Time-dependent PDEs & PETSc TS

yes, we can do time-stepping

Ed Bueler
Dept. of Mathematics and Statistics, UAF

26 April 2016

outline for today

Chapter 6 of book:

- ODE initial value problem examples and methods
- PETSc `TS` object can do time-stepping
  - explicit Runge-Kutta methods
  - implicit methods (backward Euler, θ, IMEX)
    - require solving equations at each time step: `SNES`
- `c/ch6/heat.c` solves classical heat equation

$$u_t = D\nabla^2 u + f(x, y)$$

  - diffusions are stiff, so implicit appropriate
- `c/ch6/pattern.c` solves system of coupled nonlinear diffusions for pattern formation:

$$u_t = D_u \nabla^2 u - uv^2 + F(1 - u)$$
$$v_t = D_v \nabla^2 v + uv^2 - (F + k)v$$

systems of ODEs

- we can solve ODE systems of form

$$\mathbf{y}' = \mathbf{g}(t, \mathbf{y})$$

  - $\mathbf{y}(t) \in \mathbb{R}^N$
  - ODE describes motion of point in N dimensions
  - $\mathbf{g}$ can be linear or nonlinear in t or $\mathbf{y}$
- *initial value problem* specifies solution at one time:

$$\mathbf{y}(t_0) = \mathbf{y}_0$$

  - exists unique solution if $\mathbf{g}(t, \mathbf{y})$ is continuous in both inputs and Lipschitz in $\mathbf{y}$ ... that is, you can *predict*
  - for $N < \infty$ cases (i.e. *not* PDEs), one can generally go forward or backward from $t = t_0$

example with fixed size $N = 2$

- example with initial time $t_0 = 0$

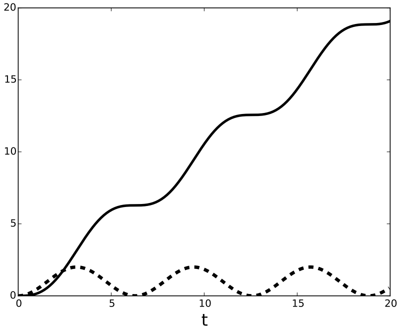
$$\mathbf{y}' = \begin{bmatrix} y_1 \\ -y_0 + t \end{bmatrix}, \quad \mathbf{y}(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- exact solution is

$$\mathbf{y}(t) = \begin{bmatrix} t - \sin t \\ 1 - \cos t \end{bmatrix}$$

- it is a linear system
$\mathbf{g}(t, \mathbf{y}) = A\mathbf{y} + \mathbf{f}(t)$ where

$$A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}, \quad \mathbf{f}(t) = \begin{bmatrix} 0 \\ t \end{bmatrix}$$



Euler numerical methods: forward and backward

- $t_0 < t_1 < t_2 < \cdots < t_L$ a sequence of times
- steps $h = t_\ell - t_{\ell-1}$; assume equal only for convenience
- $\mathbf{Y}_\ell \approx \mathbf{y}(t_\ell)$, and $\mathbf{Y}_0 = \mathbf{y}_0$
- explicit forward Euler method

$$\frac{\mathbf{Y}_\ell - \mathbf{Y}_{\ell-1}}{h} = \mathbf{g}(t_{\ell-1}, \mathbf{Y}_{\ell-1})$$

  - one can also write as update formula

$$\mathbf{Y}_\ell = \mathbf{Y}_{\ell-1} + h\,\mathbf{g}(t_{\ell-1}, \mathbf{Y}_{\ell-1})$$

- implicit backward Euler method is just as easy to state

$$\frac{\mathbf{Y}_\ell - \mathbf{Y}_{\ell-1}}{h} = \mathbf{g}(t_\ell, \mathbf{Y}_\ell)$$

  - *but* requires solving system of eqns at each time step

Runge-Kutta methods

- Runge-Kutta methods with s stages have form

$$\hat{\mathbf{Y}}_i = \mathbf{Y}_{\ell-1} + h \sum_{j=1}^{s} a_{ij}\, \mathbf{g}(t_{\ell-1} + c_j h, \hat{\mathbf{Y}}_j), \qquad 1 \leq i \leq s$$

$$\mathbf{Y}_\ell = \mathbf{Y}_{\ell-1} + h \sum_{i=1}^{s} b_i\, \mathbf{g}(t_{\ell-1} + c_i h, \hat{\mathbf{Y}}_i)$$

- some “tableau” for explicit instances:

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}
\qquad
\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}
\qquad
\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{3}{4} & 0 & \frac{3}{4} & & \\ 1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \\ \hline & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\ & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8} \end{array}$$

Table 6.1: Tableau for the explicit trapezoidal rule (RK2a; left), the classical fourth-order Runge-Kutta method (RK4; middle), and an embedded four-stage, third-order scheme (RK3bs, the PETSc default; right).

PETSc TS basic setup

```
TSCreate(COMM,&ts);
TSSetProblemType(ts,TS_NONLINEAR);
TSSetRHSFunction(ts,NULL,FormRHSFunction,NULL);
TSSetType(ts,TSRK);
TSSetInitialTimeStep(ts,t0,dt);
TSSetDuration(ts,maxsteps,tf-t0);
TSSetFromOptions(ts);
TSSolve(ts,y);
```

- setup similar to `SNES`, `KSP`, etc.
- setting `TS_NONLINEAR` says ODE is in form $\mathbf{y}' = \mathbf{g}(t, \mathbf{y})$
- calling `TSSetRHSFunction()` sets a call-back to our code `FormRHSFunction()`, which evaluates $\mathbf{g}(t, \mathbf{y})$
- `TSSetFromOptions()` allows overrides:
  - the `TSSetType()` choice by `-ts_type`
  - values of `t0,dt,maxsteps,tf` by ...
- `y` must be an allocated `Vec` of size N containing initial value $\mathbf{Y}_0$

first PETSc TS example

- recall our example:

$$\mathbf{g}(t,\mathbf{y}) = \begin{bmatrix} y_1 \\ -y_0 + t \end{bmatrix}$$

- thus implementation of $\mathbf{g}(t,\mathbf{y})$:

```
PetscErrorCode FormRHSFunction(TS ts, double t, Vec y,
                               Vec g, void *ptr) {
    const double *ay;
    double       *ag;
    VecGetArrayRead(y,&ay);
    VecGetArray(g,&ag);
    ag[0] = ay[1];
    ag[1] = - ay[0] + t;
    VecRestoreArrayRead(y,&ay);
    VecRestoreArray(g,&ag);
    return 0;
}
```

running `ode.c`

- `c/ch6/ode.c` is essentially just the stuff on the last two slides . . . simple!
- basic run with adaptive, explicit RK $O(h^3)$ method (default):

```
$ make
$ ./ode -ts_monitor
0 TS dt 0.1 time 0.
1 TS dt 0.170141 time 0.1
...
87 TS dt 0.11548 time 19.8845
88 TS dt 0.205616 time 20.
error at tf = 20.000 :  |y-y_exact|_inf = 0.00930352
```

variations:

- changing solver and time axis:

```
$ ./ode -ts_type euler -ts_final_time 1.0 -ts_dt 0.5
```

- generating run-time movie of trajectory:

```
$ ./ode -ts_monitor_lg_solution -draw_pause 0.1
```

implicit solvers need `SNES` and Jacobian

- backward Euler:

```
$ ./ode -ts_type beuler                  # error
$ ./ode -ts_type beuler -snes_fd         # works
```

- `odejac.c` adds Jacobian function to `ode.c`:

```
$ make odejac
$ ./odejac -ts_type beuler               # works
```

- larger class of implicit methods are "θ"-methods with $0 < \theta \le 1$:

$$\frac{\mathbf{Y}_\ell - \mathbf{Y}_{\ell-1}}{h} = (1-\theta)\,\mathbf{g}(t_{\ell-1}, \mathbf{Y}_{\ell-1}) + \theta\,\mathbf{g}(t_\ell, \mathbf{Y}_\ell)$$

  - $\theta = 1/2$ is $O(h^2)$ accurate: "trapezoid" or "Crank-Nicolson"

```
$ ./odejac -ts_type theta -ts_theta_theta 0.4
$ ./odejac -ts_type cn
```

"stiff" ODE systems

- consider ODE systems of the form $\mathbf{y}' = \mathbf{g}(\mathbf{y})$, i.e. "autonomous" without t-dependence
- system is *asymptotically stable* if eigenvalues λ of Jacobian $J = \frac{\partial g_i}{\partial y_j}$ have negative real part
- such a system is *stiff* if

$$\min(\mathrm{Re}\,\lambda) \ll \max(\mathrm{Re}\,\lambda) < 0$$

  - or: $\kappa(J) \gg 1$ in symmetric case
- idea:

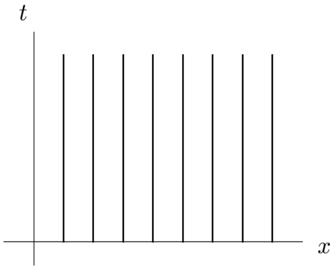
  stiff implies the "$\min(\mathrm{Re}\,\lambda)$*" eigenvalues will control time steps of explicit ODE method, causing very short time steps, even though the "*$\max(\mathrm{Re}\,\lambda)$*" eigenvalues control the macroscopic behavior of solution*

“method of lines” (MOL) for PDEs

- one might recognize time-dependent PDEs as “ODEs in ∞ dimensions”; you can discretize with this in mind
  - this is how to solve time-dependent PDEs using PETSc
- consider time-dependent PDEs of form

$$u_t = G(t, u, \nabla u, \nabla^2 u)$$

- *idea*: spatial discretization only gives system of ODEs



MOL for heat equation

- *example*: if $\{x_i\}$ is an N-point spatial grid then

$$u_t = u_{xx} \quad \to \quad U_j'(t) = \frac{U_{j-1}(t) - 2U_j(t) + U_{j+1}(t)}{\Delta x^2}$$

gives system in form "$\mathbf{y}' = \mathbf{g}(t, \mathbf{y})$" in $\mathbb{R}^N$, where $U_j(t) \approx u(t, x_j)$ and $\mathbf{U}(t) = \{U_j(t)\} \in \mathbb{R}^N$

- symmetric matrix for RHS is

$$A = \frac{1}{\Delta x^2} \begin{pmatrix} -2 & 1 & 0 & & 0 \\ 1 & -2 & 1 & & 0 \\ 0 & 1 & -2 & & 0 \\ & & & \ddots & \\ 0 & 0 & 0 & & -2 \end{pmatrix}$$

has negative real eigenvalues, some ≈ 0 and some $\lambda \to -\infty$

- this is generic:

MOL on diffusion equations gives stiff ODE systems

`TS` example codes in `c/ch6/`

- using form $\mathbf{y}' = \mathbf{g}(t, \mathbf{y})$:

`ode.c` solves above example
`odejac.c` adds Jacobian so that implicit methods can be used
`heat.c` MOL for classical heat equation

$$u_t = D\nabla^2 u + f$$

- using "implicit-explicit" form $\mathbf{f}(t, \mathbf{y}, \mathbf{y}') = \mathbf{g}(t, \mathbf{y})$:

`pattern.c` MOL for system of two coupled nonlinear heat-like PDEs

$$\begin{aligned} u_t - D_u \nabla^2 u &= -uv^2 + F(1-u) \\ v_t - D_v \nabla^2 v &= +uv^2 - (F+k)v \end{aligned}$$

  - a model for two chemical species with reaction, diffusion, and feed (Pearson, 1993)
  - "adaptive Runge-Kutta implicit-explicit" appropriate (`-ts_type arkimex`)

running `heat.c`

- `c/ch6/heat.c` solves $u_t = D\nabla^2 u + f(x,y)$
  - on unit square $\Omega = (0,1) \times (0,1)$
  - non-homogeneous Neumann boundary conditions in x
  - periodic boundary conditions in y
  - energy is conserved
- high-res Crank-Nicolson run on 4 processes with saving of time axis and solution:

```
mpiexec -n 4 ./heat -ts_type cn -da_refine 7 \
    -ts_dt 0.0001 -ts_final_time 0.05 \
    -ts_monitor binary:theat.dat \
    -ts_monitor_solution binary:uheat.dat
```

- output can be transformed into an off-line movie:

```
./plotTS.py -mx 385 -my 384 theat.dat uheat.dat \
    -oroot heat
ffmpeg -r 8 -i heat%03d.png heat.m4v
```

running `pattern.c`

- `c/ch6/pattern.c` solves

$$\begin{aligned} u_t - D_u \nabla^2 u &= -uv^2 + F(1-u) \\ v_t - D_v \nabla^2 v &= +uv^2 - (F+k)v \end{aligned}$$

  - periodic boundary conditions in both directions
  - creates 2D `DMDA` with `dof=2`
- problem in form $\mathbf{f}(t, \mathbf{y}, \mathbf{y}') = \mathbf{g}(t, \mathbf{y})$:
  - `-ts_type arkimex` is default
  - `FormIFunctionLocal()` for $\mathbf{f}$; with Jacobian
  - `FormRHSFunctionLocal()` for $\mathbf{g}$; no Jacobian
- high-res run on 4 procs, and off-line movie of u component:

```
mpiexec -n 4 ./pattern -ts_adapt_type none \
    -ts_final_time 5000 -ts_dt 10 -da_refine 8 \
    -ts_monitor binary:t8.dat \
    -ts_monitor_solution binary:uv8.dat
./plotTS.py -mx 768 -my 768 -dof 2 -c 0 \
    t8.dat uv8.dat -oroot uhigh8
ffmpeg -r 8 -i uhigh8%03d.png uhigh8.m4v
```