Understanding the Poisson solution

CG, preconditioning, and performance

Ed Bueler
Dept. of Mathematics and Statistics, UAF

23 February 2016

outline for today

finish Chapter 3 of book:

- recall Poisson code
- convergence
- CG = conjugate gradients
- preconditioning
- scaling flaw in preconditioned CG

next week:

- start Chapter 4 on Newton’s method

recall `c/ch3/poisson.c`

- `poisson.c` discretizes on a structured grid:

$$-\nabla^2 u = f \quad \to \quad A\mathbf{u} = \mathbf{b}$$

- it then hands the linear system $A\mathbf{u} = \mathbf{b}$ to PETSc:

```
formRHS(da,b);
formMatrix(da,A);
KSPCreate(PETSC_COMM_WORLD,&ksp);
KSPSetOperators(ksp,A,A);
KSPSetFromOptions(ksp);
KSPSolve(ksp,b,u);
```

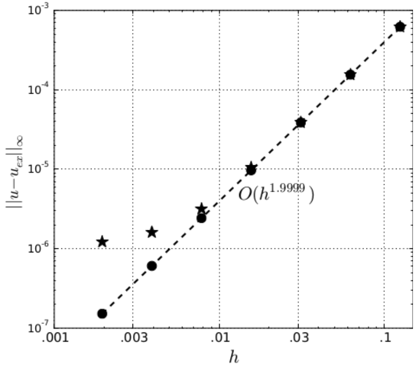
goals once you have an initial implementation

- convergence: is numerical method implemented correctly?
- exposure: what is occurring inside PETSc when we solve?
- efficiency: how to get high performance?

evidence for convergence

```
$ for K in 0 1 2 3 4 5 6; do ./poisson -da_refine $K; done
on 9 x 9 grid:  error |u-uexact|_inf = 0.000763959
on 17 x 17 grid:  error |u-uexact|_inf = 0.000196764
on 33 x 33 grid:  error |u-uexact|_inf = 4.91557e-05
on 65 x 65 grid:  error |u-uexact|_inf = 1.29719e-05
on 129 x 129 grid:  error |u-uexact|_inf = 3.76924e-06
on 257 x 257 grid:  error |u-uexact|_inf = 1.73086e-06
on 513 x 513 grid:  error |u-uexact|_inf = 1.23567e-06
```

- results above are $\star$ on graph
- add `-ksp_rtol 1.0e-12` to get ● results
- or `-ksp_type preonly -pc_type lu`
- FD method can give $O(h^2)$ at best, so `poisson.c` *is correct*



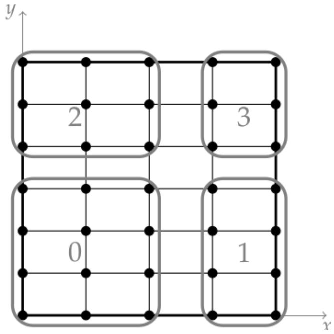
ask PETSc to show parallel structured grid

- use option `-dm_view`:

```
$ ./poisson -dm_view
$ mpiexec -n 2 ./poisson -dm_view
```

- with `draw`:

```
$ mpiexec -n 2 ./poisson -dm_view draw -draw_pause 2
$ mpiexec -n 4 ./poisson -da_grid_x 5 -da_grid_y 7 \
    -dm_view draw -draw_pause 2
```



matrix structure in `poisson.c`

- note line `MatSetOptionsPrefix(A,"a_")` in `poisson.c`
- use `-a_mat_view`:

```
$ ./poisson -da_grid_x 4 -da_grid_y 4 -a_mat_view
$ ./poisson -da_grid_x 4 -da_grid_y 4 \
    -a_mat_view ::ascii_dense
$ ./poisson -a_mat_view draw -draw_pause 5
```

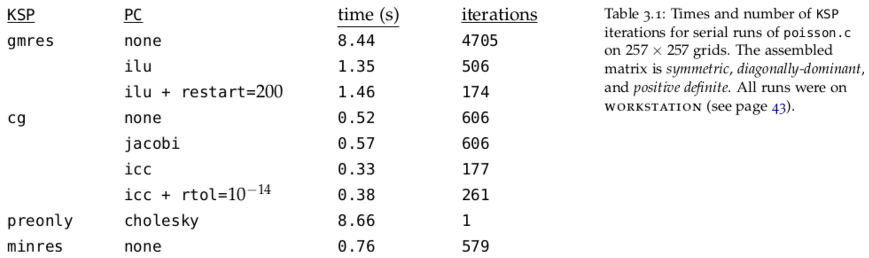
- use it to read matrix into MATLAB/OCTAVE:

```
$ ./poisson -da_grid_x 5 -da_grid_y 7 \
    -a_mat_view ascii:foo.m:ascii_matlab
$ octave
  >> foo
  >> whos
  >> spy(...)
```

try various `KSP` and `PC`

- defaults include `gmres,ilu`
- matrix is symmetric so `minres` also available
- matrix is positive-definite so `cg,icc,cholesky` also
- table below from

```
$ timer ./poisson -da_refine 5 \
    -ksp_converged_reason \
    -ksp_type KSP -pc_type PC
```



KSP	PC	time (s)	iterations
gmres	none	8.44	4705
	ilu	1.35	506
	ilu + restart=200	1.46	174
cg	none	0.52	606
	jacobi	0.57	606
	icc	0.33	177
	icc + rtol=10^{-14}	0.38	261
preonly	cholesky	8.66	1
minres	none	0.76	579

Table 3.1: Times and number of KSP iterations for serial runs of `poisson.c` on 257×257 grids. The assembled matrix is *symmetric*, *diagonally-dominant*, and *positive definite*. All runs were on WORKSTATION (see page 43).

CG = conjugate gradients

- `-ksp_type cg`
- well-known Krylov iteration for solving $A\mathbf{u} = \mathbf{b}$ $\rightarrow$
- requires A to be symmetric and positive-definite
  - sometimes converges even if not
  - reliable compared to Richardson!
- less expensive than GMRES
  - 4 vectors in memory vs many ($\sim$ 30)
  - note: both iterations do one product $A\mathbf{v}$ per iteration

```
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
repeat
    $\alpha_k := \frac{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}{\mathbf{p}_k^\mathsf{T}\mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$
    $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
    if $r_{k+1}$ is sufficiently small then exit
    $\beta_k := \frac{\mathbf{r}_{k+1}^\mathsf{T}\mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}$
    $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$
    $k := k + 1$
end repeat
The result is $\mathbf{x}_{k+1}$
```

what CG accomplishes

- let $\mathbf{u}_j$ be iterates from (un-preconditioned) CG
- errors $\mathbf{e}_j = \mathbf{u}_j - \mathbf{u}$ satisfy

$$\|\mathbf{e}_j\|_A = \min_{p \in \mathcal{P}_j^1} \|p(A)\mathbf{e}_0\|_A$$

  - where $\mathcal{P}_j^1 = \{$degree j polynomials $p(x)$ with $p(0) = 1\}$
  - $\|\mathbf{v}\|_A = \langle \mathbf{v}, A\mathbf{v}\rangle^{1/2}$ ... not the L^2 norm
  - thus $\|\mathbf{e}_j\|_A$ decrease monotonically
- the A-norm of the error is controlled by $\kappa = \kappa_2(A)$:

$$\frac{\|\mathbf{e}_j\|_A}{\|\mathbf{e}_0\|_A} \le 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^j$$

preconditioning CG

- to improve CG performance, preconditioning the system

$$A\mathbf{u} = \mathbf{b} \qquad \to \qquad M^{-1}A\mathbf{u} = M^{-1}\mathbf{b}$$

- last slide says: preconditioning needs to reduce the condition number
- we need M symmetric and positive-definite, and factored:

$$M = EE^\top$$

- goal:

$$\kappa_2(A) \gg \kappa_2(E^{-1}AE^{-\top})$$

- "incomplete cholesky" (`-pc_type icc`) computes E which is a sparse approximate factor of A:

$$EE^\top \approx A$$

- `-pc_type cholesky` does it perfectly: $EE^\top = A$

measuring performance of `poisson.c`

- timing should use `-with-debugging=0` PETSc configuration
- and we should use a fine grid so that timing is less dominated by other system activities

```
$ export PETSC_ARCH=linux-c-opt
$ make poisson
...
$ timer ./poisson -da_refine 6
on 513 x 513 grid:  error |u-uexact|_inf = 1.23567e-06
real 17.80
$ timer ./poisson -da_refine 6 -ksp_type cg -pc_type icc
on 513 x 513 grid:  error |u-uexact|_inf = 1.96296e-07
real 2.87
```

preconditioned CG: actually measure what works!

```
$ for PC in none jacobi icc cholesky; do \
    timer ./poisson -da_refine 6 -ksp_type cg -pc_type $PC \
    -ksp_converged_reason -ksp_compute_singularvalues; done
Linear solve converged due to CONVERGED_RTOL iterations 1227
Iteratively computed extreme singular values:
     max 7.99992 min 7.52989e-05 max/min 106242.
on 513 x 513 grid:  error |u-uexact|_inf = 1.94253e-07
real 4.91
Linear solve converged due to CONVERGED_RTOL iterations 1227
Iteratively computed extreme singular values:
     max 1.99998 min 1.88247e-05 max/min 106242.
on 513 x 513 grid:  error |u-uexact|_inf = 1.94253e-07
real 5.20
Linear solve converged due to CONVERGED_RTOL iterations 357
Iteratively computed extreme singular values:
     max 1.20706 min 0.000128532 max/min 9391.18
on 513 x 513 grid:  error |u-uexact|_inf = 1.96296e-07
real 2.83
Linear solve converged due to CONVERGED_RTOL iterations 1
Iteratively computed extreme singular values:
     max 1. min 1. max/min 1.
on 513 x 513 grid:  error |u-uexact|_inf = 1.92073e-07
real 112.32
```

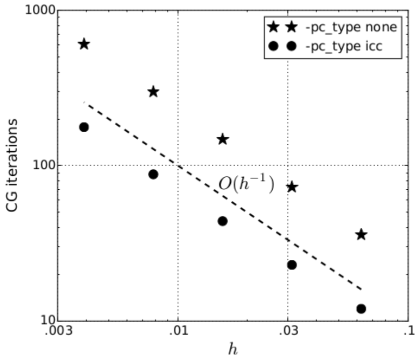
preconditioned CG: the hope

- preconditioned CG was the great hope of about 1970
- *good*: preconditioning by `icc` *does* reduce iteration count by reducing condition number, relative to no preconditioning or `jacobi`
- *bad*: preconditioning by `icc` *does not* slow down the growth condition number $\kappa_2(A)$ as the grid is refined

preconditioned CG: the scaling flaw

- do with `PC=none,icc`:

```
$ for K in 0 1 2 3 4; do \
  ./poisson -da_refine $K -ksp_converged_reason
  -ksp_type cg -pc_type PC; done
```



we need genuinely-better preconditioning!

Code	KSP	PC	time (s)	iterations
poisson	cg	icc	27.10	1056
	preonly	cholesky + nd	24.96	1
		lu + nd	16.33	1
fish2	cg	mg	2.60	5

Table 3.2: Time and iteration count for `-da_refine 7` runs, a 1025×1025 grid on a single MPI process on WORKSTATION. The `fish2` code appears in Chapter 8.