Sundance examples: A nonlinear initial-boundary-value problem

March 29, 2011

This example shows how to implement a nonlinear solve within a timestepping loop.

1 Statement of problem

In this example we'll solve the time-dependent radiation diffusion equation,

$$\frac{\partial u}{\partial t} = \frac{\partial^2}{\partial x^2}\left[u^4\right] + f(x,t)$$

on $[0,1]$ with a homogeneous Neumann boundary condition at $x = 0$ and an inhomogeneous Dirichlet boundary condition at $x = 1$,

$$u(0,t) = 1$$

$$u(1,t) = 1 + \epsilon \cos(2\pi t)\,.$$

where ϵ is some constant such that $|\epsilon| < 1$. This initial profile is

$$u(x,0) = 1 + \epsilon x^2.$$

Given a choice of f, finding an exact solution is difficult. To produce an exact solution for validation purposes we'll therefore use the method of manufactured solutions: choose some function $u(x,t)$ that obeys the BCs and IC, and then produce a function $f(x,t)$ so that u is the solution to the problem.

1.1 Manufactured solution

We'll choose as our solution the simple function

$$u(x,t) = 1 + \epsilon x^2 \cos(2\pi t)$$

and plug in to the radiation diffusion equation. Using Mathematica to do the differentiation, we find that for u to be a solution we must have

$$f(x,t) = \frac{\partial u}{\partial t} - \frac{\partial^2}{\partial x^2}\left[u^4\right] = -2\pi\epsilon \sin(2\pi t)x^2 - 8\epsilon\cos(2\pi t)\left(\epsilon\cos(2\pi t)x^2+1\right)^2\left(7\epsilon\cos(2\pi t)x^2+1\right).$$

2 Solution procedure

We begin with a very high-level view of an implicit time integration. A Crank-Nicolson step for some equation

$$y' = g\left(t, y\right)$$

is

$$t_{n+1} = t_n + \Delta t$$
$$y_{n+1} = y_n + \frac{\Delta t}{2}\left[g\left(t_n, y_n\right) + g\left(t_{n+1}, y_{n+1}\right)\right].$$

When g is a nonlinear function of y, we use Newton's method to solve the implicit equation

$$R\left(y_{n+1}\right) = 0 = y_{n+1} - y_n - \frac{\Delta t}{2}\left[g\left(t_n, y_n\right) + g\left(t_{n+1}, y_{n+1}\right)\right].$$

Let y_{n+1}^k be the $k-$th approximation to y_{n+1}. The Newton step w is found by solving the linear equation

$$R(y_{n+1}^k) + d_w R\left(y_{n+1}^k\right) = 0,$$

or, in matrix notation,

$$\mathbf{R}\left(\mathbf{y}_{n+1}^k\right) + J\left(\mathbf{y}_{n+1}^k\right)\mathbf{w} = \mathbf{0}.$$

Then $\mathbf{y}_{n+1}^{k+1} = \mathbf{y}_{n+1}^k + \mathbf{w}$.

The current solution y_n can be used as an initial guess y_{n+1}^0 in the Newton solve. Because the initial guess is usually quite good, line searching is usually not needed; if Newton's method has trouble converging, the first thing to try is to reduce the timestep.

The Newton solve takes us from time t_n to timestep t_{n+1}. We repeat until the desired endpoint is reached.

An industrial-strength solution algorithm should have accuracy monitoring that reduces the timestep if needed. In this example we show the simplest implementation with constant timestep.

2.1 Time-discretized equations

With the high-level view of the Crank-Nicolson-Newton loop in place, we can now look at the computation of the residual and Jacobian for this problem.

Let $u_n\left(x\right) = u\left(x, t_n\right)$ be the solution at a timestep t_n. We do Crank-Nicolson discretization in time on the strong form of the equation,

$$u_{n+1}(x) - u_n(x) = \frac{\Delta t}{2}\frac{\partial^2}{\partial x^2}\left[u_{n+1}^4(x) + u_n^4\left(x\right)\right] + \frac{\Delta t}{2}\left[f\left(t_n, x\right) + f\left(t_{n+1}, x\right)\right].$$

The unknown in this equation is u_{n+1}; the function u_n is known from a previous timestep. The BCs are to be applied to u_{n+1},

$$u_{n+1}'\left(0\right) = 0$$
$$u_{n+1}\left(1\right) = 1 + \epsilon\cos\left(2\pi t_n\right).$$

As usual, take inner products with a test function v and integrate by parts

$$(v, R) = -\left.\frac{\Delta t}{2}v\frac{\partial}{\partial x}\left(u_{n+1}^4 + u_n^4\right)\right|_0^1 + \int_0^1\left[\left(u_{n+1} - u_n\right)v + \frac{\Delta t}{2}\frac{\partial v}{\partial x}\left(\frac{\partial u_{n+1}^4}{\partial x} + \frac{\partial u_n^4}{\partial x}\right) - v\frac{\Delta t}{2}\left(f(t_n, x) + f(t_{n+1}, x)\right)\right]dx.$$

The boundary term is zero because $v \in H_0^1$, so

$$(v, R) = \int_0^1\left[\left(u_{n+1} - u_n\right)v + \frac{\Delta t}{2}\frac{\partial v}{\partial x}\left(\frac{\partial u_{n+1}^4}{\partial x} + \frac{\partial u_n^4}{\partial x}\right) - v\frac{\Delta t}{2}\left(f(t_n, x) + f(t_{n+1}, x)\right)\right]dx.$$

We must also impose the conditions $u_{n+1}(0) = 1$, $u_{n+1}\left(1\right) = 1 + \epsilon\ \cos\left(2\pi t_{n+1}\right)$.

Algorithm 1 Pseudocode for implicit time integration with Newton's method for solving the implicit equations.

```
  //Initialization steps
  Set $u_{prev}$ to initial conditions
  Set $N_{step}$ and timestep $\Delta t$
  Set $N_{newt}$, the maximum number of Newton steps allowed

  //Loop over timesteps
  for $n = 0$ to $N_{step}$ do

    $u^0_{next} \leftarrow u_{prev}$ //Use $u_{prev}$ as initial guess for Newton solve for $u_{next}$

    Initialize Newton iteration count: $k = 0$
    //Iterate Newton steps until convergence
    repeat
      Compute Jacobian $J(u^k_{next})$ and residual $r(u^k_{next})$
      Solve linear equation $J \cdot w = -r$ for the Newton step $w$
      $u^{k+1}_{next} \leftarrow u^k_{next} + w$
      $k \leftarrow k + 1$
    until $\|w\| \leq \tau$ or $k \geq N_{newt}$

    Write $u^k_{next}$ to visualization file
    $u_{prev} \leftarrow u^k_{next}$

  end for
```

2.2 Programming the problem setup

We'll describe in detail only those aspects particular to the nonlinear, transient nature of this problem. See the full source code for setup of the mesh, cell filters, and other components that have been documented in previous examples

2.2.1 Discrete function initialization

In this problem we have two nested loops: an outer loop over timesteps and an inner loop over Newton updates. We normally don't store every u_n and u^k_n. Rather, we store two values of u: the value of u at the start of each timestep, which we'll call u_{prev}, and the current Newton estimate, which we'll call u_{newt}. At the end of each timestep we write the current u_{newt} into the stored value u_{prev}. In the inner loop, at each Newton iteration we update u_{newt} by adding to it the Newton step w.

The functions u_{prev} and u_{newt} are stored as `DiscreteFunction` expression subtypes. Both "live" in the same vector space, represented as a `DiscreteSpace` object. To fill a discrete function u_{prev} with the initial profile u_{start}, we use an `L2Projector` object to do L^2 projection of u_{start} onto our discrete space, as shown in the next code block.

```
/* The initial profile is u(x,0) = 1 + εx^2. Project this onto a discrete function. This discrete
   function, uPrev, initially contains the initial conditions but will also be re-used as the
   starting value for each timestep */

/* Define an expression for the initial profile */
double epsilon = 0.25;
Expr uStart = 1.0 + epsilon*x*x;
/* Set up projection */
DiscreteSpace discSpace(mesh, bas, vecType);
L2Projector projector(discSpace, 1.0 + epsilon*x*x);
/* Carry out the projection */
Expr uPrev = projector.project();
```

We also need a discrete function for the current Newton estimate. A copy of u_{prev} will do. It's important to understand that this copy operation must *not* be done using the ordinary assignment operator, *i.e.*,

```
/*!  Do not do this!!! */
/* The assignment operator makes a shallow copy, leaving both uPrev and uNext pointing to the
    same data. Sometimes that's what you want, but in this context  death and destruction will
    result. */
Expr uNewt = uPrev;
```

The function `copyDiscreteFunction()` does the right thing, namely, it makes a shallow copy of the discrete function's supporting data such as degree-of-freedom maps, but makes a deep copy of the vector of coefficients.

```
/*  Copy $u_{\text{prev}}$ into $u_{\text{newt}}$. This copy operation makes a deep copy of the function's vector of
    coefficients, but a shallow copy of all supporting data. */
Expr uNewt = copyDiscreteFunction(uPrev);
```

The copy and the original now have distinct data vectors that can be modified independently.

2.2.2 Defining a time variable

```
/* Represent the time variable as a parameter expression, NOT as a double variable. The reason
     is that we need to be able to update the time value without rebuilding expressions. */
Expr t = new Sundance::Parameter(0.0);
Expr tPrev = new Sundance::Parameter(0.0);
```

2.2.3 Writing the weak form and nonlinear problem

Sundance has built-in differentiation capabilities, so when programming a nonlinear problem it is sufficient to write an expression for the weak form of the problem's residual. The formation of the Jacobian will be done through in-place automatic differentiation of this object. In this example, the residual is

```
/* Define the weak form */
Expr eqn = Integral(interior, v*(u-uPrev)
  + dt/2.0*(dx*v)*((dx*pow(u, 4.0))+(dx*pow(uPrev, 4.0)))
  - dt/2.0*v*(force(epsilon, x, t)+force(epsilon, x, tPrev)), quad);

/* Define the Dirichlet BC  */
Expr bc = EssentialBC(leftPoint, v*(u-1.0), quad)
  + EssentialBC(rightPoint, v*(u - 1.0 - epsilon*cos(2.0*pi*t)),quad);
```

```
/* We can now set up the nonlinear problem! */
NonlinearProblem prob(mesh, eqn, bc, v, u, uNewt, vecType);
```

2.3 Programming the solution loops

2.3.1 Preparing to loop

```
/* Create a linear solver to be used for the Jacobian solve at each Newton step */
LinearSolver<double> linSolver  = LinearSolverBuilder::createSolver("amesos.xml");

/* Allocate objects for the Jacobian, residual, and Newton step */       LinearOperator<double
    > J = prob.allocateJacobian();
Vector<double> resid = J.range().createMember();
Vector<double> newtonStep = J.domain().createMember();

/* Set parameters for the Newton loop */
int maxNewtIters = 10;
double newtTol = 1.0e-12;
```

2.3.2 Doing the loop

```
/* loop over timesteps */
for (int i=0; i<nSteps; i++)
{
  /* Set the times t_i and t_{i+1} */
  Out::root() << "timestep #" << i << endl;
  t.setParameterValue((i+1)*dt);
  tPrev.setParameterValue(i*dt);

  /* loop over Newton steps */
  bool newtonConverged = false;
  for (int j=0; j<maxNewtIters; j++)
  {
    prob.setInitialGuess(uNewt);
    prob.computeJacobianAndFunction(J, resid);
    SolverState<double> solveState = linSolver.solve(J, -1.0*resid, newtonStep);
      TEST_FOR_EXCEPTION(solveState.finalState() != SolveConverged, std::runtime_error, "
          linear solve failed!");
    addVecToDiscreteFunction(uNewt, newtonStep);
    double newtStepNorm = newtonStep.norm2();
    Out::root() << "|newt step| = " << newtStepNorm << endl;
    if (newtStepNorm < newtTol)
    {
      newtonConverged = true;
      break;
    }
  }

  TEST_FOR_EXCEPTION(!newtonConverged, std::runtime_error,
  "Newton's method failed to converged after " << maxNewtIters << " iterations");
  /* Overwrite the previous u value with the result of this timestep */
  updateDiscreteFunction(uNewt, uPrev);

  /* Write the step to a file */
  FieldWriter writer = new MatlabWriter("transientNonlinear1D-" + Teuchos::toString(i+1) + ".
      dat");
  writer.addMesh(mesh);
  writer.addField("u", new ExprFieldWrapper(uPrev[0]));
  writer.write();

  /* On to the next timestep! */
}
```

3 Postprocessing and results

The final step in the example is to compute an error metric. The exact solution is periodic in time with period $T = 1$, so $u(x, 0) = u(x, 1)$ and a useful error metric will be a comparison between the initial value and the value at the final timestep.

```
double err = L2Norm(mesh, interior, uPrev - uStart, quad);
Out::root() << "error = " << err << endl;
```

N_T		$N_X = 16, p = 1$	$N_X = 16, p = 2$	$N_X = N_T, p = 1$	$N_X = N_T, p = 2$
4		3.40917E-003	3.49630E-003	2.70007E-003	3.49667E-003
8		5.37579E-004	6.25988E-004	4.94458E-004	6.25988E-004
16		1.17545E-004	1.45331E-004	1.17545E-004	1.45331E-004
32		1.28124E-004	3.56897E-005	2.91023E-005	3.56897E-005
64		1.44033E-004	8.88304E-006	7.25920E-006	8.88303E-006
128		1.48485E-004	2.21831E-006	1.81380E-006	2.21831E-006

4 Summary of important ideas

- Write time discretization by hand, then let Sundance do spatial discretization and linearization of the weak form for you.
- You will need two discrete functions: one for the solution at the start of each timestep, one for the current Newton step. In order to preserve data integrity you must be sure to do copies and updates to these objects correctly, using the library functions
  - `copyDiscreteFunction()`
  - `updateDiscreteFunction()`
  - `addVectorToDiscreteFunction()`
- The time variable should be stored as a `Parameter` expression subtype rather than as an ordinary `double`. This lets updates to its value be immediately propagated throughout all expressions containing the time.
- Construct the `NonlinearProblem` object outside the timestepping and loop. This object can be used at all timesteps and Newton iteration. While it's possible to create a `NonlinearProblem` for each timestep or each Newton iteration, there's no need to do so, and doing so would be inefficient because bookkeeping steps such as DOF map building and expression preprocessing would be repeated unnecessarily.
- Depending on your problem size and the capabilities of your visualization system, you can write each step's result to its own file or write all steps into one file.