Solving tridiagonal matrix equations

In the fully implicit and semi-implicit solvers for the diffusion equation, the goal is to solve a tridiagonal matrix equation of the form

$$\begin{pmatrix} b_0 & c_0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ a_1 & b_1 & c_1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & a_2 & b_2 & c_2 & 0 & \cdots & 0 & 0 & 0 \\ . & . & . & . & . & & . & . & . \\ . & . & . & . & . & & . & . & . \\ . & . & . & . & . & & . & . & . \\ 0 & 0 & 0 & 0 & 0 & \cdots & a_{J-2} & b_{J-2} & c_{J-2} \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & a_{J-1} & b_{J-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ . \\ . \\ . \\ x_{J-2} \\ x_{J-1} \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ . \\ . \\ . \\ r_{J-2} \\ r_{J-1} \end{pmatrix},$$

where the solution vector $\mathbf{x}$ is the new state of the system, $x_j = u_j^{n+1}$ and the vector $\mathbf{r}$ is constructed from the u_j^n. As already discussed, the top (b_0, c_0, and r_0) and bottom (a_{J-1}, b_{J-1}, and r_{J-1}) rows of the matrix are used to apply the boundary conditions of the diffusion problem.

There are several ways to solve this matrix equation. The first is to use brute force and apply the `solve` function in `scipy.linalg`:

```
from scipy.linalg import solve
x = solve(A, r)
```

This will work, and may even run quite quickly given that most of the elements of $\mathbf{A}$ are zero (and the solver likely works by reducing A to triangular form and solving by back-substitution). However, the storage of the matrix $\mathbf{A}$ is very wasteful of space (it has J^2 entries, but only $3J-2$ of them are nonzero) and can easily consume a lot of memory if J is large.

An alternative is to convert to triangular form and solve by back-substitution "by hand," as implemented in the script `tridiag.py` (available on the course web page), adapted from the Numerical Recipes function of the same name:

```
x = tridiag(a, b, c, r)
```

A tridiagonal system is so close to triangular that the operations are simple to code and require only $O(J)$ memory and computational cost.

If you prefer to use Python built-ins, you can use the `scipy` function `solve_banded`, which is designed to solve "banded" matrices having some number of contiguous non-zero off-diagonal elements:

```
from scipy.linalg import solve_banded
x = solve_banded((1,1), Ab, r)
```

Here, the leading $(1,1)$ specifies the number of nonzero off-diagonal vectors below and above the diagonal and the $3 \times J$ matrix $\mathbf{Ab}$ stores the nonzero data:

$$\mathbf{Ab} = \begin{pmatrix} - & c_0 & c_1 & c_2 & \cdots & c_{J-3} & c_{J-2} \\ b_0 & b_1 & b_2 & b_3 & \cdots & b_{J-2} & b_{J-1} \\ a_1 & a_2 & a_3 & a_4 & \cdots & a_{J-1} & - \end{pmatrix}.$$

The dashes indicate elements that are not used by the solver, and can safely be set to zero. We can easily create this array in `numpy` as follows:

```
Ab = np.zeros((3,J))
Ab[0,1:] = c[:-1]
Ab[1,:] = b
Ab[2,:-1] = a[1:]
```