Spectral/*hp* element modelling in the Nektar++ framework

D. Moxey, C. Cantwell, S. J. Sherwin
Department of Aeronautics, Imperial College London

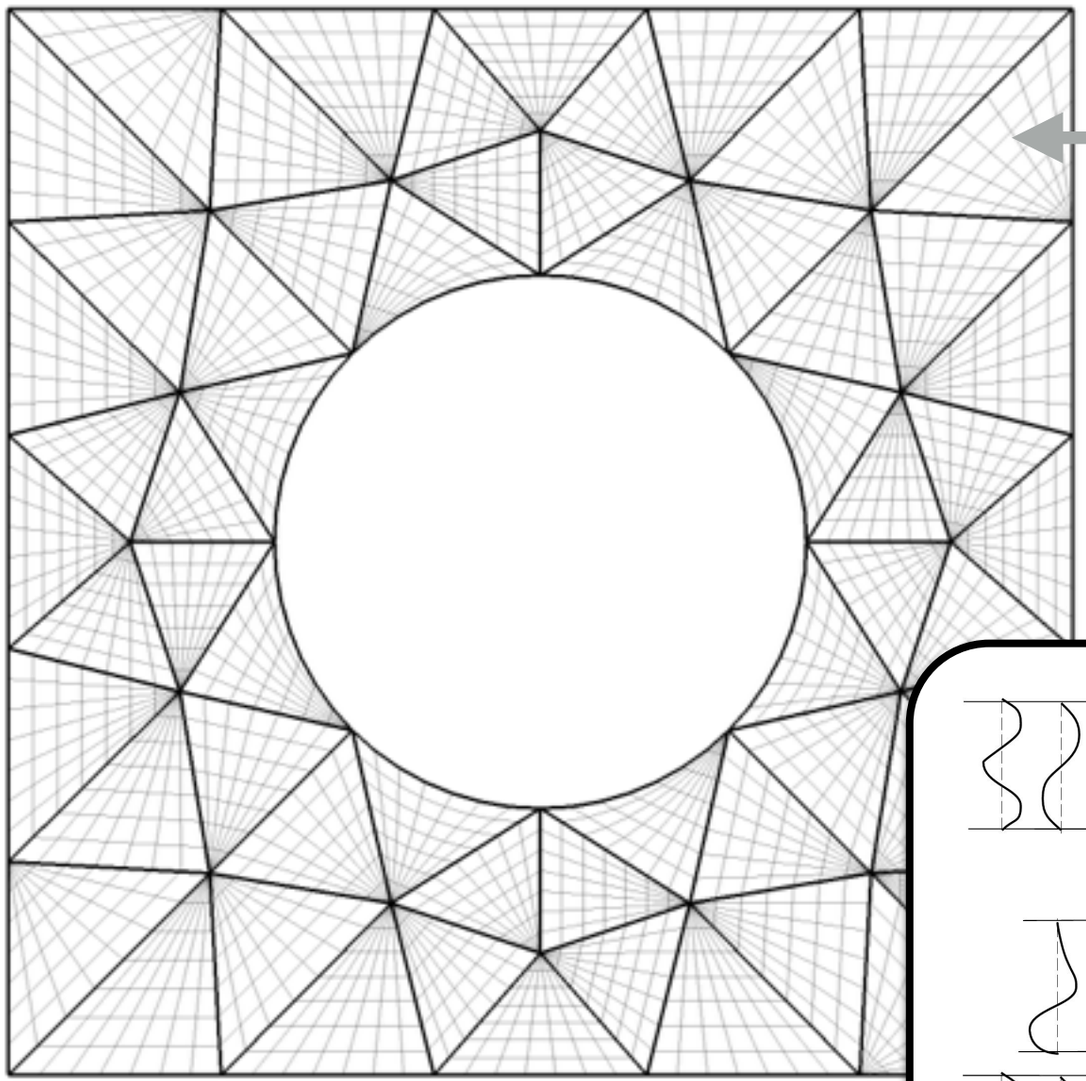
R. M. Kirby
Scientific Computing and Imaging Institute, University of Utah

SIAM CSE 2015, Salt Lake City, Utah
17th March 2015

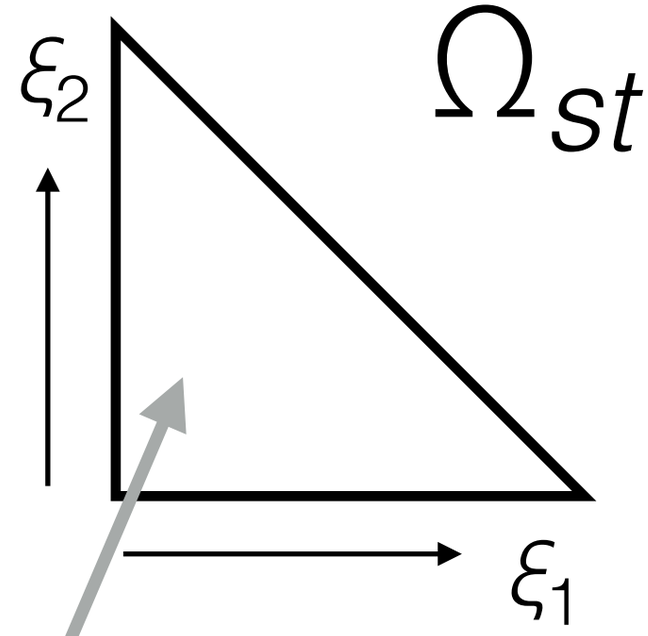
Outline

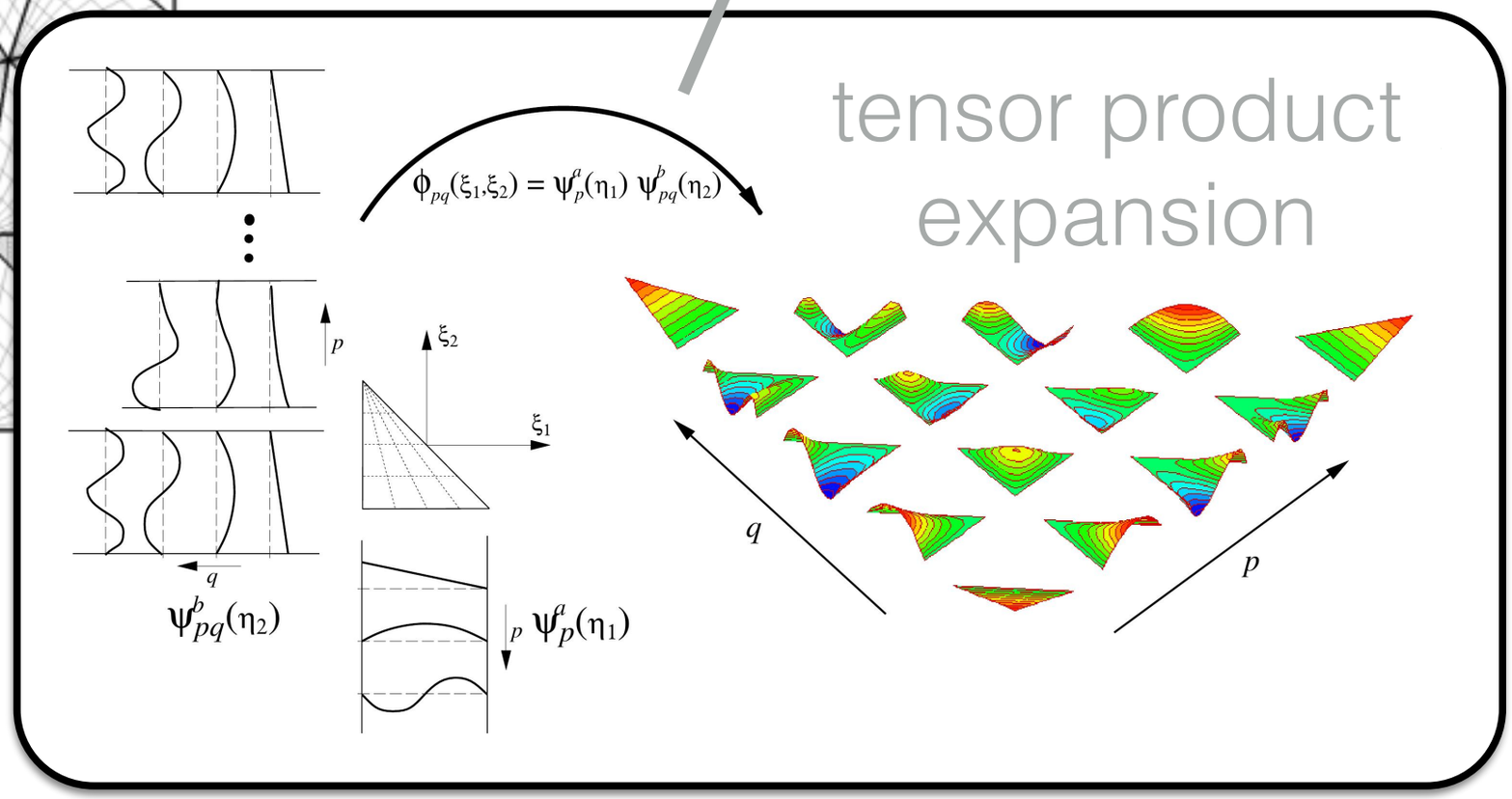
- Nektar++ framework overview
- Goals and structure
- Examples
- Conclusions

Spectral/*hp* element method



map from reference element





Nektar++ goals

- Make it simple to develop solvers for a range of fields
- Support 1/2/3D and unstructured hybrid meshes
- Scale to large numbers of processors
- Be efficient across a range of polynomial orders and core counts
- Provide a good suite of pre- and post-processing tools

Design

Consider the Helmholtz equation:

$$\Delta u + \lambda u = f$$

Put it into weak form:

$$-(\nabla u, \nabla v) + \lambda(u, v) + (\nabla u, v)|_{\partial\Omega} = (f, v)$$

Expand in terms of local (per element) or global modes:

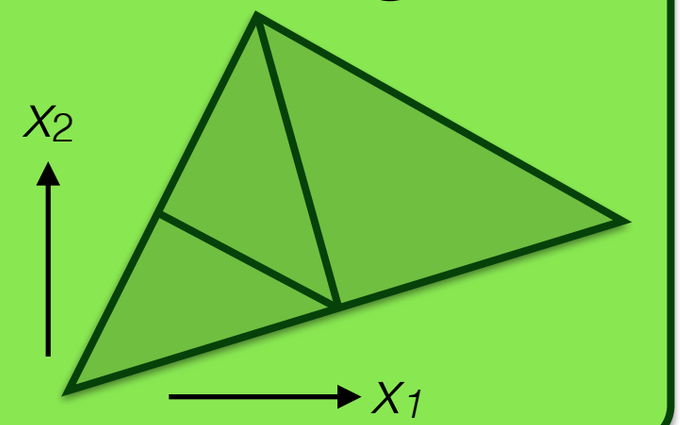
$$u_e^\delta = \sum_p \hat{u}_p \phi_p(x) \qquad u^\delta = \sum_i \hat{u}_i \Phi_i(x)$$

Framework design

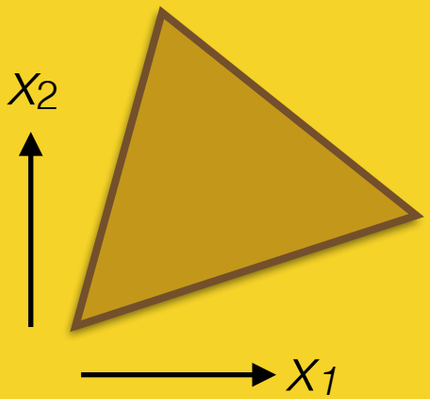
$$u^{\delta} = \sum_i \hat{u}_i \Phi_i(x)$$

$$u_e^{\delta} = \sum_p \hat{u}_p \phi_p(x)$$

MultiRegions



LocalRegions

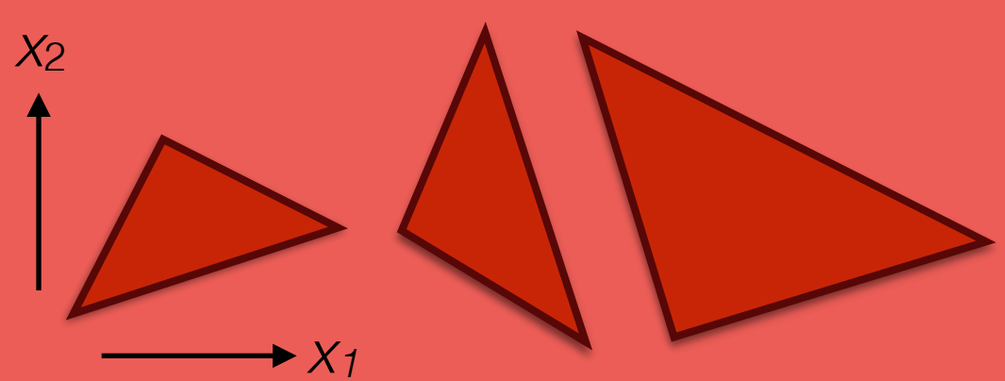


SpatialDomains

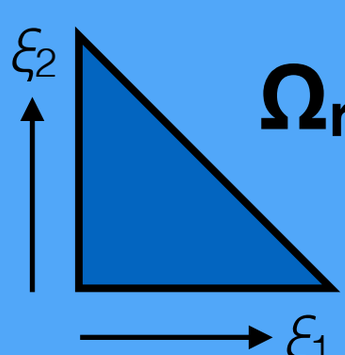
$$\mathbf{x} = X^e(\xi)$$

$$\frac{\partial x_i}{\partial \xi_j} \quad \frac{\partial \xi_i}{\partial x_j}$$

Collections



StdRegions



Framework design

IncNavierStokes **CompressibleFlow** **ADR** **ImageWarping** **...**

SolverUtils

Core Nektar++ libraries

MultiRegions **LocalRegions** **SpatialDomains**

Collections **StdRegions**

LibUtilities

Quadrature, bases, partitioning, input/output, linear algebra, interpreter, FFT, ...

Boost **Metis** **TinyXML** **Gslib** VTK PETSc ARPACK

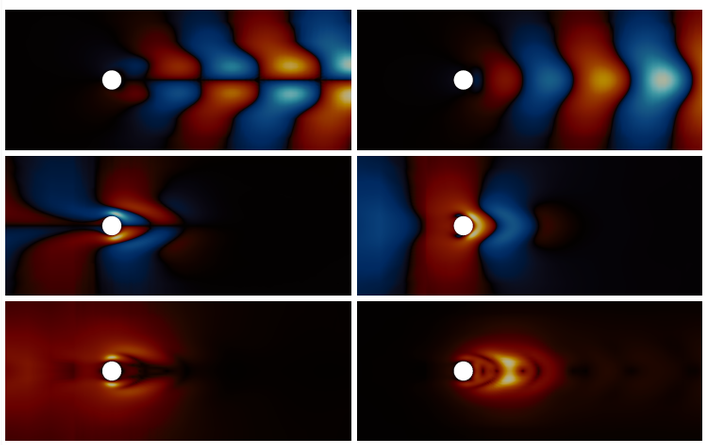
FFTW Scotch Zlib QT

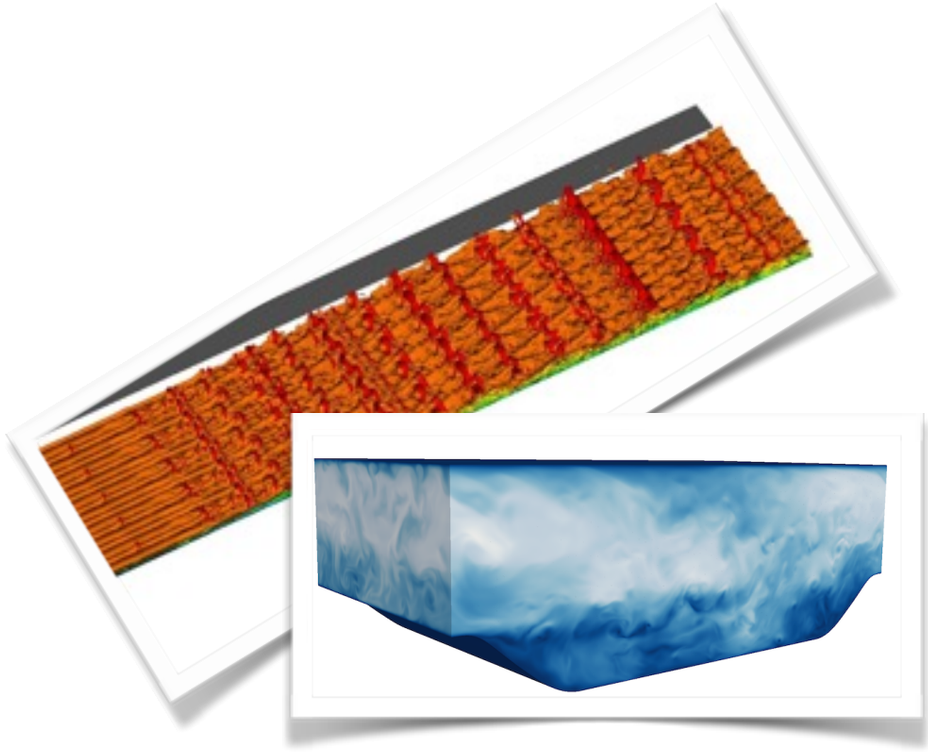
Nektar++ high-order framework

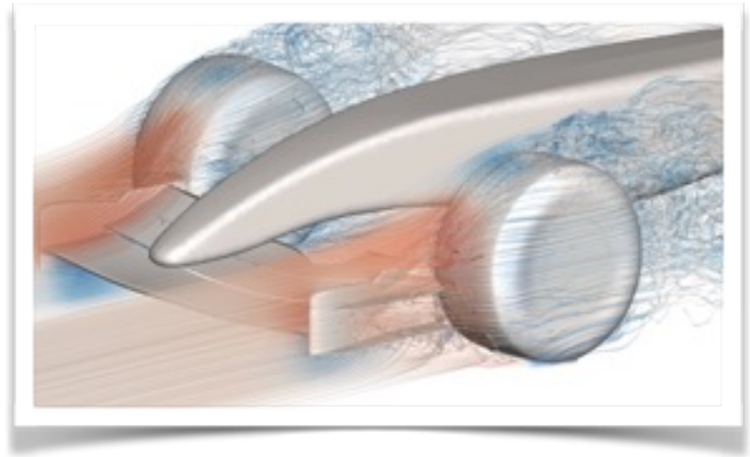
Framework for spectral(/hp) element method:

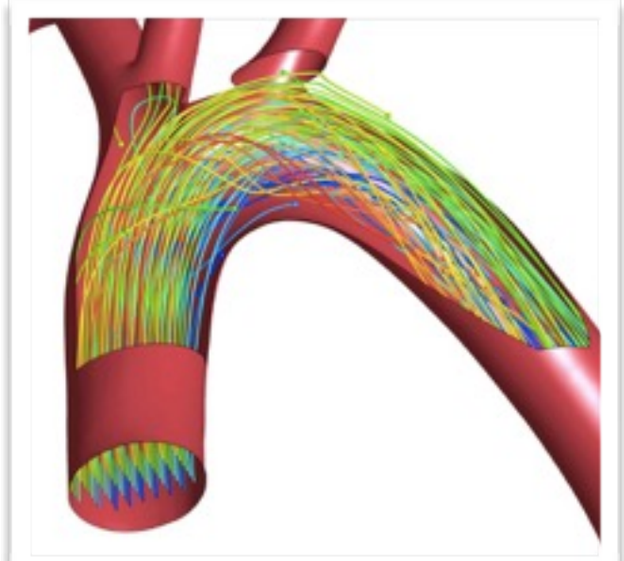
- Dimension independent, supports CG/DG/HDG
- Mixed elements (quads/tris, hexes, prisms, tets, pyramids) using hierarchical modal and classical nodal formulations
- Solvers for (in)compressible Navier-Stokes, advection-diffusion-reaction, shallow water equations, ...
- Parallelised with MPI, tested scaling up to ~10k cores

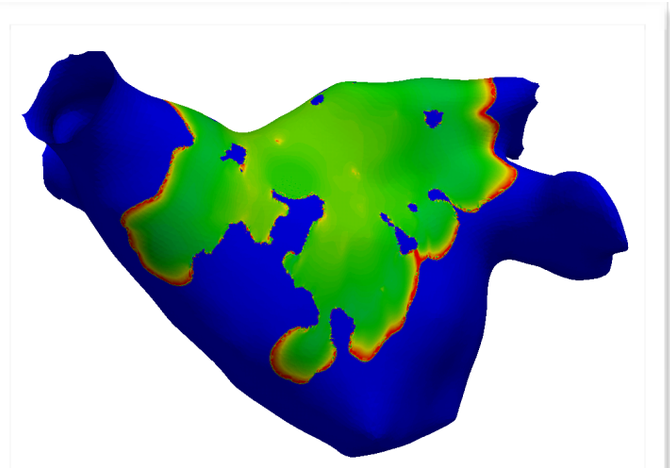
http://www.nektar.info/
nektar-users@imperial.ac.uk

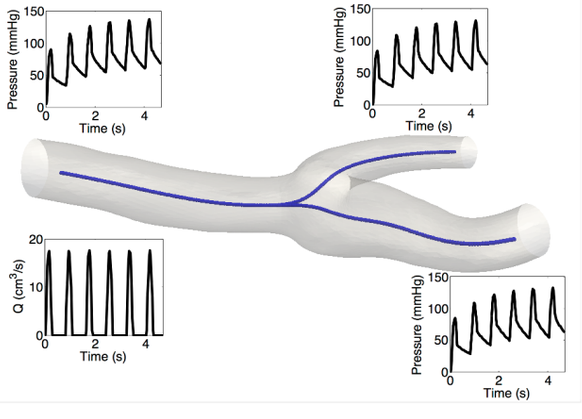


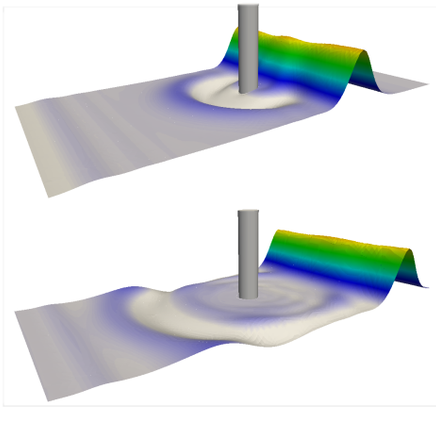


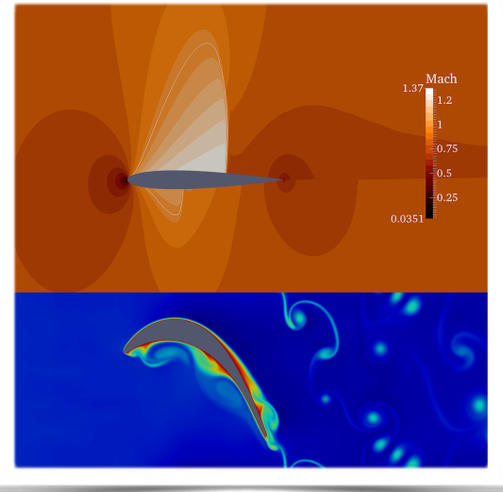


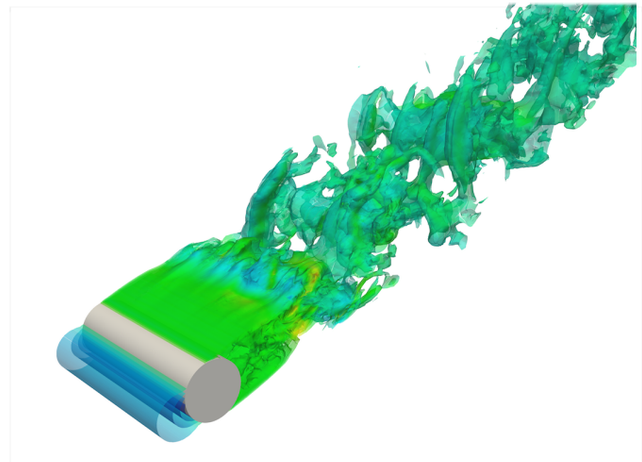





Pressure (mmHg)
150
100
50
0
0
2
4
Time (s)
Pressure (mmHg)
150
100
50
0
0
2
4
Time (s)
Q (cm3/s)
20
10
0
0
2
4
Time (s)
Pressure (mmHg)
150
100
50
0
0
2
4
Time (s)




Mach
1.37
1.2
1
0.75
0.5
0.25
0.0351



Examples

- Diffusion solver
- Multiple discretisation strategies (CG/DG)
- Experimentation and development

Diffusion solver

- Quick example: barebones diffusion solver

$$\partial_t u = \varepsilon \Delta u$$

- Uses only core library routines, CG discretisation
- Backward Euler time integration
- Main computational problem: solving linear system

$$\Delta u + \lambda u = f$$

Diffusion solver

Create session

```
session = LibUtilities::SessionReader
    ::CreateInstance(argc, argv);
```

Set up a mesh

```
string var = session->GetVariable(0);
mesh  = SpatialDomains::MeshGraph::Read(session);
field = MemoryManager<MultiRegions::ContField2D>
     ::AllocateSharedPtr(session, mesh, var);
```

Initial conditions

```
int nq = field->GetNpoints();
Array<OneD, double> x0(nq), x1(nq), x2(nq);
field->GetCoords(x0, x1, x2);
icond->Evaluate(
     x0, x1, x2, 0.0, field->UpdatePhys());
```

Parameters

```
double epsilon = session->GetParameter("epsilon");
double delta_t = session->GetParameter("delta_t");
```

Diffusion solver

Time integrate

```
for (int n = 0; n < nSteps; ++n)
{
    Vmath::Smul(nq, -1.0/delta_t/epsilon,
                    field->GetPhys(),    1,
                    field->UpdatePhys(), 1);

    field->HelmSolve(field->GetPhys(),
                     field->UpdateCoeffs(),
                     NullFlagList,
                     factors);

    field->BwdTrans (field->GetCoeffs(),
                     field->UpdatePhys());
}
```

Output

```
fldIO->Write(outFile, FieldDef, FieldData);
```

ADRSolver

- Solver for linear **a**dvection-**d**iffusion-**r**eaction
- Various support for steady/unsteady equations
- Unsteady advection and diffusion support CG and DG/HDG discretisation
- Easy to experiment with different numerical schemes
- Can change basis functions, polynomial order, timestepping scheme from through XML file

General linear methods

We can change time integration schemes by changing an entry in session file.

```
<SOLVERINFO>
  <I PROPERTY="TimeIntegrationMethod" VALUE="RungeKutta4" />
</SOLVERINFO>
```

4th order Runge-Kutta

```
<SOLVERINFO>
  <I PROPERTY="TimeIntegrationMethod" VALUE="ForwardEuler" />
</SOLVERINFO>
```

Forward Euler

General linear methods

Take an ODE problem

$$\frac{\mathrm{d}\hat{\mathbf{y}}}{\mathrm{d}t} = \hat{\mathbf{f}}(\hat{\mathbf{y}})$$

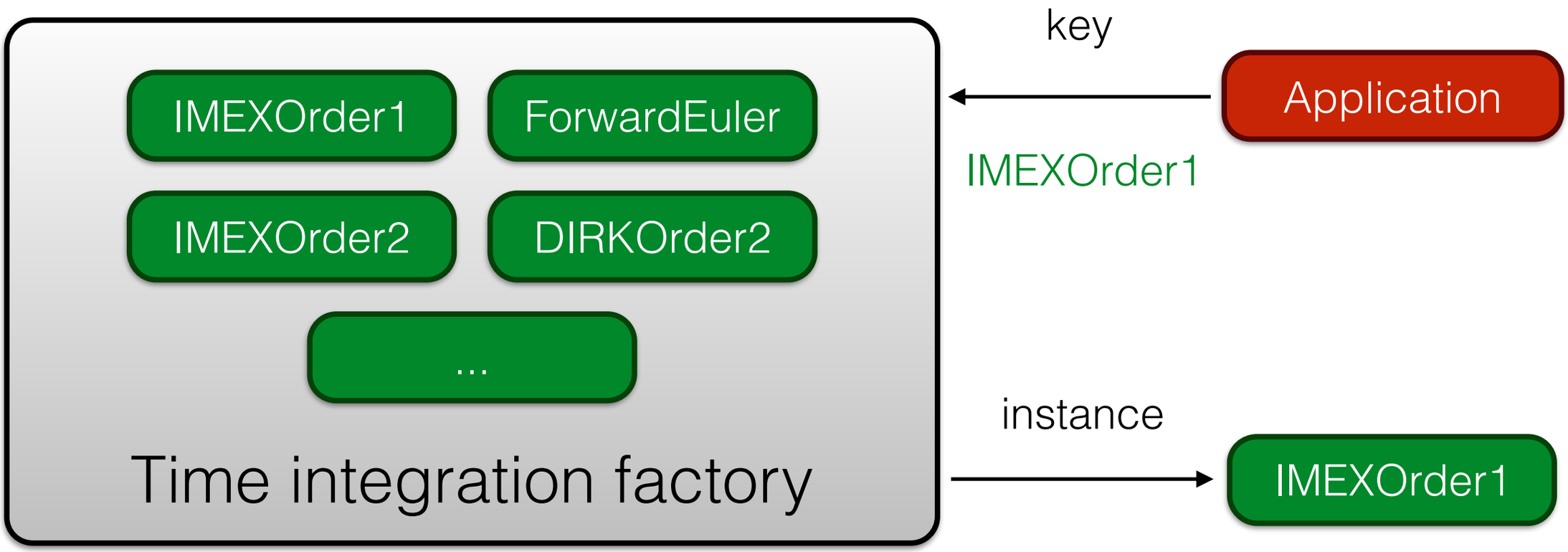
GLM for a method with r stages and s steps:

$$\mathbf{Y}_i = \Delta t \sum_{j=0}^{s-1} a_{ij}\mathbf{F}_j + \sum_{j=0}^{r-1} u_{ij}\hat{\mathbf{y}}_j^{[n-1]}, \qquad i = 0, 1, \ldots, s-1$$

$$\hat{\mathbf{y}}_i^{[n]} = \Delta t \sum_{j=0}^{s-1} b_{ij}\mathbf{F}_j + \sum_{j=0}^{r-1} v_{ij}\hat{\mathbf{y}}_j^{[n-1]}, \qquad i = 0, 1, \ldots, r-1$$

Factory patterns

Kept modular through use of factory pattern:
given a key and registered classes, return an object



Mixing discretisations

- Consider the following linear system describing a model for image warping

$$I_t + \nabla \cdot (\mathbf{u} I) = I \nabla \cdot \mathbf{u}$$

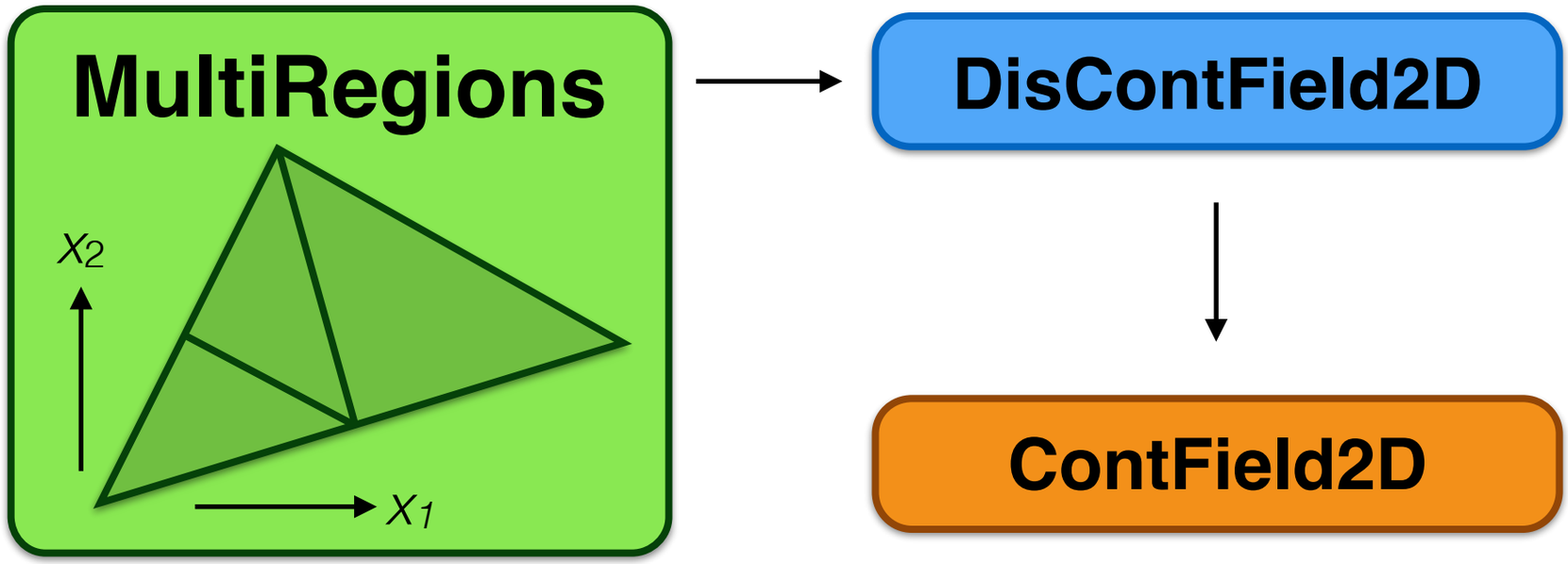
$$\phi_t + \nabla \cdot (\mathbf{u} \phi_t) = 0$$

$$(1 - \alpha^2 \nabla^2)\mathbf{u} = -\phi_t \nabla I$$

- Has 2 advection + 2 Helmholtz equations

- Appropriate schemes: hyperbolic terms in DG, elliptic solve in CG

Mixing discretisations



- Discontinuous fields hold elements, boundary conditions and can also do hybridised operators

- Continuous fields augment this with a CG assembly map, lifting Dirichlet boundary conditions, ...

Experimentation

- Quite often we want to mess around with numerical methods and surrounding infrastructure
- Might want to tie into new libraries to exploit functionality and try something out
- C++ gives great performance but can sometimes be "challenging"
- Don't want to spend hours writing interfaces without knowing it has some chance of success

Python bindings

- Nektar++ uses shared (smart) pointers extensively
- Most 'traditional' wrappers (e.g. swig) are not good at dealing with this
- Automated bindings really don't work (at least for us)
- Focus on high-quality, handwritten high-level bindings
- Use `boost::python`, has good support for inheritance, shared pointers

Syntax

```cpp
#include <LibUtilities/BasicUtils/SessionReader.h>
#include <SpatialDomains/MeshGraph.h>

session = SessionReader::CreateInstance(argc, argv);
mesh    = SpatialDomains::Read(session);
cout << mesh->GetMeshDimension() << endl;
```

C++

```python
from NekPy.LibUtilities import SessionReader
from NekPy.SpatialDomains import MeshGraph

session = SessionReader.CreateInstance(sys.argv)
mesh    = MeshGraph.Read(session)
print(mesh.GetMeshDimension())
```

Python

Example: mesh visualisation

- Curved mesh visualisation is an unsolved problem
- One strategy is to create many subdivisions
- Want to evaluate isoparametric mapping at points within reference element
- Trivially parallelisable so could use GPU for calculation + OpenGL interop to visualise

Example: mesh visualisation

- Put together across a couple of weekends
- Uses `pyopencl`, `PyQt5`, `numpy` and Nektar++ bindings
- ~1000 lines of Python, GLSL and OpenCL
- Pretty fast, shows proof of concept

Conclusions

- Nektar++ gives a open-source environment for spectral element development
- Flexible enough to allow users to select features like time integration and discretisation dynamically
- Provides a nice environment for rapid solver development, including mixing discretisations
- Python bindings are in development but show promise

Thanks for listening!

@davidmoxey

d.moxey@imperial.ac.uk

Nektar++ paper: tiny.cc/nektar