Chapter 1

Example problem: Solution of the 2D unsteady heat equation with temporal adaptivity

This example illustrates `oomph-lib's` adaptive timestepping capabilities. We consider, yet again, the 2D unsteady heat equation

The two-dimensional unsteady heat equation in a square domain.

Solve

$$\sum_{i=1}^{2} \frac{\partial^2 u}{\partial x_i^2} = \frac{\partial u}{\partial t} + f\left(x_1, x_2, t\right), \qquad (1)$$

in the square domain $D = \{x_i \in [0,1]; i = 1,2\}$, subject to the Dirichlet boundary conditions

$$u|_{\partial D} = g_0 \qquad (2)$$

and initial conditions

$$u(x_1, x_2, t = 0) = h_0(x_1, x_2), \qquad (3)$$

where the functions g_0 and h_0 are given.

Here we choose the forcing function and the boundary and initial conditions so that

$$u_0(x_1, x_2, t) = \frac{1}{2}\Big(1 + \tanh\left(\gamma \cos(2\pi t)\right)\Big) \sin\left(K\left(x_1 \cos\Phi + x_2 \sin\Phi\right)\right), \qquad (4)$$

is the exact solution. The solution represents a 1D sin profile with wavenumber K, rotated against the x_1-axis by an angle Φ, and modulated by a time-periodically varying amplitude. The parameter γ controls the rate at which the amplitude changes. For large
values of γ, the amplitude remains constant for most of the period but changes rapidly at $t = (1/4 + i/2), i = 0, 1, 2, ...$ To resolve these rapid changes accurately, very small timesteps are required. Conversely, relatively large timesteps could be employed during the phases when the solution remains approximately constant. The problem therefore presents an ideal test case for an adaptive timestepping scheme.

The figure below shows a snapshot from the `animation of the exact and computed solutions,` obtained from a simulation with `oomph-lib's` adaptive `BDF<2>` timestepper. Since the time interval between

subsequent frames in this animation varies, each frame shows a blue line (in the top left corner) whose length is proportional to the elapsed time. The different rate at which the length of the line increases reflects the fact that smaller timesteps are taken when the solution varies rapidly.

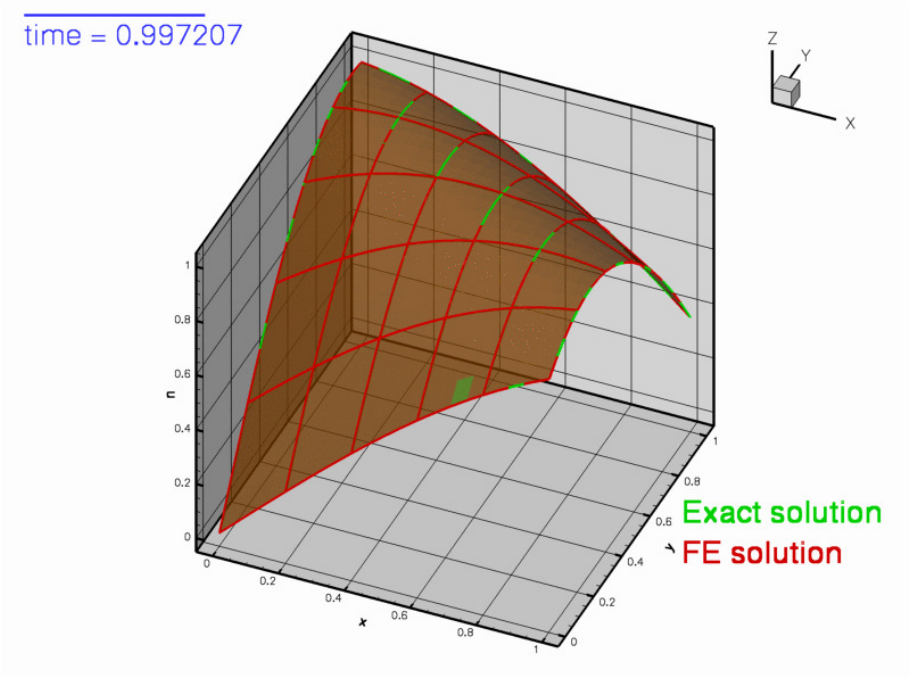

Figure 1.1 Snapshot from the animation of the exact and computed solutions.

The figure below shows the time trace of the solution and documents the timesteps chosen by the adaptive timestepping scheme. Note how smaller timesteps are chosen when the solution undergoes rapid changes.

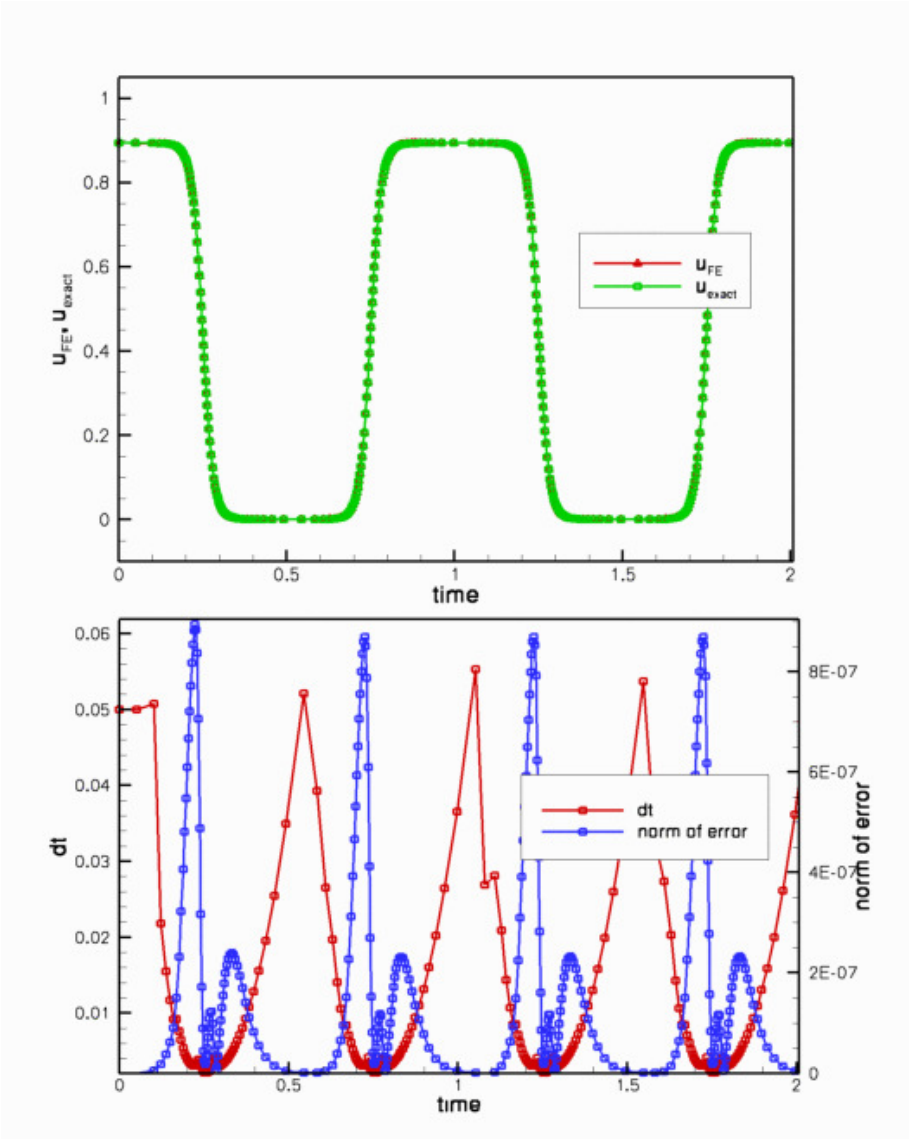

Figure 1.2 Upper plot: Time evolution of the computed and exact solutions at a control node. Lower plot: The timestep dt and the norm of the error.

Most of the driver code for this example is identical to that discussed in the previous example, therefore we only discuss the modifications required to enable temporal adaptivity:

- We pass a boolean flag to the constructor of the `BDF<2>` timestepper.
- We define a global error norm for the adaptive timestepper by overloading the (empty) virtual function `Problem::global_temporal_error_norm()`.
- We replace the call to `Problem::unsteady_newton_solve(...)` by a call to `Problem::adaptive_unsteady_newton_solve(...)` and specify a target for the temporal error norm.

1.1 Global parameters and functions

We store the problem parameters and define the source function and the exact solution in the usual namespace.

```
//======start_of_ExactSolnForUnsteadyHeat=====================
/// Namespace for forced exact solution for UnsteadyHeat equation
//============================================================
namespace ExactSolnForUnsteadyHeat
```

```
{

 /// Factor controlling the rate of change
 double Gamma=10.0;

 /// Wavenumber
 double K=3.0;

 /// Angle of bump
 double Phi=1.0;

 /// Exact solution as a Vector
 void get_exact_u(const double& time, const Vector<double>& x,
                  Vector<double>& u)
 {
  double zeta=cos(Phi)*x[0]+sin(Phi)*x[1];
  u[0]=sin(K*zeta)*
       0.5*(1.0+tanh(Gamma*cos(2.0*MathematicalConstants::Pi*time)));
 }

 /// Exact solution as a scalar
 void get_exact_u(const double& time, const Vector<double>& x, double& u)
 {
  double zeta=cos(Phi)*x[0]+sin(Phi)*x[1];
  u=sin(K*zeta)*
       0.5*(1.0+tanh(Gamma*cos(2.0*MathematicalConstants::Pi*time)));
 }

 /// Source function to make it an exact solution
 void get_source(const double& time, const Vector<double>& x, double& source)
 {
  source=
   -0.5*sin(K*(cos(Phi)*x[0]+sin(Phi)*x[1]))*K*K*pow(cos(Phi),2.0)*(
    0.1E1+tanh(Gamma*cos(0.2E1*0.3141592653589793E1*time)))-
   0.5*sin(K*(cos(Phi)*x[0]+sin(Phi)*x[1]))*K*K*pow(sin(Phi),2.0)*
   (0.1E1+tanh(Gamma*cos(0.2E1*0.3141592653589793E1*time)))+
   0.1E1*sin(K*(cos(Phi)*x[0]+sin(Phi)*x[1]))*
   (1.0-pow(tanh(Gamma*cos(0.2E1*0.3141592653589793E1*time)),2.0))*
   Gamma*sin(0.2E1*0.3141592653589793E1*time)*0.3141592653589793E1;
 }
} // end of ExactSolnForUnsteadyHeat
```

1.2 The driver code

Temporal adaptivity only requires a few straightforward changes to the time-stepping loop. Since the number of timesteps required to reach the end of the simulation is not known a priori, we replace the `for` - loop over the fixed number of timesteps, employed in the non-adaptive version of the code, by a `while` - loop that continues the time-integration until $t \geq t_{max}$.
The adaptive timestepper, `Problem::adaptive_unsteady_newton_solve(...)` takes two arguments. The first one is a suggestion for the size of the next timestep; the second specifies the target error The adaptive timestepper automatically adjusts the timestep until the error estimate computed by `Problem::global_temporal_error_norm()` is less than that target. If the error estimate for the solution computed with the suggested value of `dt` is too large, `dt` is reduced by a factor of 2 and the solution is recomputed. This process is repeated until

- the estimated error has become sufficiently small

or

- `dt` has been reduced below a threshold.

The threshold is stored in the private member data `Problem::Minimum_dt` and is initialised to 10^{-12}. This default can be changed with the access function `Problem::minimum_dt()`. It is also possible to specify a maximum value for the timestep by overwriting the default value 10^{12} for the corresponding data member, `Problem::Maximum_dt`, using the function `Problem::maximum_dt()`.
The adaptive unsteady Newton solver returns a suggestion for the size of the next timestep.
Here is the relevant code fragment from the otherwise unchanged `main` function:

```
 // Target error for adaptive timestepping
 double epsilon_t=1.0e-4;
 // Timestepping loop: Don't know how many steps we're going to take
 // in advance
 while (problem.time_pt()->time()<t_max)
  {

   // Take an adaptive timestep -- the input dt is the suggested timestep.
   // The adaptive timestepper will adjust dt until the required error
```

```
   // tolerance is satisfied. The function returns a suggestion
   // for the timestep that should be taken for the next step. This
   // is either the actual timestep taken this time or a larger
   // value if the solution was found to be "too accurate".
   double dt_next=problem.adaptive_unsteady_newton_solve(dt,epsilon_t);
   // Use dt_next as suggestion for the next timestep
   dt=dt_next;

   //Output solution
   problem.doc_solution(doc_info,trace_file);

   //Increment counter for solutions
   doc_info.number()++;
  } // end of timestepping loop
```

The rest of the main function is identical to that in the previous, non-adaptive example.

1.3 The problem class

The problem class contains a single additional member function

```
 /// Global error norm for adaptive time-stepping
 double global_temporal_error_norm();
```

which we will discuss below.

1.4 The problem constructor

The problem constructor is almost identical to that in the previous example. The only difference is that the boolean flag `true` is passed to the constructor of the `BDF<2>` timestepper, which means that a predictor step is computed for each timestep, see background for more details.

1.5 The problem destructor

The problem destructor is identical to that in the previous example.

1.6 Actions before timestep

This function is identical to that in the previous example.

1.7 Set initial condition

This function is identical to that in the previous example.

1.8 Post-processing

The Problem member function `doc_solution(...)` is virtually identical to that in the previous example. We merely add the timestep `dt` to the trace file.

1.9 Dumping the solution

This function is identical to that in the previous example, indicating that the generic `Problem::dump()` function can deal with time-dependent simulations.

1.10 Reading a solution from disk

This function is identical to that in the previous example, indicating that the generic `Problem::read()` function can deal with time-dependent simulations.

1.11 Defining the global error norm for the adaptive timestepper

1.11.1 Background

`oomph-lib's` adaptive timesteppers employ a predictor-corrector scheme to control the size of the timestep. In these schemes a low-order explicit timestepper is used to predict the solution at the next timestep. This prediction

is compared to the solution computed with the actual (usually implicit) timestepper itself. The difference between the two predictions is then used to derive an estimate of the error (exploiting the different truncation errors of the two timestepping schemes).
Interfaces to the functions that compute the temporal error estimates are defined as broken virtual function in the `TimeStepper` base class. Specific `TimeSteppers` that allow adaptive timestepping
therefore overload the broken virtual function
`TimeStepper::temporal_error_in_value(data_pt,i)`
which computes an estimate of the error of `i` - th value stored in the `Data` object pointed to by `data_pt`. In free-boundary problems in which the position of the nodes is an unknown, the corresponding function
`TimeStepper::temporal_error_in_position(node_pt,i)`
may be used to obtain an estimate of the error in the `i` - th nodal coordinate of the `Node` pointed to by `node_pt`. These individual error estimates must be combined into a problem-specific, scalar error norm, $\mathcal{E}$ say, which forms the basis of the adaptive adjustment of the timestep in `Problem::adaptive_unsteady_newton_solve(...)`.

1.11.2 Implementation

In the present problem, we choose the RMS of the errors e_j $(j = 1, ..., N_{node})$ at the nodes as the global error norm

$$\mathcal{E} = \sqrt{\frac{1}{N_{node}} \sum_{j=1}^{N_{node}} e_j^2}.$$

This may be implemented in a few lines of code:

```
//========start_of_global_temporal_error_norm=============================
/// Global error norm for adaptive timestepping: RMS error, based on
/// difference between predicted and actual value at all nodes.
//========================================================================
template<class ELEMENT>
double UnsteadyHeatProblem<ELEMENT>::global_temporal_error_norm()
{
 double global_error = 0.0;

 //Find out how many nodes there are in the problem
 unsigned n_node = mesh_pt()->nnode();
 //Loop over the nodes and calculate the estimated error in the values
 for(unsigned i=0;i<n_node;i++)
  {
   // Get error in solution: Difference between predicted and actual
   // value for nodal value 0
   double error = mesh_pt()->node_pt(i)->time_stepper_pt()->
    temporal_error_in_value(mesh_pt()->node_pt(i),0);

   //Add the square of the individual error to the global error
   global_error += error*error;
  }

 // Divide by the number of nodes
 global_error /= double(n_node);
 // Return square root...
 return sqrt(global_error);
} // end of global_temporal_error_norm
```

1.12 Comments and Exercises

As demonstrated above, enabling temporal adaptivity for a given problem is extremely straightforward as it only requires the implementation of the problem-specific function `Problem::global_temporal_error_norm()`. In most cases, the RMS of the nodal errors (or some suitable generalisation for vector-valued problems) is an obvious choice. In free-boundary problems, the RMS of the error estimate for the nodal positions is often a useful error measure.

1.12.1 How to choose the target for the temporal error norm

Having decided on an error norm, how does one choose the target for the error norm? The answer is the same as in a simulation with a fixed timestep: Trial and error, followed by careful convergence tests. We usually employ the following strategy:

- Implement the function `Problem::global_temporal_error_norm()` and perform an initial computation with a fixed timestep, choosing its size heuristically, exploiting prior knowledge that we (usually!) have about our problem. For instance, if we expect a periodic solution with an approximate period T, we may

start with a fixed timestep $dt = T/20$, say. The computed results are likely to be very inaccurate but the time-trace will usually reveal the characteristic features of the solution and thus identify phases during which the solution varies rapidly.

- Now repeat the simulation with a smaller time-step, $dt = T/40$, say, and check if any new features develop in the time-trace. If the time-trace appears to be robust, monitor the temporal error norm, e.g. by including the output of `Problem::global_temporal_error_norm()` into the trace file.
- Use the maximum of the temporal error norm observed during the simulation with the fixed timestep as the target in a first simulation with temporal adaptivity. The timestepper should now increase the size of the timestep in regions where the error estimate was small.
- Now repeat the simulation with smaller and smaller target errors until further reductions do not lead to further changes in the computed results.

1.12.1.1 Exercise:

Employ the above procedure to determine the target error $\hat{\mathcal{E}}$ required for the computed solution to be graphically indistinguishable from that obtained with a target error of $\hat{\mathcal{E}}/2$.

1.12.2 Restarting from a simulation with temporal adaptivity

We mentioned above that the data recorded/read by the generic `Problem::dump(...)` and `Problem::read(...)` functions is sufficient to restart a temporally adaptive simulations as the functions record the history values and the history of previous timesteps. Here is an illustration of a simulation that was started from the restart file produced at $t = 0.175$ in the original run. The solution computed in the restarted run follows that obtained in the original simulation but it does not employ exactly the same timesteps.

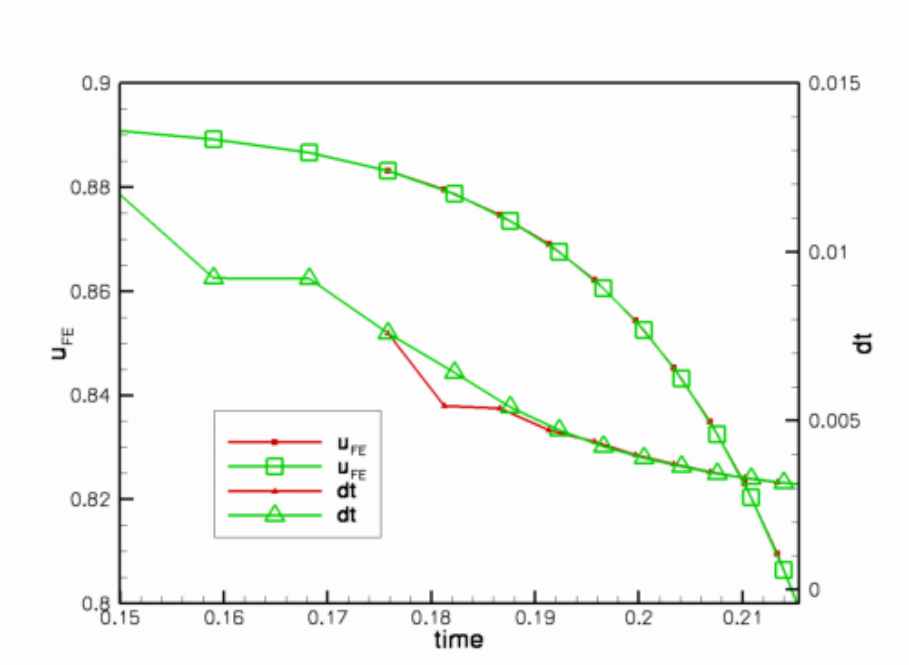

Figure 1.3 Time-trace of the solution and the timestep chosen by the adaptive timestepper in the original (green) and restarted (red) simulation.

1.12.2.1 Exercise:

Explain why the restarted simulation uses slightly different timesteps and modify the `dump_it(...)` and `restart(...)` functions to solve this problem. [**Hints: (i)** Recall that the adaptive timestepper returns a suggestion for the next timestep. This is not recorded in the restart data! **(ii)** If you can't solve the problem, have a look at the `discussion of oomph-lib's doubly-adaptive unsteady Newton solver` where an improved dump/restart procedure is implemented.]

1.13 Source files for this tutorial

- The source files for this tutorial are located in the directory:

  demo_drivers/unsteady_heat/two_d_unsteady_heat_t_adapt/

- The driver code is:

  demo_drivers/unsteady_heat/two_d_unsteady_heat_t_adapt/two_d_unsteady_heat_t_adapt.cc

---

1.14 PDF file

A pdf version of this document is available.