Nonlinear examples

June 21, 2012

Contents

1 Bratu's equation

Bratu's problem is to solve the nonlinear equation

$$-\nabla^2 u = \lambda e^u + f(x) \quad \text{in } \Omega \tag{1}$$

with boundary conditions such as

$$u = 0 \quad \text{on } \partial\Omega.$$

In one dimension with $f = 0$, the equation can be solved in closed form by an energy method. Otherwise, it is not generally tractable in closed form. However, we can always construct a solution by the method of manufactured solutions (MMS). In this set of examples, we'll produce an exactly solvable problem by the MMS, then proceed to solve it using a number of different methods.

1.1 Manufacturing a problem to match a solution

The MMS is very simple. Define a problem domain Ω, and pick any convenient function $\hat{u}$ defined on that domain. Suppose R is the operator appearing in the equation under consideration; in the case of the Bratu problem, $R(u) = -\nabla^2 u - \lambda e^u$. Construct the equation

$$R(u) = R(\hat{u})$$

and boundary conditions of a form appropriate to the problem of interest. For example, construct Robin boundary conditions as

$$\alpha u + \beta \frac{\partial u}{\partial n} = \alpha \hat{u} + \beta \frac{\partial \hat{u}}{\partial n}.$$

More generally, if the boundary conditions are nonlinear (such as radiative boundary conditions in heat transfer), construct manufactured boundary conditions

$$B(u) = B(\hat{u})$$

where B is the nonlinear operator appearing in the BCs.

By construction, $u = \hat{u}$ is an exact solution to the manufactured boundary value problem. Note that the solution may not be unique.

1.1.1 A manufactured Bratu's problem in 1D

We'll work on $\Omega = [0,1]$ with homogeneous Dirichlet boundary conditions. The function $\hat{u}(x) = \sin(\pi x)$ obeys the boundary conditions. Compute

$$\begin{aligned} R(\hat{u}) &= -\hat{u}'' - \lambda e^{\hat{u}} \\ &= \pi^2 \hat{u} - \lambda e^{\hat{u}} \end{aligned}$$

and then we've found a problem for which our solution is exact:

$$-u'' - \lambda e^u - \pi^2 \sin(\pi x) + \lambda e^{\sin(\pi x)} = 0$$

$$u(0) = u(1) = 0.$$

1.2 Weak form

Derivation of the Galerkin weak form proceeds as usual, resulting in

$$\int_0^1 v'u' - v\lambda e^u - vR(\hat{u})\, dx = 0 \quad \forall v \in H_0^1.$$

We'll look at two different nonlinear solvers applied to the Bratu problem: fixed-point iteration and Newton's method.

2 Solution by fixed-point iteration

The simplest approach to solving Bratu's problem is fixed-point iteration on the sequence of linear problems

$$-\nabla^2 u_n - \lambda e^{u_{n-1}} = R(\hat{u}) \quad \text{in } \Omega$$
$$u_n = 0 \quad \text{on } \Gamma$$

in which we have replaced the nonlinear term λe^u by its value at the previous iterate, $\lambda e^{u_{n-1}}$. We iterate until

$$\|u_n - u_{n-1}\| \leq \epsilon$$

for some specified tolerance ϵ and norm $\|\cdot\|$. The new features of this problem are

1. We need a way to represent the previous solution u_{n-1} as a symbolic object. There is an object type, `DiscreteFunction`, designed for exactly this purpose.
2. We won't store all the previous solutions: we only need one, which we'll call `uPrev`. After each step, we'll write the solution into `uPrev`, and
3. In order to check convergence, we need to specify how to compute a norm.

The program for this algorithm can be found in the source file **FixedPointBratu1D.cpp**.

The lagged function u_{n-1} is represented as a discrete function `uPrev`. A discrete function is based on a `DiscreteSpace` object, which encapsulates a mesh, one or more basis families, and a specification of the linear algebra representation to be used.

```
DiscreteSpace discSpace(mesh, basis, vecType);
Expr uPrev = new DiscreteFunction(discSpace, 0.5);
Expr uCur = copyDiscreteFunction(uPrev);
```

The discrete function has been initialized to the constant value 0.5.

With the discrete function ready, we can write the weak form and the linear problem. Notice the use of the lagged function `uPrev` in the nonlinear term.

```
Expr eqn = Integral(interior,
  (grad*v)*(grad*v) - v*lambda*exp(uPrev) - v*R), quad);
Expr bc = EssentialBC(bdry, v*u, quad);

LinearProblem prob(mesh, eqn, bc, v, u, vecType);
```

The square of the norm $\|u_n - u_{n-1}\|$ can be written in terms of a `Functional`. Note the use of the unknown function `u` in the definition of the functional. A `FunctionalEvaluator` object is created to evaluate the functional at the point `u=uCur`.

```
Expr normSqExpr = Integral(interior, pow(u-uPrev, 2.0), quad2);
Functional normSqFunc(mesh, normSqExpr, vecType);
FunctionalEvaluator normSqEval = normSqFunc.evaluator(u, uCur);
```

We now write the fixed-point iteration loop, which involves the norm check and the updating of the solution vector.

```
Out::root() << "Fixed-point iteration" << endl;
int maxIters = 20;
Expr soln ;
bool converged = false;

for (int i=0; i<maxIters; i++) {
  /* solve for the next iterate, to be written into uCur */
  prob.solve(linSolver, uCur);
  /* evaluate the norm of (uCur-uPrev) using
   * the FunctionalEvaluator defined above */
  double deltaU = sqrt(normSqEval.evaluate());
  Out::root() << "Iter=" << setw(3) << i << " ||Delta u||=" << setw(20)
       << deltaU << endl;
  /* check for convergence */
  if (deltaU < convTol) {
    soln = uCur;
    converged = true;
    break;
  }
  /* get the vector from the current discrete function */
  Vector<double> uVec = getDiscreteFunctionVector(uCur);
  /* copy the vector into the previous discrete function */
  setDiscreteFunctionVector(uPrev, uVec);
}

TEUCHOS_TEST_FOR_EXCEPTION(!converged, std::runtime_error,
  "Fixed point iteration did not converge after " << maxIters << "
     iterations");
```

If the algorithm has converged, the expression `soln` now contains the approximate solution. It can be written to a file, or used in postprocessing calculations.

3 Solution by Newton's method

When it works, fixed-point iteration converges linearly. Newton's method can converge quadratically.

In Newton's method for solving a nonlinear equation $F(u) = 0$, we linearize the problem about an estimated solution u_{n-1},

$$F(u_{n-1}) + \left.\frac{\partial F}{\partial u}\right|_{u_{n-1}} (u_n - u_{n-1}) = 0.$$

This linear equation is solved for $w = (u_n - u_{n-1})$, and then $u_n = u_{n-1} + w$ is the next estimate for the solution. Given certain conditions on the Jacobian $\frac{\partial F}{\partial u}$ and an initial guess sufficiently close to the solution, the algorithm will converge quadratically. Without a good

initial guess, the method can converge slowly or not at all. High-quality nonlinear solvers will have a method for improving global convergence. One class of globalization methods, the *line search* methods, limit the size of the step, *i.e.*, they update the solution estimate by

$$u_n = u_{n-1} + \alpha w$$

for some $\alpha \in (0, 1]$ chosen to ensure improvement in the solution. Refer to a text on nonlinear solvers (*e.g.* Dennis and Schnabel, or Kelley) for information on globalization methods.

Our example problem is to solve

$$F(u) = -\nabla^2 u - \lambda e^u - R(\hat{u}) = 0.$$

The derivative $\frac{\partial F}{\partial u}$ is the Frechet derivative, computed implicitly through the Gateaux differential

$$d_w F = \frac{\partial F}{\partial u} w.$$

Note that the Gateaux differential is exactly what appears in the equation for the Newton step, so we can write the linearized problem as

$$F(u_{n-1}) + d_w F(u_{n-1}) = 0.$$

The Gateaux differential is defined by

$$d_w F(u_{n-1}) = \lim_{\epsilon \to 0} \frac{F(u_{n-1} + \epsilon w) - F(u_{n-1})}{\epsilon}$$

from which the usual rules of calculus can be derived. For our example problem, we find

$$d_w F(u_{n-1}) = -\lambda e^{u_{n-1}} w - \nabla^2 w.$$

Therefore the linearized equation for the Newton step w is

$$\left[-\nabla^2 u_{n-1} - \lambda e^{u_{n-1}}\right] + \left[-\nabla^2 w - \lambda e^{u_{n-1}} w\right] - R(\hat{u}) = 0.$$

The linearized boundary conditions are

$$u_{n-1} + w = 0.$$

While we can do linearization by hand, it is difficult and error-prone for complicated problems. Sundance has a built-in automated differentiation capability so that linearized equations can be derived automatically from a symbolic specification of the nonlinear equations. We will show examples of Newton's method with hand-coded linearized equations and with automated linearization.

3.1 Newton's method with hand-coded derivatives

This example is found in the source file **HandLinearizedNewtonBratu1D.cpp**.

We set up a `LinearProblem` object for the linearized equations. The unknown function is the step w. The previous iterate u_n is represented by a `DiscreteFunction`. We do Newton iteration without line search. The convergence test simply uses the norm of the step vector $\mathbf{w}$, which is *not* the same as the norm of the discrete function w that uses that vector, but the difference is unimportant unless the mesh is significantly non-uniform. The method is considered converged when $\|\mathbf{w}\|_2$ goes below a specified tolerance `convTol`.

```
DiscreteSpace discSpace(mesh, basis, vecType);
Expr uPrev = new DiscreteFunction(discSpace, 0.5);
Expr stepVal = copyDiscreteFunction(uPrev);

Expr eqn = Integral(interior, (grad*v)*(grad*w) + (grad*v)*(grad*uPrev)
-   v*lambda*exp(uPrev)*(1.0+w) - v*R, quad4);

Expr h = new CellDiameterExpr();
Expr bc = EssentialBC(left+right, v*(uPrev+w)/h, quad2);

LinearProblem prob(mesh, eqn, bc, v, w, vecType);

LinearSolver<double> linSolver = LinearSolverBuilder::createSolver("amesos.xml")
    ;

Out::root() << "Newton iteration" << endl;
int maxIters = 20;
Expr soln ;
bool converged = false;

for (int i=0; i<maxIters; i++) {
    /* solve for the Newton step */
  prob.solve(linSolver, stepVal);
  /* check the norm of the step vector */
  Vector<double> stepVec = getDiscreteFunctionVector(stepVal);
  double deltaU = stepVec.norm2();
  Out::root() << "Iter=" << setw(3) << i << " ||Delta u||=" << setw(20) <<
      deltaU << endl;
  /* update the solution */
  addVecToDiscreteFunction(uPrev, stepVec);
  /* if converged, break */
  if (deltaU < convTol) {
    soln = uPrev;
    converged = true;
    break;
  }
}

TEUCHOS_TEST_FOR_EXCEPTION(!converged, std::runtime_error,
  "Newton iteration did not converge after " << maxIters << " iterations");
```

Newton's method is done. The remainder of the code is for output and postprocessing.

3.2 Newton's method with automated derivatives

One of the most useful features of Sundance is its built-in automatic differentiation capability. You can write a nonlinear PDE as a Sundance `Expr`, and Sundance will do the Newton linearization for you.

We start by writing the fully nonlinear weak form for the unknown u,

```
Expr eqn = Integral(interior, (grad*u)*(grad*v) - v*lambda*exp(u) - v*R, quad);
```

Notice that the nonlinear part is not "lagged." Write boundary conditions as well,

```
Expr bc = EssentialBC(bdry, v*u/h, quad);
```

Now we use these to construct a `NonlinearProblem` object,

```
NonlinearProblem prob(mesh, eqn, bc, v, u, uPrev, vecType);
```

As in the previous examples, `uPrev` is a discrete function containing the initial guess. Unlike previous examples, `uPrev` does not appear in the weak form; it is given as an argument to the nonlinear problem constructor.

Now we can write the Newton loop. We obtain the discrete Jacobian J as a `LinearOperator` and the discrete residual F as a Playa `Vector` through a member function of the `NonlinearProblem`. We then solve the equation $Jw = -F(u_{n-1})$ for the step vector w. The convergence check and update to `uPrev` are done as before.

```
Out::root() << "Newton iteration" << endl;
int maxIters = 20;
Expr soln ;
bool converged = false;

LinearOperator<double> J = prob.allocateJacobian();
Vector<double> residVec = J.range().createMember();
Vector<double> stepVec;

for (int i=0; i<maxIters; i++) {
  prob.setInitialGuess(uPrev);
  prob.computeJacobianAndFunction(J, residVec);
  linSolver.solve(J, -1.0*residVec, stepVec);
  double deltaU = stepVec.norm2();
  Out::root() << "Iter=" << setw(3) << i << " ||Delta u||=" << setw(20) <<
      deltaU << endl;
    addVecToDiscreteFunction(uPrev, stepVec);
  if (deltaU < convTol) {
    soln = uPrev;
    converged = true;
    break;
  }
}

TEUCHOS_TEST_FOR_EXCEPTION(!converged, std::runtime_error,
  "Newton iteration did not converge after " << maxIters << " iterations");
```

3.3 Black-box Newton-Armijo method with automated derivatives

Finally, we show the most streamlined method for setting up and solving a nonlinear problem, found in the source file **FullyAutomatedNewtonBratu1D.cpp.** The example uses a packaged nonlinear solver. Here's the meat. Simply write the nonlinear equation in boundary conditions in weak form, provide a discrete function for the initial guess, create a nonlinear problem, and hand of to a Playa `NonlinearSolver` for solution. Playa currently supports the NOX nonlinear solvers as well as its own native nonlinear solvers. In this example Playa's Newton-Armijo solver is used, with parameters set in the file `playa-newton-amesos.xml`.

```
DiscreteSpace discSpace(mesh, basis, vecType);
Expr uPrev = new DiscreteFunction(discSpace, 0.5);

Expr eqn = Integral(interior, (grad*v)*(grad*u) - v*lambda*exp(u) - v*R, quad4);

Expr h = new CellDiameterExpr();
Expr bc = EssentialBC(left+right, v*u/h, quad2);

NonlinearProblem prob(mesh, eqn, bc, v, u, uPrev, vecType);

NonlinearSolver<double> solver = NonlinearSolverBuilder::createSolver("playa-
    newton-amesos.xml");

Out::root() << "Newton solve" << endl;
SolverState<double> state = prob.solve(solver);

TEUCHOS_TEST_FOR_EXCEPTION(state.finalState() != SolveConverged,       std::
    runtime_error,
  "Nonlinear solve failed to converge: message=" << state.finalMsg());
```

Unless your problem needs special handling, the fully automated approach is usually the most efficient and robust against user error.

4 Continuation on a parameter

Bratu's problem becomes more strongly nonlinear as the parameter $|\lambda|$ increases. A reasonable strategy in such problems, called *continuation*, is to begin with a parameter setting for which the problem is exactly or nearly linear, then solve a sequence of problems in which the parameter is adjusted systematically for each new solve. This can be helpful in problems where the nonlinear solver has a small radius of convergence.

Source code is in **ContinuationBratu1D.cpp**.

The key is to write the adjustable parameter as a `Parameter` object rather than a double. The parameter value can be modified externally by calling the `setParameterValue` member function (as shown in the loop below) and the change in the parameter is automatically reflected in the shallow copies of the parameter stored within a problem's expressions.

```
Expr lambda = new Sundance::Parameter(0.0);
```

The continuation loop is shown below; in each iteration, a new value of λ is chosen, and then a nonlinear solve is carried out at that value. The result (stored in the discrete function `uPrev`) is then used as the initial guess for the next continuation step.

```
for (int n=0; n<nSteps; n++) {
  double lambdaVal = n*lambdaMax/(nSteps-1.0);
  /* update the value of the parameter */
  lambda.setParameterValue(lambdaVal);
  Out::root() << "continuation step n=" << n  << " of " << nSteps << ", lambda="
      << lambdaVal << endl;
  SolverState<double> state = prob.solve(solver);
  TEUCHOS_TEST_FOR_EXCEPTION(state.finalState() != SolveConverged,          std::
      runtime_error,
    "Nonlinear solve failed to converge: message=" << state.finalMsg());
  Expr soln = uPrev;
  /* === Solution output code omitted === */
}
```

This example shows only the simplest continuation method: increasing the parameter in uniform steps. More sophisticated strategies are possible.

5 Exercises

1. Write a solver for Bratu's equation with no forcing ($f = 0$) on the unit square with boundary conditions $u = 0$. Use continuation to explore the behavior as λ is increased from zero. Over what range of parameters are you able to find a solution?

2. The continuation example above was built around the fully-automated Newton's method. However, continuation strategies could be used with any of the other approaches considered here, such as fixed-point methods or a hand-coded Newton's method. Modify one of those examples to implement a continuation strategy.

3. Consider the steady-state radiation diffusion equation,

$$\nabla^2 \left(u^4 \right) = 0$$

   on the unit square with boundary conditions

$$u = (1 + \sin(\pi x))^{1/4} \quad \text{along the line } y = 1$$

$$u = 1 \quad \text{elsewhere on } \Gamma.$$

   (a) Derive a weak form of the problem

   (b) Derive a linearized weak form for the Newton step $w = u_n - u_{n-1}$.

   (c) Suppose you were to use an initial guess $u_0 = 0$ in Newton's method. What would happen, and why?

   (d) Write a program to set up and solve this problem using Newton's method.