_collocation_update=False*, ***kwargs*`)`

Bases: *`StageCoupledTimeStepper`*

`get_form_and_bcs``(`*stages*, *tableau=None*, *F=None*`)`

`get_update_solver``(`*update_solver_parameters*`)`

`irksome.stage_value.`**`getFormStage`**`(`*F*, *butch*, *t*, *dt*, *u0*, *stages*, *bcs=None*, *splitting=None*, *vandermonde=None*`)`

Given a time-dependent variational form and a `ButcherTableau`, produce UFL for the s-stage RK method.

Parameters

- **F** – UFL form for the semidiscrete ODE/DAE
- **butch** – the `ButcherTableau` for the RK method being used to advance in time.
- **t** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time.
- **dt** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time step. The user may adjust this value between time steps.
- **u0** – a `Function` referring to the state of the PDE system at time *t*
- **stages** – a `Function` representing the stages to be solved for. It lives in a `firedrake.FunctionSpace` corresponding to the s-way tensor product of the space on which the semidiscrete form lives.
- **splitting** – a callable that maps the (floating point) Butcher matrix a to a pair of matrices *A1, A2* such that *butch.A = A1 A2*. This is used to vary between the classical RK formulation and Butcher's reformulation that leads to a denser mass matrix with block-diagonal stiffness. Only *AI* and *IA* are currently supported.
- **vandermonde** – a numpy array encoding a change of basis to the Lagrange polynomials associated with the collocation nodes from some other (e.g. Bernstein or Chebyshev) basis. This allows us to solve for the coefficients in some basis rather than the values at particular stages, which can be useful for satisfying bounds constraints. If none is provided, we assume it is the identity, working in the Lagrange basis.
- **bcs** – optionally, a `DirichletBC` object (or iterable thereof) containing

(possibly time-dependent) boundary conditions imposed on the system.

- **nullspace** – A list of tuples of the form (index, VSB) where index is an index into the function space associated with *u* and VSB is a :class: *firedrake.VectorSpaceBasis* instance to be passed to a *firedrake.MixedVectorSpaceBasis* over the larger space associated with the Runge-Kutta method

On output, we return a tuple consisting of several parts:

- *Fnew*, the `Form`
- *bcnew*, a list of `firedrake.DirichletBC` objects to be posed on the stages,

`irksome.stage_value.`**`to_value`**`(`*u0*, *stages*, *vandermonde*`)`

convert from Bernstein to Lagrange representation

the Bernstein coefficients are [u0; ZZ], and the Lagrange are [u0; UU] since the value at the left-endpoint is unchanged. Since u0 is not part of the unknown vector of stages, we disassemble the Vandermonde matrix (first row is [1, 0, . . .]).

4.1.22 irksome.stepper module

`irksome.stepper.`**`TimeStepper`**`(`*F*, *butcher_tableau*, *t*, *dt*, *u0*, ***kwargs*`)`

Helper function to dispatch between various back-end classes
for doing time stepping. Returns an instance of the appropriate class.

Parameters

- **F** – A `ufl.Form` instance describing the semi-discrete problem F(t, u; v) == 0, where *u* is the unknown `firedrake.Function and `v` iss the :class:firedrake.TestFunction`.
- **butcher_tableau** – A `ButcherTableau` instance giving the Runge-Kutta method to be used for time marching.
- **t** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time.
- **dt** – a `Function` on the Real space over the same mesh as *u0*. This serves as a variable referring to the current time step. The user may adjust this value between time steps.
- **u0** – A `firedrake.Function` containing the current state of the problem to be solved.
- **bcs** – An iterable of `firedrake.DirichletBC` or :class: *firedrake.EquationBC* containing the strongly-enforced boundary conditions. Irksome will manipulate these to obtain boundary conditions for each stage of the RK method.
- **nullspace** – A list of tuples of the form (index, VSB) where index is an index into the function space associated with *u* and VSB is a :class: *firedrake.VectorSpaceBasis* instance to be passed to a *firedrake.MixedVectorSpaceBasis* over the larger space associated with the Runge-Kutta method

- **`stage_type`** – Whether to formulate in terms of a stage derivatives or stage values. Support for *firedrake.EquationBC* in *bcs* is limited to the stage derivative formulation.
- **`splitting`** – An callable used to factor the Butcher matrix
- **`bc_type`** – For stage derivative formulation, how to manipulate the strongly-enforced boundary conditions. Support for *firedrake.EquationBC* in *bcs* is limited to DAE style BCs.
- **`solver_parameters`** – A `dict` of solver parameters that will be used in solving the algebraic problem associated with each time step.
- **`update_solver_parameters`** – A `dict` of parameters for inverting the mass matrix at each step (only used if stage_type is "value")
- **`adaptive_parameters`** – A `dict` of parameters for use with adaptive time stepping (only used if stage_type is "deriv")
- **`use_collocation_update`** – An optional kwarg indicating whether to use the terminal value of the collocation polynomial as the solution update. This is needed to bypass the mass matrix inversion when enforcing bounds constraints with an RK method that is not stiffly accurate. Currently, only constant-in-time boundary conditions are supported.

`irksome.stepper.`**`imex_separation`**`(`*F*, *Fexp_kwarg*, *label*`)`

4.1.23 irksome.tools module

`irksome.tools.`**`AI`**`(`*A*`)`

`irksome.tools.`**`ConstantOrZero`**`(`*x*, *MC=None*`)`

`irksome.tools.`**`IA`**`(`*A*`)`

`class` `irksome.tools.`**`MeshConstant`**`(`*msh*`)`

Bases: `object`

`Constant``(`*val=0.0*`)`

`irksome.tools.`**`dot`**`(`*A*, *B*`)`

`irksome.tools.`**`flatten_dats`**`(`*dats*`)`

`irksome.tools.`**`getNullspace`**`(`*V*, *Vbig*, *num_stages*, *nullspace*`)`

Computes the nullspace for a multi-stage method.

Parameters

- **`V`** – The `FunctionSpace` on which the original time-dependent PDE is posed.
- **`Vbig`** – The multi-stage `FunctionSpace` for the stage problem
- **`num_stages`** – The number of stages in the RK method
- **`nullspace`** – The nullspace for the original problem.

On output, we produce a `MixedVectorSpaceBasis` defining the nullspace for the multistage problem.

`irksome.tools.`**`get_stage_space`**`(`*V*, *num_stages*`)`

`irksome.tools.`**`is_ode`**`(`*f*, *u*`)`

Given a form defined over a function *u*, checks if (each bit of) u appears under a time derivative.

`irksome.tools.`**`replace`**`(`*e*, *mapping*`)`

A wrapper for ufl.replace that allows numpy arrays.

`irksome.tools.`**`replace_auxiliary_variables`**`(`*F*, *u0*, *aux_indices*`)`

Discretize the fields corresponding to aux_indices in Dt(V).

`irksome.tools.`**`reshape`**`(`*expr*, *shape*`)`

`irksome.tools.`**`unique_mesh`**`(`*mesh*`)`

4.1.24 irksome.wso_dirk_tableaux module

class `irksome.wso_dirk_tableaux.`**`WSODIRK`**`(`*ns*, *order*, *ws_order*`)`

Bases: *`ButcherTableau`*

4.1.25 Module contents

PYTHON MODULE INDEX

INDEX