Parabolic Equation in One Space Dimension

This project is due at 11 PM on Sunday, May 8.

You are to implement and test a solver for a scalar one dimensional reaction diffusion equation. This project uses a Galerkin method based on continuous piecewise linear functions to model diffusion. The technique to be used here will be the basis for a more sophisticated solver developed in the next project, so it is important that you complete this project.

Let $I = (a, b)$ and $J = (0, T)$, where $a < b$ and $0 < T$. Denote by Q the set $I \times J = \{(x, t) : x \in I, t \in J\}$. The solver will approximate the solution, $u(x, t)$, of an initial/boundary value problem. The differential equation is

$$c\partial_t u - \partial_x(d\partial_x u) = f \text{ on } Q, \tag{1}$$

where $c = c(x)$, $d = d(x)$, and $f = f(x, u)$ are given. In this equation ∂_t and ∂_x are shortened notations for $\frac{\partial}{\partial t}$ and $\frac{\partial}{\partial x}$, respectively. We assume that c and d are continuous on $[a, b]$, except for a finite number of jump discontinuities, and that c and d are positive on (a, b). In addition we assume that f is differentiable with respect to u.

The initial conditions for u are defined by a given function u_0:

$$u(x, 0) = u_0(x) \text{ on } I. \tag{2}$$

There is one boundary condition on each end of (a, b) for each time $t \in J$, but it can be of two different forms. At $x = a$ the condition can be either $u(a, t) = u_a(t)$ or $-d(a)\partial_x u(a, t) = u_a(t)$, where u_a is given. Similarly at $x = b$ the condition is either $u(b, t) = u_b(t)$ or $-d(b)\partial_x u(b, t) = u_b(t)$, where u_b is given. The conditions that set the value of u are called essential or Dirichlet boundary conditions and the conditions that set the derivative are called natural or Neumann boundary conditions. We will refer to the boundary conditions as DD, DN, ND, or NN to indicate the four possible types; DD means Dirichlet at each end, DN means Dirichlet at $x = a$ and Neumann at $x = b$, etc.

Continuous-Time Galerkin Method

Take $\sigma = \{a = x_0 < x_1 \ldots < x_N = b\}$ to be a partition of $[a, b]$ and let $\mathcal{M}$ be the space of all continuous functions on $[a, b]$ which are polynomials of degree at most one on each $I_i = (x_{i-1}, x_i)$. A function W in $\mathcal{M}$ can be represented as

$$W(x) = \sum_{i=0}^{N} W(x_i)\varphi_i(x), \tag{3}$$

where φ_i is the member of $\mathcal{M}$ which is 1 at x_i and 0 at all x_j for $j \neq i$.

We will switch between thinking of $W \in \mathcal{M}$ as an N+1-vector and a function of x. The approximate solution $U(t)$ will sometimes be viewed as a function of time with values in $\mathcal{R}^{N+1}$ and other times as a function $U(x,t)$ such that the value at each time is an element of $\mathcal{M}$. Hopefully the context will make it clear which view to take.

To define the continuous-time Galerkin method we first consider the case in which the boundary conditions are both Neumann. In this case the approximate solution $U(t)$ in $\mathcal{M}$ satisfies

$$\int_a^b (c\,\partial_t U\varphi_i + d\,\partial_x U\varphi_i' - f\varphi_i)dx - u_a\varphi_i(a) + u_b\varphi_i(b) = 0 \tag{4}$$

for $i = 0, \ldots, N$ and all $t \in J$. The function f is evaluated as $f(x, U(x))$. If the boundary condition at a is of Dirichlet type we replace the first equation above, i.e. $i = 0$, with

$$U(a,t) = u_a(t).$$

Similarly, if the boundary condition at b is of Dirichlet type we replace the last equation above, i.e. $i = N$, with

$$U(b,t) = u_b(t).$$

The integrals involved in the system should be approximated by assuming that the variable terms ($c(x)$, $d(x)$, and $f(x, u(x))$) are constant on each subinterval I_i. Define $U(0)$ by taking in initial value to agree with u_0 and each of the points in the partition.

Backward Difference Discrete-Time Galerkin

The differential equations in the continuous-time Galerkin method are stiff. In this project you are to implement a backward difference-like procedure to approximate the solution.

The basic step advances from t_{now} to $t_{new} = t_{now} + dt$. Let U denote the value of the discrete solution at time t_{now} and let dU be the change over the step of the vector U.

In order to define the change over the step we first define a function called `resid` that takes dU as one of its arguments and returns an $N+1$-vector R. In the NN case the $R[i]$ is given by

$$\int_a^b ((c/dt)dU\varphi_i + d\,\partial_x W\varphi_i' - f\varphi_i)dx - u_a\varphi_i(a) + u_b\varphi_i(b) \tag{5}$$

where the integrals are approximated as discussed above, W is $U + dU$, f is evaluated as $f(x, W(x))$, and u_a and u_b are evaluated at t_{new}. The function `resid` is defined for all dU, not just the one that ends up being the change over the step; you can think of it as a trial value for the change over the step. If one or both of the boundary conditions are Dirichlet then the first and/or last component of R is changed to $W[0] - u_a(t_{new})$ or $W[N] - u_b(t_{new})$, as appropriate.

The backward difference method would solve for a dU such that the vector R returned by `resid` is zero. For this project the nonlinear problem will be replaced by a linearized version of it, where the linear equations can be solved using `solve_banded` from Scipy.

The map from dU to R has a Jacobi matrix that is tridiagonal, and this allows us to compute the approximate Jacobi matrix with only 4 calls to `resid`. The technique for doing this will be discussed below. Let C be the approximate Jacobi matrix for this map and let R_{base} be the return from `resid` with input $dU = 0$. Then the change over the step will be defined by

$$R_{base} + C\,dU = 0.$$

Construction of C

Let C be the approximate Jacobi matrix for this mapping; that is,

$$C[i,j] \approx \frac{\partial R[i]}{\partial dU[j]}.$$

Let $Cdata$ be the 3×(N+1) array that holds the diagonals in the form needed for `solve_banded`: it is only $Cdata$ that we need to find. The association between C and $Cdata$ is

$$\begin{aligned} C[i-1,i] &= Cdata[0,i] \\ C[i,i] &= Cdata[1,i] \\ C[i+1,i] &= Cdata[2,i]. \end{aligned}$$

You should check the documentation to see that this is correct.

The technique for approximating C involves the fact that if we perturb dU only at location i then the return value is only perturbed at locations $i-1$, i, and $i+1$. (When $i = 0$ or $i = N$ there are only two perturbed returned values.)

Let `epsilon` be a small number, say $1e-6$. Pseudo code that computes $Cdata$ is as follows:

```
dU = 0
Rbase = resid(dU)
for k = 0, 1, 2
  dU = 0
  for i in range(k, N+1, 3)
    dU[i] += epsilon
  R = (resid(dU) - Rbase)/epsilon
  for i in range(k, N+1, 3)
    Cdata[0,i] = R[i-1]
    Cdata[1,i] = R[i]
    Cdata[2,i] = R[i+1]
```

In using this pseudo code you will need to expand the call to resid to include all its arguments, and you will need to check that R is not used with an index outside of $[0, N]$.

Demonstrating Your Code

In these cases use $[a, b] = [0, 1]$, a mesh with $N = 30$ intervals and $x[i] = p(i/N)$, where $p(x) = x + 0.9 * x * (1-x)$. The plots should have titles and/or captions that indicate what the plot is showing.

Case 0. Use NN boundary conditions with $u_a(t) = u_b(t) = 1$ and initial data $u_0(x) = x$. Take $c(x) = d(x) = 1$ and $f(x, u) = 0$. Use $dt = 0.05$. Plot the solution at $t = 0, 0.05, 0.1, 0.2, 1.0$. The solution should converge to $1-x$.

Case 1. Use ND boundary conditions with $u_a(t) = 0$, $u_b(t) = 1$ and initial data $u_0(x) = 0$. Take $c(x) = d(x) = 1$ and $f(x, u) = 0$. Use $dt = 0.01$. Plot the curves $u(0, t)$ for $t \in [0, 2]$ and $u(x, 2)$ for $x \in [0, 1]$.

Case 2. Like Case 1, except use $c(x) = d(x) = x$.

Case 3. Like Case 1, except use $c(x) = d(x) = x^2$.

Case 4. Like Case 0, except use $u_0(x) = 1$ and $f(x, u) = u$. The solution of the differential problem is $u(x, t) = exp(t)$

Some Implementation Notes

I like to encapsulate the parabolic problem as a object. The problem includes

- the mesh,
- the functions, c, d, and f,
- the initial value u_0, and
- the boundary condition type and data for a and for b.

The computation of the R returned from `resid` is easily done by first looping over the subintervals of the mesh, ignoring the boundary conditions, and then modifying the $R[0]$ and/or $R[N]$ values based on the boundary conditions. The arguments to `resid` should probably include U, dU, dt, $time$, and a parabolic problem object.

Last modified 5/3/22 – tfd