Solver for Evolution Equations in One Space Dimension

This project is due at 11 PM on Tuesday, May 31.

This project generalizes your solution to project 4 to allow for systems of time dependent PDE's in one space dimension.

In this project you are to aim at making a "package" that approximates the solutions of problems of this class with minimal changes to low level code. In particular, there should be a class that is called `Problem_Def` that contains the information that defines the differential problem and its spatial discretization, and a class called `simTime` that contains the information used in the adaptive time stepping process. These classes are similar to the classes used in project 4.

You should have a function that initializes the state of the approximate solution (based on the information in a `Problem_Def` object) and initializes the simTime object that will guide your time integration. I often call this function `setup`.

You will exercise this more general code on two examples, one that is a reaction diffusion problem and one that is version of the second-order wave equation. In each case there will be two differential equations that are coupled. The spatial discretization on the first example will be based on a Galerkin scheme, and in the second it will be a staggered-grid finite difference method. These two very different systems and discretization methods give a hint of the generality of the tool that you have built.

Problem Definition

The `Problem_def` class should contain

1. functions used in the system of PDE's
2. initial values of the solution
3. functions and variables used in the boundary conditions
4. the mesh
5. the number of components in the solution

6. the half bandwidth of the Jacobi matrix of the `Resid` function

The only things that are new here, when compared to project 4, are the number of components in the solution and the half bandwidth; I'll call these `vars` and `hbw`, respectively, in this write-up. The `hbw` is the number of diagonals above or below the main diagonal of the Jacobi matrix for the `Resid` function. In project 4, you had (implicitly) `hbw` $= 1$, because the matrix in question was tridiagonal. Here, as in project 4, the Jacobi matrix is formulated for `Resid` as a function of the change in the solution over the step. It is desirable to order the unknowns and the equations so that the bandwidth is as small as possible.

For this project the mesh is the same for each component of the solution.

Stacked versus Interwoven

Think about the case where there are two components to the solution, I'll call them z and q. The solution consists of two functions $z(x,t)$ and $q(x,t)$. For a given time t the approximations to z and q will each be stored in an array of length $N+1$, where N is the number of intervals in our mesh. Call these arrays `Z` and `Q` We will suppose that the state of the system at each time t is stored in a 2-d numpy array `U` that has two rows, where `Z` is the first row and `Q` is the second row. Thus we can write

```
Z = U[0]
Q = U[1]
```

The function `Resid` will take `U`, `dU`, and a few other arguments and return a $2\times(N+1)$ numpy array `R` that is the residual for the backward difference equation for advancing time. There is a discussion below that indicates how this works in the examples for this project. The two rows of `R` we will think of as being the residuals for the `Z` and `Q` equations, and inside `Resid` we will address these rows as `RZ` and `RQ`. This way of storing the solution and the residual will be referred to as the **stacked** form.

When we are approximating the Jacobi matrix and trying to make it have a small half bandwidth, we will think of both the solution `U` and the residual `R` in a different way. The rows of a stacked version will be interwoven to form a 1-d numpy array. In the case of `U`, with $N = 4$, the interwoven version RI will be

```
UI = [Z[0], Q[0], Z[1], Q[1], Z[2], Q[2], Z[3], Q[3], Z[4], Q[4]]
```

In python this can be expressed as

```
UI = U.reshape(-1,order='F')
```

This does not actually create a new copy of `U`, just a new way to address its entries. If we have an interwoven version of `dU` called `dUI` then `dU = dUI.reshape(2,N+1,order='F')`

Forming the Approximate Jacobi Matrix

In project 3 you approximated the Jacobi matrix by using 4 calls to `Resid`; the first with `dU = 0` and the remaining 3 had `dU[i]` set to a small number for `i in range(k,N+1,3)` for $k = 0, 1, 2$. In this project the bandwidth of the Jacobi matrix will be `bw = 1 + 2*hbw`, and you will need `bw + 1` calls to `Resid` to do the generalization of what was done in the tridiagonal case.

Advancing Time

You should use the SDoLE procedure from project 4 to advance time.

Example 1, Solid-Solid Combustion Model

This example is a simple model of combustion. In this model z will represent temperature and q will represent the concentration of the limiting reactant. (This is different from the notation I used in discussing this model some of the time. Sorry.) The spatial domain is $I = (0, 10)$. The mesh is $N = 100$ uniform intervals: $x[i] = i * dx = i * (10/100)$. The differential system is

$$\partial_t z - \partial_x(d(x)\partial_x z) = f(z, q) \tag{1}$$

$$\partial_t q = -0.5 f(z, q) \tag{2}$$

where $f(z, q) = z * q$, when z and q are nonnegative, and is zero otherwise. The boundary conditions are

$$z(0, t) = 1.0 - exp(-5 * t) \tag{3}$$

$$\partial_x z(10, t) = 0. \tag{4}$$

The initial conditions are $z(x,0) = 0$ and $q(x,0) = 1$ for all $xinI$. The function $d(x)$ is constant with value one.

The spatial discretization should be done using a Galerkin process that is like that used in projects 3 and 4; this should be applied to both z and q. The `Resid` function returns `R` a $2\times(N+1)$ numpy array. The first row, should contain the residuals for the z equations, including the Dirichlet boundary condition. The second row should contain the residuals for the q equations. These are built (separately or together) working by intervals across the domain.

I suggest that the time stepping be done with `dtmin=1e-4`, `dtmax=0.1`, and `tol = 0.01`. The error indicator should be the max norm on the difference between to single and double step of SDoLE.

You should produce 11 plots showing $z(x,t)$ and $q(x,t)$ at $t = 0, 1, 2, \ldots, 10$. You should have a plot of $log_{10}(dt)$ versus time on $[0, 10]$. In addition print a summary containing the number of steps accepted and the number of steps rejected.

In debugging you code, you might want to use the fact that if q starts out being zero, it stays zero. Hence this code can model a simple diffusion problem, for which you already have a code.

Example 2, A Wave Equation System

In this example we use a pair of equations to model pressure and velocity pulses in a rigid pipe. Use the same spatial domain and mesh as in the previous example. The PDEs are

$$\partial_t v + \partial_x p = 0, \tag{5}$$

$$\partial_t p + \partial_x v = 0. \tag{6}$$

The boundary conditions are $v(0,t) = v(10,t) = 0$ for $t > 0$. The initial conditions are $v(x,0) = 0$ for $x \in [0, 10]$, $p(x,0) = 2(1-x^2)^3$ for $x \in [0, 1]$, and $p(x,0) = 0$ otherwise.

The solution will be represented using a $2\times(N+1)$ numpy array, `U`, as the state of the system, where the first row in `U` consists of the approximate values of $v(x,t$ at the points `x[i]` at time t, and the second row consists of the approximate values of $p(x,t$ at points `(x[i]+x[i+1])/2`, for $i = 0, 1, 2, \ldots, N-1$.

The last value of the second row is just set to zero. (This differs a little form my suggestion in class.) We'll use the identification

```
V = U[0]
P = U[1].
```

The Residual `R` will have a top row `RV` and a 2nd row `RP`. The function `Resid` will start with

```
W = U + dU
WV = W[0]
WP = W[1]
dV = dU[0]
dP = dU[1]
```

The first and last component of `RV` will just be `WV[0] WV[N]`. For $0 < i < N$, `RV[i]` should be

```
(1/dt) dV[i] +(1/dx) (WP[i]-WP[i-1]).
```

For $0 \leq i < N$ the i-th component of `RP` is

```
(1/dt) dP[i] +(1/dx) (WV[i+1]-WV[i]).
```

The last component of `RP` is just `WP[N]`.

For this example, I suggest `dtmin = dtmax = 0.05`

Show plots of $v(x,t)$ and $p(x,t)$ at $t = 0, 5,$ and 13.

Full disclosure: This project, and the previous two, are based on a system that was developed by Bruce Ayati called BuGS.

Last modified 5/23/22 – tfd