Nonlinear PDEs and Jacobians

1D example solved by 21st century means

Ed Bueler
Dept. of Mathematics and Statistics, UAF

29 March 2016

outline for today

Chapter 4 of book:

- recall 1D diffusion-reaction example `c/ch4/reaction.c`:

$$-u'' - R(u) = f(x)$$

- show:
  - evidence for convergence
  - finite-difference Jacobian
    - optionally by graph coloring
  - matrix-free Newton-Krylov
    - optionally with a preconditioning matrix

recall: problem and solution method

- balance of diffusion/reaction/source processes:

$$-u'' - R(u) = f(x), \quad u(0) = \alpha, \quad u(1) = \beta$$

  - an ODE 2-point BVP, but acts like elliptic PDE (e.g. Poisson)
  - $R(u)$ is any nonlinear function
  - exact solution known when $R(u) = -\rho\sqrt{u}$ and $f(x) = 0$

- discretize on N-point structured grid by finite differences:

$$-\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} - R(u_i) = f(x_i)$$

- becomes "residual=0" equations in $\mathbb{R}^N$: $\mathbf{F}(\mathbf{x}) = 0$
- Newton's method: choose $\mathbf{x}_0$ and iterate

$$J_{\mathbf{F}}(\mathbf{x}_k)\,\mathbf{s} \stackrel{*}{=} -\mathbf{F}(\mathbf{x}_k), \qquad \mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}$$

  - solve each linear system $*$ by preconditioned Krylov: "Newton-Krylov"

recall: PETSc implementation

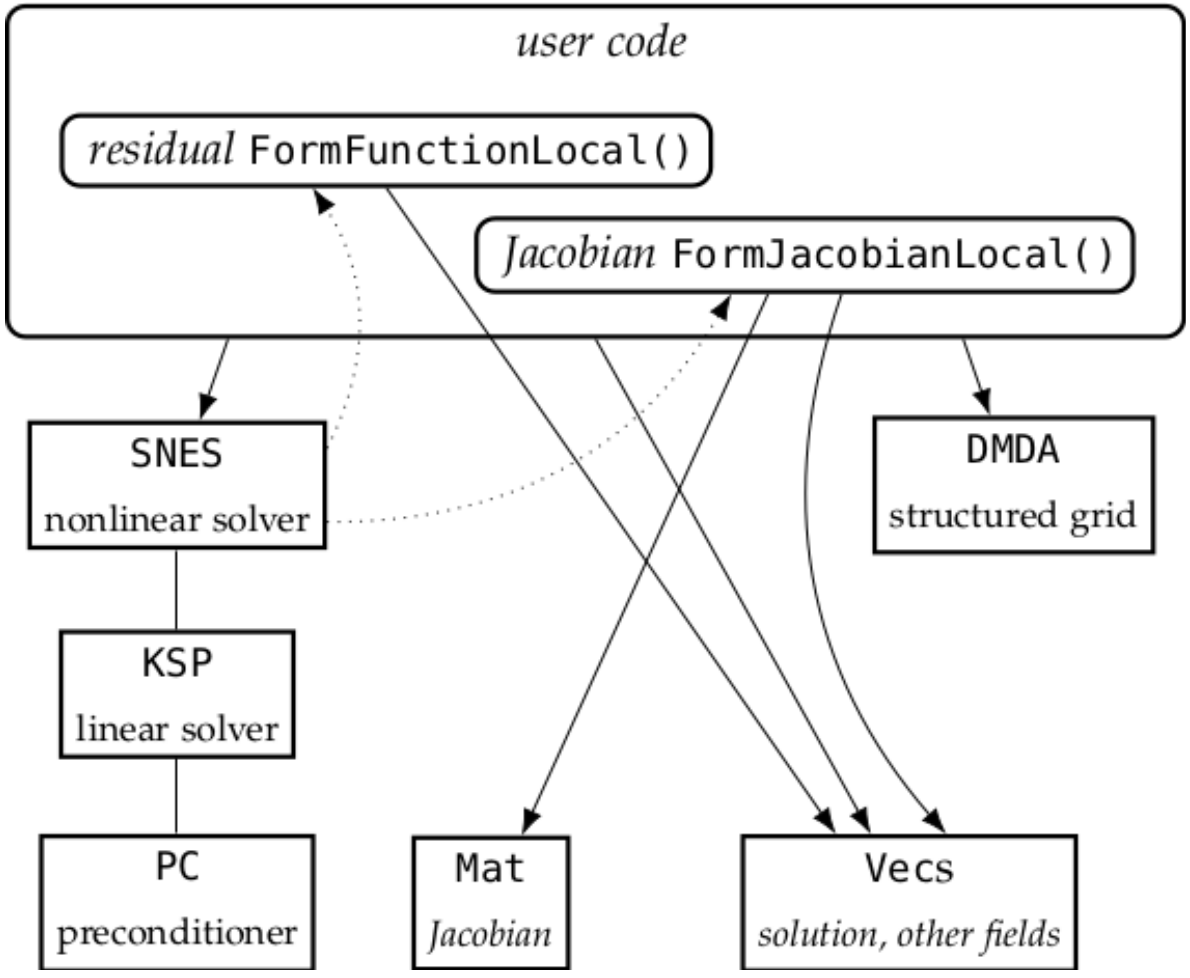
- `c/ch4/reaction.c`

- `DMDA` manages grid (in parallel)
- `SNES` manages Newton's method (in parallel)
  - has Krylov solver `KSP` and preconditioner `PC` inside

- we write these parts of code:
  - initial iterate function
  - residual function $\mathbf{F}(\mathbf{x})$, a `SNES` call-back
  - Jacobian $J_{\mathbf{F}}(\mathbf{x})$, a `SNES` call-back
  - `main()`:
    - call `Create/SetFromOptions` on objects
    - call initial iterate function
    - call `SNESSolve()`
    - measure error $\|\mathbf{x}_k - \mathbf{x}\|$ (relative to exact solution $\mathbf{x}$)
    - call `Destroy` on objects

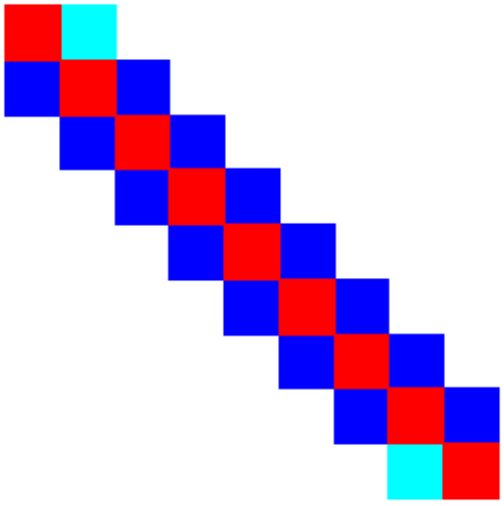
recall: structure of `c/ch4/reaction.c`

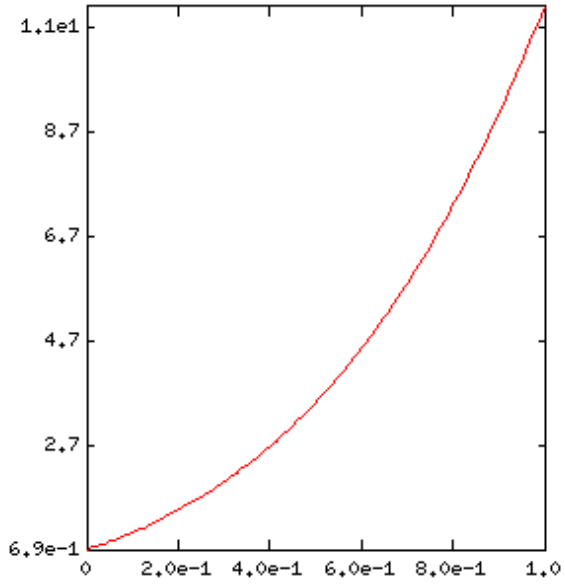
- solid arrows mean “user code acts directly on”
- dotted arrows are call-backs



basic runs

```
$ ./reaction -da_refine 4                                  # refine by 2^4
$ ./reaction -snes_monitor -ksp_converged_reason           # info on run
$ ./reaction -snes_view                                    # expose solvers
$ ./reaction -mat_view draw -draw_pause 1                  # show matrix (L)
$ ./reaction -da_refine 4 -snes_monitor \                  # show iterates (R)
   -snes_monitor_solution draw -draw_pause -1
```

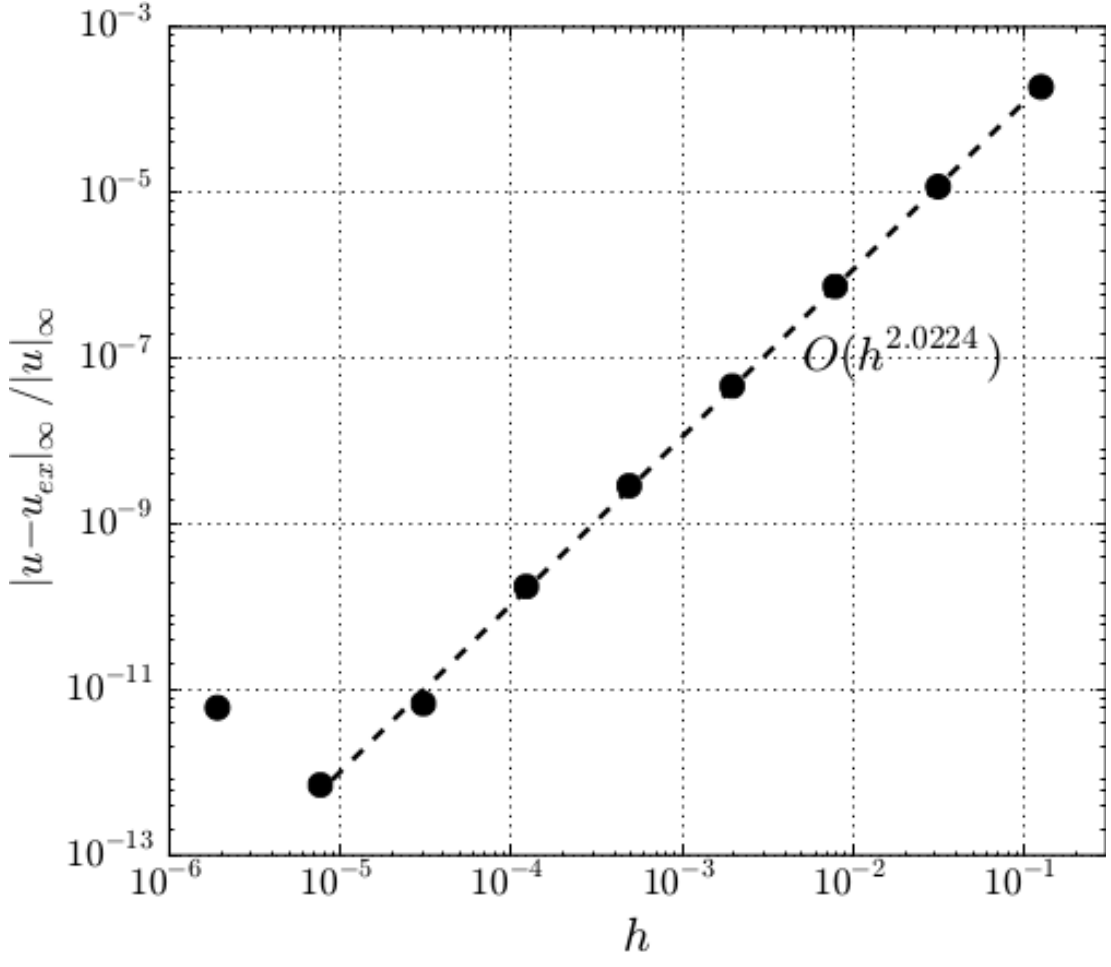





```
$ mpiexec -n 4 ./reaction -da_refine 12 \                  # parallel
    -snes_monitor -ksp_converged_reason
```

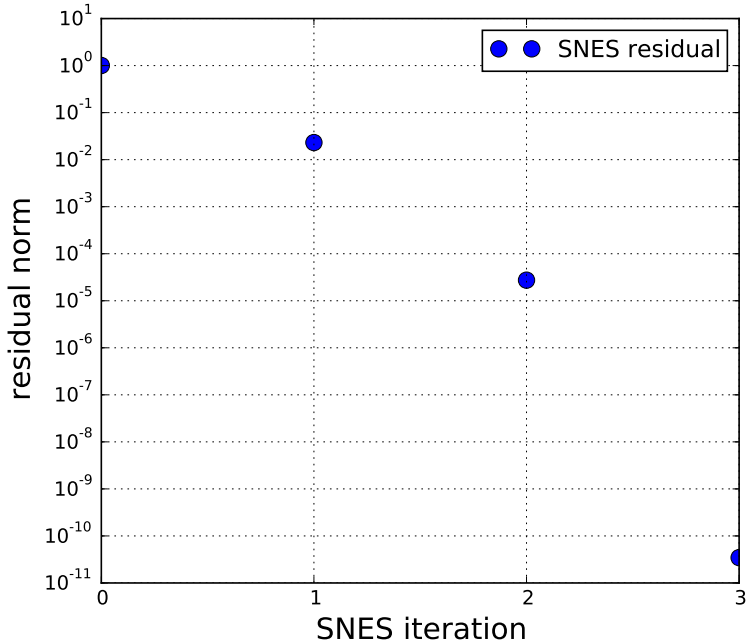
convergence

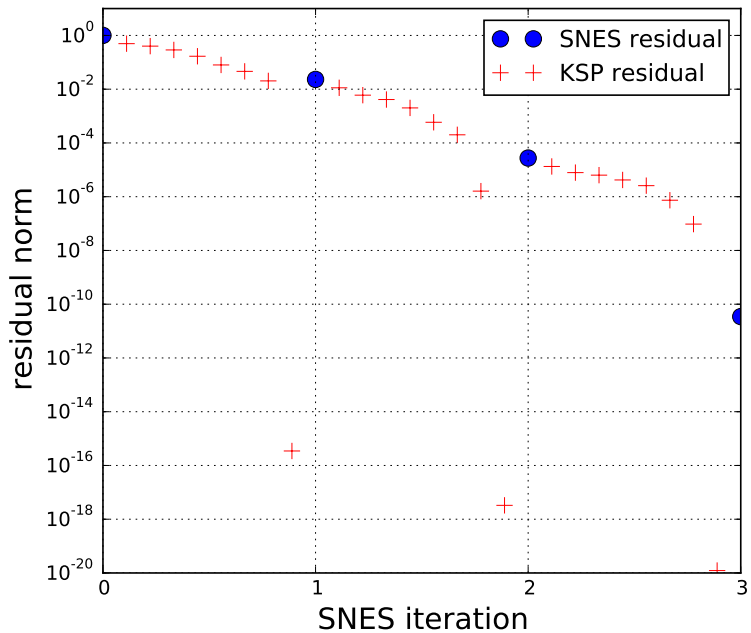
```
$ for N in 0 2 4 6 8 10 12 14 16; do
>   ./reaction -da_refine $N -snes_rtol 1.0e-10; done
```



distinguish meanings of “convergence”!

- previous: convergence of numerical soln to exact soln
  - usually the main goal
- left: convergence of residual in the nonlinear equations
  - Newton “outer” iteration (SNES residual)
- right: convergence of residual in linear eqns per Newton step
  - Krylov “inner” iteration (KSP residual)






```
$ ./reaction -snes_monitor -ksp_type preonly -pc_type lu
$ ... -ksp_monitor -ksp_type gmres -pc_type jacobi
```

approximate Jacobians

- in our code we do have `FormJacobianLocal()`
  - . . . but suppose we did not . . .

    implementing Jacobians can be an error-prone pain
  - of course, we *must* have a `FormFunctionLocal()` which computes $\mathbf{F}(\mathbf{x})$
- one idea for approximate Jacobian is `-snes_fd`:

$$J_{ij} = \frac{\partial F_i}{\partial x_j} \approx \frac{F_i(\mathbf{x} + \delta \mathbf{e}_j) - F_i(\mathbf{x})}{\delta}$$

  - $\delta = \sqrt{\epsilon}$ where $\epsilon = 2.2 \times 10^{-16}$ is machine precision
- different idea, suitable for Krylov iteration, is `-snes_mf`:

$$J\mathbf{v} \approx \frac{\mathbf{F}(\mathbf{x} + \delta \mathbf{v}) - \mathbf{F}(\mathbf{x})}{\delta}$$

approximate Jacobians: the caveat

- neither `-snes_fd` nor `-snes_mf` scale well!
- for different reasons:
  - `-snes_fd` does not scale well because if grid has N points then $N + 1$ residual evaluations are needed per Newton step
    - need to avoid redundant residual evaluations
  - `-snes_mf` does not scale well because only $J^k\mathbf{v}$ can be calculated, so Krylov method will not converge fast if spectrum of J is spread out
    - need to add a preconditioner
- fortunately, modified versions `-snes_fd_color` and `-snes_mf_operator` exist

finite-difference Jacobians

- options `-snes_fd` and `-snes_fd_color` only need residual implementation (`FormFunctionLocal()`)
- recall we can see how many times the residual function `FormFunctionLocal()` is evaluated:

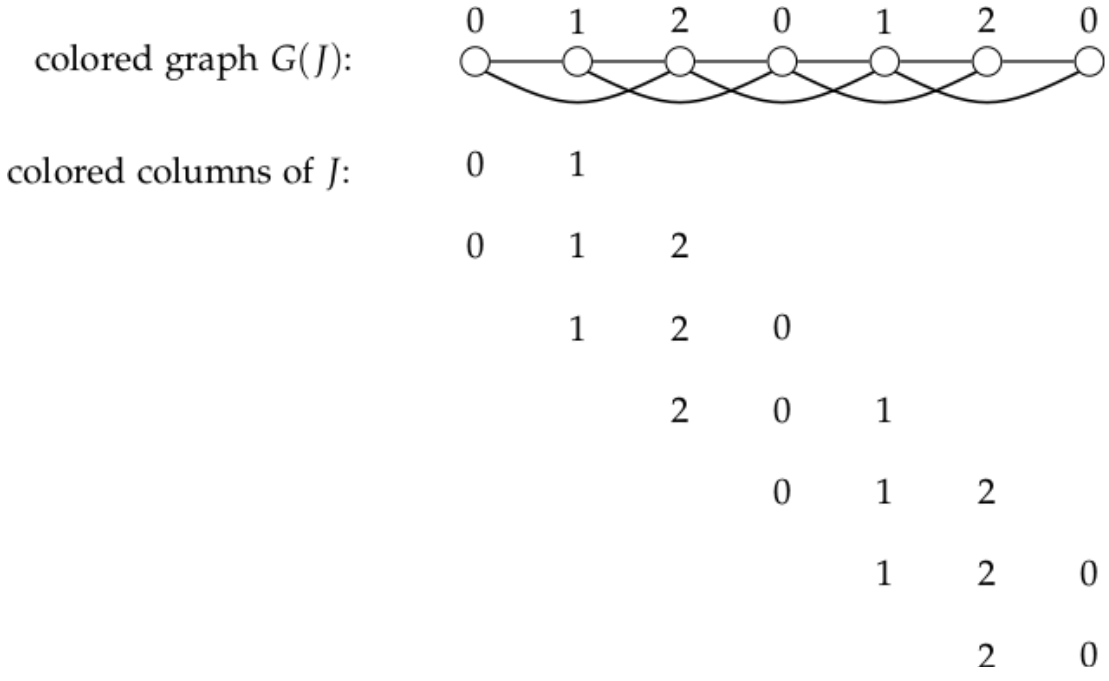
```
$ ... -log_view |grep SNESFunctionEval
```

- for example:

```
$ ./reaction -da_refine 10
$ ./reaction -da_refine 10 -snes_fd
$ ./reaction -da_refine 10 -snes_fd_color
```

gives 4, 16390, 13 evaluations, respectively

how `-snes_fd_color` works



- graph has edge if two nodes are connected in stencil
- J is built by evaluating **F** once for all columns of one color
  - graph needs $c = 3$ colors in this case
- number of evals of **F** is $q(c+1)+1$ for q Newton steps
  - independent of refinement level

matrix-free Newton-Krylov ("JFNK"): issues

- option `-snes_mf` only needs residual implementation (`FormFunctionLocal()`)
- . . . but it tends not to work for fine grids
- to illustrate the scaling problem
  - does not converge:

    ```
    $ ./reaction -da_refine 10 -snes_mf
    ```
  - gives 3225 evaluations:

    ```
    $ ./reaction -da_refine 4 -snes_mf
    ```
    - because of $\approx$ 1000 GMRES iterations per Newton step
  - for comparison,

    ```
    $ ./reaction -da_refine 4
    $ ./reaction -da_refine 4 -snes_fd
    $ ./reaction -da_refine 4 -snes_fd_color
    ```

    gives 4, 394, 13 evaluations, respectively

matrix-free Newton-Krylov ("JFNK"): improvement

- option `-snes_mf_operator` needs Jacobian code
  - yes, you need to write `FormJacobianLocal()`
  - *but* it can be only an approximate Jacobian "P"
  - $P \approx J$ will be used for preconditioning
- for example, P could be the Jacobian corresponding to the Poisson equation

$$-w'' = f(x)$$

instead of the actual problem

$$-u'' + R(u) = f(x)$$

trying out matrix-free Newton-Krylov

- for example, code `c/ch4/reaction.c` has option `-rct_noRinJ` which removes "$R(u)$" part of (correct) Jacobian

- with analytical Jacobian

```
$ ./reaction -snes_monitor -da_refine 10
```

- with inexact Jacobian

```
$ ./reaction -snes_monitor -da_refine 10 \
    -rct_noRinJ
```

- with inexact Jacobian *used as preconditioner*

```
$ ./reaction -snes_monitor -da_refine 10 \
    -rct_noRinJ -snes_mf_operator
```

Jacobian options comparisons

	no option	`-snes_fd` `-snes_fd_color`	`-snes_mf`	`-snes_mf_operator`
only **F**	error	✓	✓	error
F and P	✓	✓	✓	✓
F and J	✓	✓	✓	✓

Table 4.2: Jacobian options when using `SNES`. Rows give which mathematical objects have been provided through user code. Symbol "P" denotes an approximate Jacobian while "J" denotes the actual Jacobian. A large check mark shows recommended usage.

	no option	`-snes_fd` `-snes_fd_color`	`-snes_mf` `-snes_mf_operator`
residual **F**	$q+1$	$q(N+1)+1$ $q(c+1)+1$	qm_1 qm_2
Jacobian J or P	q	0 0	0 q

Table 4.3: Jacobian options compared by number of residual and Jacobian evaluations. Here q is the number of Newton iterations, N is the dimension of the problem, c is the number of colors on $G(J)$, m_1 is the dimension of the Krylov space for J, and m_2 is the Krylov space dimension for the preconditioned operator $M^{-1}J$.

- main idea:

  you have options, after implementing the residual $\mathbf{F}(\mathbf{x})$*, for solving without an analytical Jacobian, or with a merely approximate Jacobian*