Adapted *a posteriori* meshes for Burgers' equation

Róisín Hill, Niall Madden

NUI Galway Postgraduate Modelling Research Group

08 February, 2019

Motivation

Our goal is to design an algorithm that automatically constructs layer-resolving meshes for differential equations whose solutions exhibit layer-type phenomena, without *a priori* knowledge of the location of layers. Solutions computed on these meshes should be robust for any layer width.
The main challenges to be addressed are

1. how to estimate errors in the computed solution and quantify them locally to drive the mesh adaption algorithm, and
2. how to solve the non-linear adaptivity problem.

Today I'll mainly talk about a method to solve the non-linear adaptivity problem.

Burgers' equation

Our model non-linear one-dimensional time-dependent problem is Burgers' equation,

$$u_t = \varepsilon u_{xx} - \left(\frac{u^2}{2}\right)_x, \quad \text{for } x \in (0,1), \quad t > 0, \tag{1a}$$

subject to the boundary conditions,

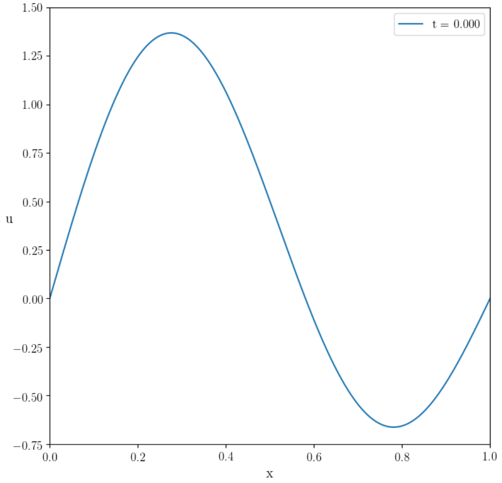
$$u(0,t) = u(1,t) = 0, \tag{1b}$$

and the initial conditions,

$$u(x,0) = \sin(2\pi x) + \frac{1}{2}\sin(\pi x). \tag{1c}$$

When $\varepsilon > 0$ is small the solution is initially smooth, but a shock develops over time.

Solution to model example, $\varepsilon = 10^{-4}$



Numerical method

We use a standard Galerkin finite element method (FEM) to compute our numerical solutions to (1). Let V be the finite dimensional subspace of $H_0^1(\Omega)$ made up of the piecewise linear functions on a mesh $\{0 = x_0 < x_1 < \cdots < x_N = 1\}$.

The weak form of (1) is: *find $u \in V$ such that*

$$F(u, v) = 0 \quad \forall v \in V, \tag{2}$$

where

$$F(u, v) = \int_\Omega \varepsilon u' v' + (u_t + uu')v dx. \tag{3}$$

In the following example we use backward difference approximation to discretise the time derivative and implement the method in FEniCS [Langtangen and Logg, 2016], a Python based system for the automatic solution of PDEs.

FEniCS code

Extract of python code to implement the method

```
# Create mesh, define function space and initial solution
mesh = UnitIntervalMesh(N)
V = FunctionSpace(mesh, "P", 1)
u0 = interpolate(u_0,V)
```

FEniCS code

Extract of python code to implement the method

```
# Create mesh, define function space and initial solution
mesh = UnitIntervalMesh(N)
V = FunctionSpace(mesh, "P", 1)
u0 = interpolate(u_0,V)

# Define test and trial function
v = TestFunction(V)
u = Function(V)
```

FEniCS code

Extract of python code to implement the method

```
# Create mesh, define function space and initial solution
mesh = UnitIntervalMesh(N)
V = FunctionSpace(mesh, "P", 1)
u0 = interpolate(u_0,V)

# Define test and trial function
v = TestFunction(V)
u = Function(V)

# Assign the boundary condition
bc = DirichletBC(V, g, "on_boundary")
```

FEniCS code

Extract of python code to implement the method

```
# Create mesh, define function space and initial solution
mesh = UnitIntervalMesh(N)
V = FunctionSpace(mesh, "P", 1)
u0 = interpolate(u_0,V)

# Define test and trial function
v = TestFunction(V)
u = Function(V)

# Assign the boundary condition
bc = DirichletBC(V, g, "on_boundary")

while t <= T:
   # Define weak form
   F = tau*epsilon*dot(u.dx(0), v.dx(0))*dx \
     + tau*u*u.dx(0)*v*dx \
     + u*v*dx \
     - u0*v*dx
```

FEniCS code

Extract of python code to implement the method

```python
# Create mesh, define function space and initial solution
mesh = UnitIntervalMesh(N)
V = FunctionSpace(mesh, "P", 1)
u0 = interpolate(u_0,V)

# Define test and trial function
v = TestFunction(V)
u = Function(V)

# Assign the boundary condition
bc = DirichletBC(V, g, "on_boundary")

while t <= T:
   # Define weak form
   F = tau*epsilon*dot(u.dx(0), v.dx(0))*dx \
     + tau*u*u.dx(0)*v*dx \
     + u*v*dx \
     - u0*v*dx

   solve(F == 0, u, bc, solver_parameters={"newton_solver": {"
        relative_tolerance": 1e-9, "absolute_tolerance": 1e-9}})

   u0.assign(u)
   t +=tau
```

FEniCS code

Extract of python code to implement the method

```
# Create mesh, define function space and initial solution
mesh = UnitIntervalMesh(N)
V = FunctionSpace(mesh, "P", 1)
u0 = interpolate(u_0,V)

# Define test and trial function
v = TestFunction(V)
u = Function(V)

# Assign the boundary condition
bc = DirichletBC(V, g, "on_boundary")

while t <= T:
   # Define weak form
   F = tau*epsilon*dot(u.dx(0), v.dx(0))*dx \
     + tau*u*u.dx(0)*v*dx \
     + u*v*dx \
     - u0*v*dx

   solve(F == 0, u, bc, solver_parameters={"newton_solver": {"
        relative_tolerance": 1e-9, "absolute_tolerance": 1e-9}})

   u0.assign(u)
   t +=tau
```

Layer-resolving meshes

a priori meshes: generated before the problem is solved, usually using specific information about solution: the location of layers and their width (e.g., Shishkin and Bakhvalov meshes).

a posteriori meshes: generated using information gained by solving the problem on an initial mesh and then refining the mesh. Methods to refine the mesh to better resolve regions of interest, e.g., where layers occur, include:

1. *h*-refinement: reduce the local mesh width (*h*) in regions where large errors are detected.
2. *r*-refinement: mesh points are relocated to regions where large errors are detected, but the total number of points is unchanged.

Layer-resolving meshes

a priori meshes: generated before the problem is solved, usually using specific information about solution: the location of layers and their width (e.g., Shishkin and Bakhvalov meshes).

a posteriori meshes: generated using information gained by solving the problem on an initial mesh and then refining the mesh. Methods to refine the mesh to better resolve regions of interest, e.g., where layers occur, include:

1. *h*-refinement: reduce the local mesh width (*h*) in regions where large errors are detected.
2. *r*-refinement: mesh points are relocated to regions where large errors are detected, but the total number of points is unchanged.

Our focus is on *r*-refinement.

Mesh generating function

Moving Mesh PDE (MMPDE) [Huang and Russell, 2011]:

The mesh generating function is obtained by integrating a parabolic differential equation, such as the heat equation,

$$x_t = \frac{1}{\rho\gamma}\left(\rho x_\xi\right)_\xi, \quad \xi \in (0,1), \qquad (4)$$
$$\text{with } x(0,t) = 0,\; x(1,t) = 1 \text{ and } x(\xi,0) = \xi.$$

Here the function $\rho(x(\xi,t),t)$ indicates where the mesh should be fine/coarse, and the constant γ controls the speed of the movement. To implement this numerically we solve (4) in its weak form,

$$\int_{\Omega_c} (\rho x_\xi v_\xi + \rho\gamma x_t v) d\xi = 0, \qquad (5)$$

on the mesh $\{\xi_i\}$.

Mesh density function

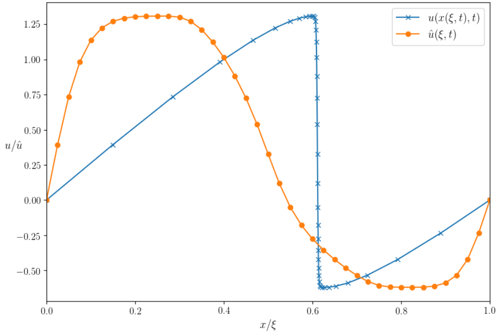
The mesh density function ρ is an arbitrary, strictly positive function defined on Ω that is used to indicate where a mesh should be fine/coarse.
Often ρ is based on a measurement of the error in the solution, or on one of its derivatives.

The resulting mesh should be denser in regions where the value of ρ is larger.

This can be understood as a coordinate transformation from the computational domain Ω_c with mesh points $\xi_i = i/N$ (uniform mesh) to the physical domain Ω with mesh points x_i.

Typical solution on Ω_c and Ω

Solution to (1) with $\varepsilon = 10^{-4}$ at time $t = 0.25$ on Ω_c and Ω



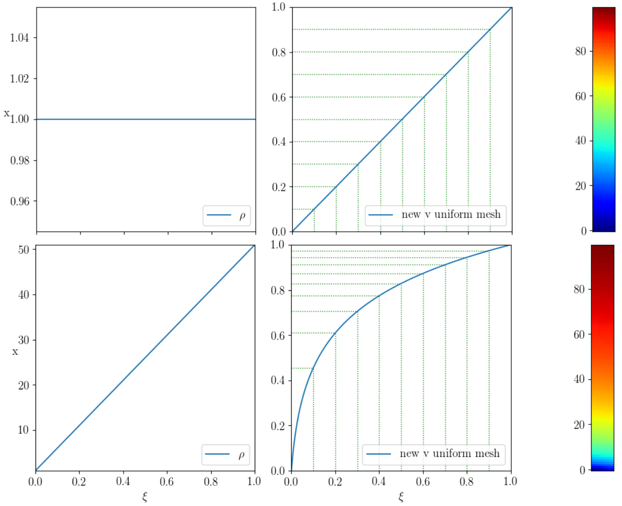
Mesh adaption using a MMPDE

Steps to generate an adapted mesh using an MMPDE

1. compute the numerical solution of (1) on an initial (uniform) mesh,
2. calculate the mesh density function ρ on each mesh interval,
3. compute the numerical solution to the MMPDE (4),
4. generate a new mesh, using the solution to (4),
5. compute the numerical solution of (1) on the new mesh, and
6. repeat steps 2 – 5 until a stopping criterion is achieved.

Why the heat equation is appropriate

Examples of two choices of ρ

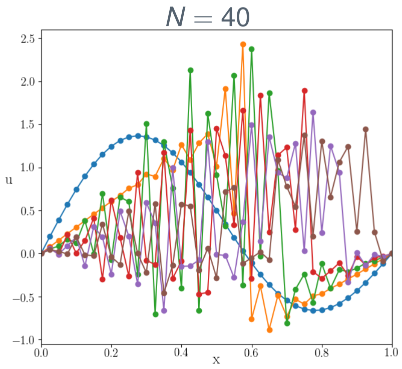


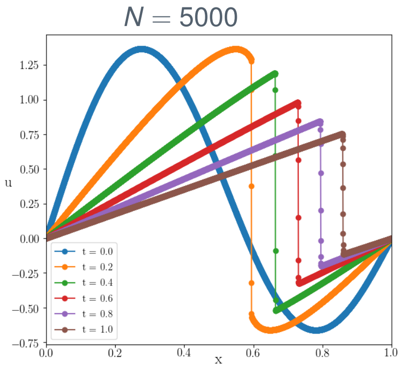
Why the heat equation is appropriate

1. Heat conducts monotonically through an insulated bar:
   $\Longrightarrow$ mesh tangling won't happen.

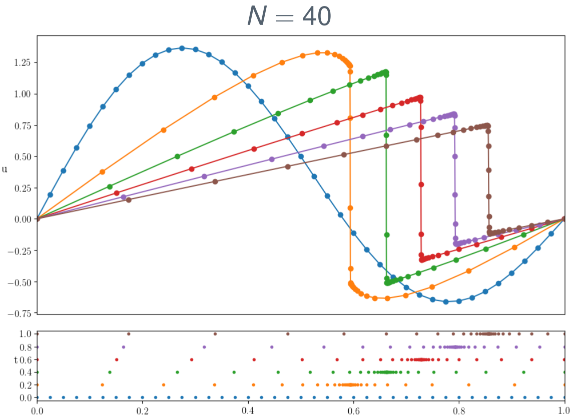
2. Changes in temperature will be small for small changes in distance:
   $\Longrightarrow$ reduces the chance of mesh starvation.

Solution to (1), uniform mesh, $\varepsilon = 10^{-4}$

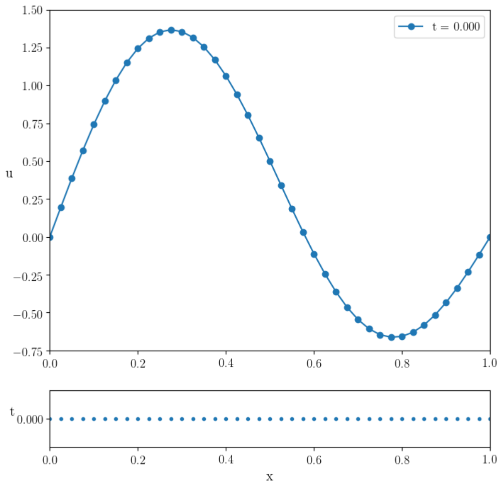




Solution to (1), adapted mesh, $\varepsilon = 10^{-4}$



Solution and mesh movement



Conclusion

1. Numerical solutions computed on uniform meshes need a large number of mesh points to resolve the interior layer region in Burgers' equation.

2. Specially fitted *a priori* meshes need to be able to track the layer, making them unsuitable.

3. We can construct suitable *a posteriori* meshes using an MMPDE which will resolve the layer region using significantly less points.

Future work

1. Investigate the suitability of other MMPDEs.
2. Explore different types of error measurements.
3. Look at other time integration methods.
4. Extend this approach to convection diffusion problems, particularly with a constricted flow.

References

[Huang and Russell, 2011] Huang, W. and Russell, R. D. (2011).
Adaptive moving mesh methods, volume 174 of *Applied Mathematical Sciences*.
Springer, New York.

[Langtangen and Logg, 2016] Langtangen, H. P. and Logg, A. (2016).
Solving PDEs in Python: The FEniCS Tutorial I.
Springer.

Acknowledgement
This project is funded by the Irish Research Council, (GOIPG/2017/463).

Thank you for listening, any questions!