Introduction to Nonlinear Solvers Using PETSc/TAO

Presented to
ATPESC 2021 Participants

Richard Tran Mills
Argonne National Laboratory
August 10, 2021

PETSc TAO

U.S. DEPARTMENT OF ENERGY | Office of Science

ATPESC Numerical Software Track

ECP EXASCALE COMPUTING PROJECT

Argonne NATIONAL LABORATORY · BERKELEY LAB · Sandia National Laboratories · Rensselaer · SMU

Iterative Solvers for Nonlinear Systems

Systems of nonlinear equations

$$F(x) = b \quad \text{where} \quad F : \mathbb{R}^N \to \mathbb{R}^N \tag{1}$$

arise in countless settings in computational science.
Direct methods for general nonlinear systems do not exist. Iterative methods are required!

Nonlinear Richardson (simple) iteration:

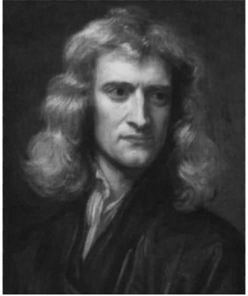
$$x_{k+1} = x_k + \lambda(b - F(x_k)) \tag{2}$$

This has linear convergence at best: $\|e_{k+1}\| \leq C\|e_k\|$

Nonlinear Krylov methods

Nonlinear CG - Mimic CG to force each new search direction to be orthogonal to previous directions.
Nonlinear GMRES (Anderson mixing) - minimize $\|F(x_{k+1} - b)\|$ by using x_{k+1} as a linear combination of previous solutions and solving a linear least squares problem.
These have superlinear convergence at best: $\|e_{k+1}\| \leq C\|e_k\|^{\alpha \geq 1}$

Newton's Method: The workhorse of nonlinear solvers



- Standard form of a nonlinear system

$$F(u) = 0$$

- Iteration

$$\begin{aligned} \text{Solve:} \quad & J(u)w = -F(u) \\ \text{Update:} \quad & u^+ \leftarrow u + w \end{aligned}$$

  Where the Jacobian $J(u) = F'(u) = \frac{\partial F(u)}{\partial u}$.

- Quadratically convergent near a root: $\left|u^{n+1} - u^*\right| \in \mathcal{O}\left(\left|u^n - u^*\right|^2\right)$

Example (Nonlinear Poisson)

$$\begin{aligned} F(u) = 0 \quad & \sim \quad -\nabla\cdot\left[(1+u^2)\nabla u\right] - f = 0 \\ J(u)w \quad & \sim \quad -\nabla\cdot\left[(1+u^2)\nabla w + 2uw\nabla u\right] \end{aligned}$$

High-Performance Nonlinear Solvers Are Available Via

PETSc TAO

PETSc: Portable, Extensible Toolkit for Scientific Computation

SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers

Trilinos: NOX (Nonlinear Object-Oriented Solutions) and LOCA (Library of Continuation Algorithms) packages.

This presentation focuses on PETSc.

PETSc TAO

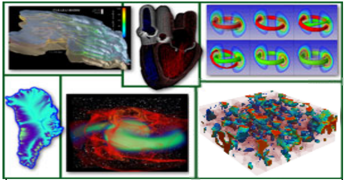
Portable, Extensible Toolkit for Scientific Computation / Toolkit for Advanced Optimization

Scalable algebraic solvers for PDEs. Encapsulate parallelism in high-level objects. Active & supported user community. Full API from Fortran, C/C++, Python.

- Optimization
- Time Integrators
- Nonlinear Algebraic Solvers
- Krylov Subspace Solvers
- Preconditioners
- Domain-Specific Interfaces
  - Networks
  - Quadtree / Octree
  - Unstructured Mesh
  - Structured Mesh
- Vectors | Index Sets | Matrices
- Computation & Communication Kernels

- **Easy customization and composability of solvers at runtime**
  - Enables optimality via flexible combinations of physics, algorithmics, architectures
  - Try new algorithms by composing new/existing algorithms (multilevel, domain decomposition, splitting, etc.)
- **Portability & performance**
  - Largest DOE machines, also clusters, laptops
  - Thousands of users worldwide

Argonne NATIONAL LABORATORY



PETSc provides the backbone of diverse scientific applications.
clockwise from upper left: hydrology, cardiology, fusion, multiphase steel, relativistic matter, ice sheet modeling

ECP xSDK FASTMATH R&D 100

https://www.mcs.anl.gov/petsc

EXASCALE COMPUTING PROJECT

Overview of PETSc Components

Functionality	More Details (Algorithms, Data Structures, etc.)
Optimization (TAO)	PDE Constrained, Adjoint Based, Derivative Free, Others
Time Integrators (TS)	Pseudo-transient, Runge-Kutta, Strong Stability Preserving, Others General Linear, IMEX, Rosenbrock-W
Nonlinear Algebraic Solvers (SNES)	Line Search Newton, Quasi-Newton (BFGS), Nonlinear Gauss Seidel, Nonlinear CG Trust Region Newton, Nonlinear Multigrid (FAS), Successive Substitutions, Active Set VI
Krylov Subspace Solvers (KSP)	Pipeline methods, GMRES, Chebyshev, BiCG-Stabilized, CG Hierarchical Krylov, LSQR, SYMMLQ, TFQMR, Others
Preconditioners (PC)	Blocks (by field), Additive Schwarz, ILU/ICC, Schur Complement Algebraic Multigrid, Geometric Multigrid, App-specific, Others
Domain-Specific Interfaces: **Networks**	Infrastructure networks, e.g., electrical, gas, water
Domain-Specific Interfaces: **Quadtree / Octree**	Structured mesh refinement
Domain-Specific Interfaces: **Unstructured Mesh**	Complex domains with finite element and finite volume discretizations
Domain-Specific Interfaces: **Structured Mesh**	Simple domains and discretizations, e.g., finite difference methods
Vectors, **Index Sets**, **Matrices**	Compressed Sparse Row (AIJ), Block AIJ, Matrix Blocks (MatNest) Symmetric Block AIJ, Dense, GPU and Phread Matrices
Computation & Communication Kernels	MPI, OpenMP, MPI-IO, CUDA, Pthreads, BLAS, LAPACK, etc.

EXASCALE COMPUTING PROJECT

PETSc is a platform for experimentation!

Optimal solvers must consider the interplay among physics, algorithms, and architectures.
This need motivates several elements of the PETSc design philosophy:

Algebraic solvers must be

- Composable: Separately developed solvers should be easy to combine, by non-experts, to form a more powerful solver.
- Hierarchical: Outer solvers may iterate over all variables for a global problem, while inner solvers handle smaller subsets of physics, smaller physical subdomains, or coarser meshes.
- Nested: Outer solvers call nested inner solvers
- Extensible: Easily customized or extended by users

To facilitate experimentation, many solver configurations can be set at runtime; no need to recompile.

Inexact Newton; Newton-Krylov

In practice, incurring the expense of an exact solve for the Newton step is often not desirable:

Inexact Newton methods find an approximate Newton direction Δx_k that satisfies

$$\|F'(x_k)\Delta x_k + F(x_k)\| \leq \eta \|F(x_k)\| \tag{3}$$

for a forcing term $\eta \in [0, 1)$ (static or chosen adaptively via Eisenstat-Walker method, `-snes_ksp_ew`).

Newton-Krylov methods, which use Krylov subspace projection methods as the "inner", linear iterative solver, are a robust and widely-used variant.

PETSc provides a wide range of Krylov methods and linear preconditioners that can be accessed via runtime options (`-ksp_type <ksp_method> -pc_type <pc_method>`).

EXASCALE COMPUTING PROJECT

Globalization Strategies

Newton has quadratic convergence *only when the iterate is sufficiently close to the solution*. Far from the solution, the computed Newton step is often too large in magnitude.

In practice, some globalization strategy is often needed to expand the domain of convergence. PETSc offers several options; most common (and default) is backtracking line search.

Backtracking line search

- Replaces the full Newton step s with some scalar multiple: $x_{k+1} = x_k + \lambda_k s, \quad \lambda > 0$
- Introduce merit function $\phi(x) = \frac{1}{2}\|F(x)\|_2^2$, (approximately) find $\min_{\lambda > 0} \phi(x_k + \lambda s)$
- Accurate minimization not worth the expense; simply ensure sufficent decrease:

$$\phi(x_k + \lambda s) \leq \phi(x_k) + \alpha \lambda \, s^\top \nabla \phi(x_k) \tag{4}$$

  (α is a user-tunable parameter; defaults to `1e-4`)
- Builds polynomial model for $\phi(x_k + \lambda s)$ (default is cubic; change via `-snes_linesearch_order <n>`).

PETSc SNES Paradigm

The PETSc SNES (Scalable Nonlinear Equation Solvers) interface is based upon callback functions

- `FormFunction()`, set by `SNESSetFunction()`
- `FormJacobian()`, set by `SNESSetJacobian()`

When PETSc needs to evaluate the nonlinear residual $F(x)$,

- Solver calls the **user's** function
- User function gets application state through the `ctx` variable
  - PETSc *never* sees application data

SNES Function

The user provided function which calculates the nonlinear residual has signature

```
PetscErrorCode (*func)(SNES snes,Vec x,Vec r,void *ctx)
```

`x`: The current solution

`r`: The residual

`ctx`: The user context passed to `SNESSetFunction()`

- Use this to pass application information, e.g. physical constants

SNES Jacobian

The user provided function that calculates the Jacobian has signature

```
PetscErrorCode (*func)(SNES snes,Vec x,Mat J,
                       Mat Jpre,void *ctx)
```

`x`: The current solution

`J`: The Jacobian

`Jpre`: The Jacobian preconditioning matrix (possibly J itself)

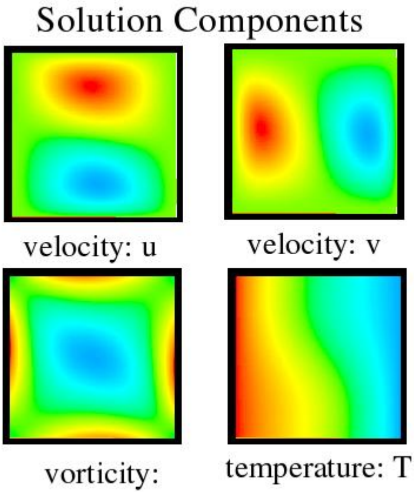
`ctx`: The user context passed to `SNESSetFunction()`

- Use this to pass application information, e.g. physical constants

Alternatively, you can use

- a builtin sparse finite difference approximation ("coloring")

SNES Example ex19, Steady-State Nonisothermal Lid-Driven Cavity



`src/snes/tutorials/ex19.c`

$$-\Delta U - \partial_y \Omega = 0$$
$$-\Delta V + \partial_x \Omega = 0$$
$$-\Delta\Omega + \nabla\cdot([U\Omega, V\Omega]) - \mathrm{Gr}\,\partial_x T = 0$$
$$-\Delta T + \mathrm{Pr}\,\nabla\cdot([UT, VT]) = 0$$

- 2D square domain with a moving lid; nonisothermal (temperature T)
- Flow driven by lid motion and buoyancy effects
- Velocity(U, V)-vorticity(Ω) formulation
- Finite difference discretization
- Logically regular grid
  - Parallelized with `DMDA`
- Analytical Jacobian not provided; Calculated by finite-differences (using coloring)
- Contributed by David Keyes

SNES Example

Driven Cavity Application Context

```c
/*
   User-defined data structures
*/

/* Collocated at each node */
typedef struct {
  PetscScalar u,v,omega,temp;
} Field;

typedef struct {
   PetscReal lidvelocity,prandtl,grashof;  /* physical parameters */
   PetscBool  draw_contours;                /* flag - 1 indicates drawing countours */
} AppCtx;
```


EXASCALE COMPUTING PROJECT

SNES Function with local evaluation

```
PetscErrorCode FormFunctionLocal(DMDALocalInfo *info,Field **x,Field **f,void *ptr)
{
  ...
  xints = info->xs; xinte = info->xs+info->xm; yints = info->ys; yinte = info->ys+info->ym;
  /* Handle boundaries ... */
  /* Compute over the interior points */

  for (j=yints; j<yinte; j++) {
    for (i=xints; i<xinte; i++) {
      /* convective coefficients for upwinding ... */
      /* U velocity */
      u          = x[j][i].u;
      uxx        = (2.0*u - x[j][i-1].u - x[j][i+1].u)*hydhx;
      uyy        = (2.0*u - x[j-1][i].u - x[j+1][i].u)*hxdhy;
      f[j][i].u  = uxx + uyy - .5*(x[j+1][i].omega-x[j-1][i].omega)*hx;
      /* V velocity, Omega ... */
      /* Temperature */
      u             = x[j][i].temp;
      uxx           = (2.0*u - x[j][i-1].temp - x[j][i+1].temp)*hydhx;
      uyy           = (2.0*u - x[j-1][i].temp - x[j+1][i].temp)*hxdhy;
      f[j][i].temp =  uxx + uyy + prandtl
        * (  (vxp*(u - x[j][i-1].temp) + vxm*(x[j][i+1].temp - u)) * hy
           + (vyp*(u - x[j-1][i].temp) + vym*(x[j+1][i].temp - u)) * hx);
}}}
```

$PETSC_DIR/src/snes/tutorials/ex19.c

Hands-on: Running the driven cavity

Run SNES ex19 with a single MPI rank (see full instructions for all hands-on exercises here):

```
./ex19 -snes_monitor -snes_converged_reason -da_grid_x 16 -da_grid_y 16 -da_refine
    2 -lidvelocity 100 -grashof 1e2
```

Hands-on: Running the driven cavity

Run SNES ex19 with a single MPI rank (see full instructions for all hands-on exercises here):

```
./ex19 -snes_monitor -snes_converged_reason -da_grid_x 16 -da_grid_y 16 -da_refine
    2 -lidvelocity 100 -grashof 1e2
```

Explanation of the above command-line options:

- -snes_monitor: Show progress of the SNES solver
- -snes_converged_reason: Print reason for SNES convergence or divergence
- -da_grid_x 16: Set initial grid points in x direction to 16
- -da_grid_y 16: Set initial grid poings in y direction to 16
- -da_refine 2: Refine the initial grid 2 times before creation
- -lidvelocity 100: Set dimensionless velocity of lid to 100
- -grashof 1e2: Set Grashof number to 1e2

An element of the PETSc design philosophy is extensive runtime customizability;
Use -help to enumerate and explain the various command-line options available.

Hands-on: Running the driven cavity

Run SNES ex19 with a single MPI rank (see full instructions for all hands-on exercises here):

```
./ex19 -snes_monitor -snes_converged_reason -da_grid_x 16 -da_grid_y 16 -da_refine
    2 -lidvelocity 100 -grashof 1e2
```

```
lid velocity = 100., prandtl # = 1., grashof # = 100.
  0 SNES Function norm 7.681163231938e+02
  1 SNES Function norm 6.582880149343e+02
  2 SNES Function norm 5.294044874550e+02
  3 SNES Function norm 3.775102116141e+02
  4 SNES Function norm 3.047226778615e+02
  5 SNES Function norm 2.599983722908e+00
  6 SNES Function norm 9.427314747057e-03
  7 SNES Function norm 5.212213461756e-08
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 7
Number of SNES iterations = 7
```

What is the SNES solver actually doing? Add -snes_view to see

```
SNES Object: 1 MPI processes
  type: newtonls
  maximum iterations=50, maximum function evaluations=10000
  tolerances: relative=1e-08, absolute=1e-50, solution=1e-08
  total number of linear solver iterations=835
  total number of function evaluations=11
  norm schedule ALWAYS
  Jacobian is built using colored finite differences on a DMDA
  SNESLineSearch Object: 1 MPI processes
    type: bt
      interpolation: cubic
      alpha=1.000000e-04
    maxstep=1.000000e+08, minlambda=1.000000e-12
    tolerances: relative=1.000000e-08, absolute=1.000000e-15, lambda=1.000000e-08
    maximum iterations=40
  KSP Object: 1 MPI processes
    type: gmres
      restart=30, using Classical (unmodified) Gram-Schmidt Orthogonalization with no iterative refinement
      happy breakdown tolerance 1e-30
    maximum iterations=10000, initial guess is zero
    tolerances:  relative=1e-05, absolute=1e-50, divergence=10000.
    left preconditioning
    using PRECONDITIONED norm type for convergence test
  PC Object: 1 MPI processes
    type: ilu
      out-of-place factorization
      0 levels of fill
      tolerance for zero pivot 2.22045e-14
      matrix ordering: natural
      factor fill ratio given 1., needed 1.
        Factored matrix follows:
          Mat Object: 1 MPI processes
            type: seqaij
            rows=14884, cols=14884, bs=4
            package used to perform factorization: petsc
            total: nonzeros=293776, allocated nonzeros=293776
                   [continued...]
```

Managing PETSc options

PETSc offers a very large number of runtime options.
All can be set via command line, but can also be set from input files and shell environment variables.

To facilitate readability, we'll put the command-line arguments common to the remaining hands-on exercises in `PETSC_OPTIONS`.

```
export PETSC_OPTIONS="-snes_monitor -snes_converged_reason -lidvelocity 100
    -da_grid_x 16 -da_grid_y 16 -ksp_converged_reason -log_view :log.txt"
```

We've added `-ksp_converged_reason` to see how and when linear solver halts.

We've also added `-log_view` to write the PETSc performance logging info to a file.
We don't have time to explain the performance logs; find the overall wall-clock time via

```
grep Time\ \(sec\): log.txt
```


EXASCALE COMPUTING PROJECT

Hands-on: Exact vs. Inexact Newton

PETSc defaults to inexact Newton. To run exact (and check the execution time), do

```
./ex19 -da_refine 2 -grashof 1e2 -pc_type lu
grep Time\ \(sec\): log.txt
```

Now run inexact Newton and vary the liner solve tolerance (`-ksp_rtol`).

```
./ex19 -da_refine 2 -grashof 1e2 -ksp_rtol 1e-8
./ex19 -da_refine 2 -grashof 1e2 -ksp_rtol 1e-5
./ex19 -da_refine 2 -grashof 1e2 -ksp_rtol 1e-3
./ex19 -da_refine 2 -grashof 1e2 -ksp_rtol 1e-2
./ex19 -da_refine 2 -grashof 1e2 -ksp_rtol 1e-1
```

What happens to the SNES iteration count? When does it diverge?
What yields the shortest execution time?

Hands-on: Scaling up grid size and running in parallel

What happens to iteration counts (and execution time) as we scale up the grid size?
For this exercise, run in parallel because experiments may take too long otherwise.
We also use BiCGStab (`-ksp_type bcgs`) because the default GMRES(30) fails for some cases.

Hands-on: Scaling up grid size and running in parallel

What happens to iteration counts (and execution time) as we scale up the grid size?
For this exercise, run in parallel because experiments may take too long otherwise.
We also use BiCGStab (`-ksp_type bcgs`) because the default GMRES(30) fails for some cases.

Run with default preconditioner. What happens to iteration counts and execution time?

```
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 2
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 3
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 4
```


EXASCALE COMPUTING PROJECT

Hands-on: Scaling up grid size and running in parallel

What happens to iteration counts (and execution time) as we scale up the grid size?
For this exercise, run in parallel because experiments may take too long otherwise.
We also use BiCGStab (`-ksp_type bcgs`) because the default GMRES(30) fails for some cases.

Run with default preconditioner. What happens to iteration counts and execution time?

```
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 2
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 3
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 4
```

```
$ mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -da_refine 4
lid velocity = 100., prandtl # = 1., grashof # = 100.
  0 SNES Function norm 1.545962539057e+03
  Linear solve converged due to CONVERGED_RTOL iterations 172
  1 SNES Function norm 9.780980584978e+02
  Linear solve converged due to CONVERGED_RTOL iterations 128
  2 SNES Function norm 6.620854219003e+02
  Linear solve converged due to CONVERGED_RTOL iterations 600
  3 SNES Function norm 3.219025282761e+00
  Linear solve converged due to CONVERGED_RTOL iterations 470
  4 SNES Function norm 9.280944447516e-03
  Linear solve converged due to CONVERGED_RTOL iterations 467
  5 SNES Function norm 1.354460792476e-07
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 5
```

Hands-on: Scaling up grid size and running in parallel

What happens to iteration counts (and execution time) as we scale up the grid size?
For this exercise, run in parallel because experiments may take too long otherwise.
We also use BiCGStab (`-ksp_type bcgs`) because the default GMRES(30) fails for some cases.

Let's try geometric multigrid (defaults to V-cycle) by adding `-pc_type mg`

```
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 2
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 3
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 4
```

Hands-on: Scaling up grid size and running in parallel

What happens to iteration counts (and execution time) as we scale up the grid size?
For this exercise, run in parallel because experiments may take too long otherwise.
We also use BiCGStab (`-ksp_type bcgs`) because the default GMRES(30) fails for some cases.

Let's try geometric multigrid (defaults to V-cycle) by adding `-pc_type mg`

```
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 2
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 3
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 4
```

```
mpirun -n 12 ./ex19 -ksp_type bcgs -grashof 1e2 -pc_type mg -da_refine 4
lid velocity = 100., prandtl # = 1., grashof # = 100.
  0 SNES Function norm 1.545962539057e+03
  Linear solve converged due to CONVERGED_RTOL iterations 6
  1 SNES Function norm 9.778196290981e+02
  Linear solve converged due to CONVERGED_RTOL iterations 6
  2 SNES Function norm 6.609659458090e+02
  Linear solve converged due to CONVERGED_RTOL iterations 7
  3 SNES Function norm 2.791922927549e+00
  Linear solve converged due to CONVERGED_RTOL iterations 6
  4 SNES Function norm 4.973591997243e-03
  Linear solve converged due to CONVERGED_RTOL iterations 6
  5 SNES Function norm 3.241555827567e-05
  Linear solve converged due to CONVERGED_RTOL iterations 9
  6 SNES Function norm 9.883136583477e-10
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 6
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
```

```
./ex19 -da_refine 2 -grashof 1.3e4
lid velocity = 100., prandtl # = 1., grashof # = 13000.
  0 SNES Function norm 7.971152173639e+02
  Linear solve did not converge due to DIVERGED_ITS iterations 10000
Nonlinear solve did not converge due to DIVERGED_LINEAR_SOLVE iterations 0
```

Oops! Failure in the linear solver? What if we use a stronger preconditioner?

EXASCALE COMPUTING PROJECT

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
```

```
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
lid velocity = 100., prandtl # = 1., grashof # = 13000.
  ...
  4 SNES Function norm 3.209967262833e+02
  Linear solve converged due to CONVERGED_RTOL iterations 9
  5 SNES Function norm 2.121900163587e+02
  Linear solve converged due to CONVERGED_RTOL iterations 9
  6 SNES Function norm 1.139162432910e+01
  Linear solve converged due to CONVERGED_RTOL iterations 8
  7 SNES Function norm 4.048269317796e-01
  Linear solve converged due to CONVERGED_RTOL iterations 8
  8 SNES Function norm 3.264993685206e-04
  Linear solve converged due to CONVERGED_RTOL iterations 8
  9 SNES Function norm 1.154893029612e-08
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 9
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type mg
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type mg
```

```
lid velocity = 100., prandtl # = 1., grashof # = 13373.
...
 48 SNES Function norm 3.124919801005e+02
  Linear solve converged due to CONVERGED_RTOL iterations 17
 49 SNES Function norm 3.124919800338e+02
  Linear solve converged due to CONVERGED_RTOL iterations 17
 50 SNES Function norm 3.124919799645e+02
Nonlinear solve did not converge due to DIVERGED_MAX_IT iterations 50
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type lu
```

Hands-on: Increasing strength of the nonlinearity

What happens as we increase the nonlinearity by raising the Grashof number?

```
./ex19 -da_refine 2 -grashof 1e2
./ex19 -da_refine 2 -grashof 1e3
./ex19 -da_refine 2 -grashof 1e4
./ex19 -da_refine 2 -grashof 1.3e4
./ex19 -da_refine 2 -grashof 1.3e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type lu
```

```
./ex19 -da_refine 2 -grashof 1.3373e4 -pc_type lu
...
 48 SNES Function norm 3.193724239842e+02
  Linear solve converged due to CONVERGED_RTOL iterations 1
 49 SNES Function norm 3.193724232621e+02
  Linear solve converged due to CONVERGED_RTOL iterations 1
 50 SNES Function norm 3.193724181714e+02
Nonlinear solve did not converge due to DIVERGED_MAX_IT iterations 50
```

A strong linear solver can't help us here. What now?

Let's try combining Newton's method with one of the other nonlinear solvers we mentioned in the introduction, using PETSc's support for nonlinear composition and preconditioning.

Abstract Nonlinear System and Solver

To discuss nonlinear composition and preconditioning, we introduce some definitions and notation.

Our prototypical nonlinear equation is of the form

$$F(x) = b \tag{5}$$

and we define the residual as

$$r(x) = F(x) - b \tag{6}$$

We use the notation $x_{k+1} = \mathcal{M}(F, x_k, b)$ for the action of a nonlinear solver.

Nonlinear Composition: Additive

Nonlinear composition consists of a sequence of two (or more) methods $\mathcal{M}$ and $\mathcal{N}$, which both provide an approximation solution to $F(x) = b$.

In the linear case, application of a stationary solver by defect correct can be written as

$$x_{k+1} = x_k - P^{-1}(Ax_k - b) \tag{7}$$

where P^{-1} is a linear preconditioner. (Richardson iteration applied to a preconditioned system.)

An additive composition of preconditioners P^{-1} and Q^{-1} with weights α and β may be written as

$$x_{k+1} = x_k - (\alpha P^{-1} + \beta Q^{-1})(Ax_k - b) \tag{8}$$

Analogously, for the nonlinear case, additive composition is

$$x_{k+1} = x_k + \alpha \cdot (\mathcal{M}(F, x_k, b) - x_k) + \beta \cdot (\mathcal{N}(F, x_k, b) - x_k) \tag{9}$$

Nonlinear Composition: Multiplicative

A multiplicative combination of linear preconditioners may be written as

$$\begin{aligned} x_{k+1/2} &= x_k - P^{-1}(Ax_k - b), \\ x_{k+1} &= x_{k+1/2} - Q^{-1}(Ax_{k+1/2} - b), \end{aligned} \tag{10}$$

Analogously, for the nonlinear case

$$x_{k+1} = \mathcal{M}(F, \mathcal{N}(F, x_k, b), b) \tag{11}$$

which simply indicates to update the solution using the current solution and residual with the first solver and then update the solution again using the resulting new solution and new residual with the second solver.

Nonlinear Left Preconditioning

Recall that the stationary iteration for our left-preconditioned linear system is

$$x_{k+1} = x_k - P^{-1}(Ax_k - b) \tag{12}$$

And since $Ax_k - b = r$, for the linear case we can write the action of our solver $\mathcal{N}$ as

$$\mathcal{N}(F, x, b) = x_k - P^{-1}r \tag{13}$$

With slight rearranging, we can express the left-preconditioned residual

$$P^{-1}r = x_k - \mathcal{N}(F, x, b) \tag{14}$$

And generalizing to the nonlinear case, the left preconditioning operation provides a modified residual

$$r_L = x_k - \mathcal{N}(F, x, b) \tag{15}$$

Nonlinear Right Preconditioning

For a right preconditioned linear system $AP^{-1}Px = b$, we solve the systems

$$\begin{aligned} AP^{-1}y &= b \\ x &= P^{-1}y \end{aligned} \tag{16}$$

Analogously, we define the right preconditioning operation in the nonlinear case as

$$\begin{aligned} y &= \mathcal{M}(F(\mathcal{N}(F, \cdot, b)), x_k, b) \\ x &= \mathcal{N}(F, y, b) \end{aligned} \tag{17}$$

(Note: In the linear case the above actually reduces to $A(I - P^{-1}A)y = (I - AP^{-1})b$, but the inner solver is applied before the function evaluation (matrix-vector product in the linear case), so we retain the "right preconditioning" name.)

Nonlinear Composition and Preconditioning

Type	Sym	Statement	Abbreviation
Additive	$+$	$x + \alpha(\mathcal{M}(\mathcal{F}, x, b) - x)$ $+ \beta(\mathcal{N}(\mathcal{F}, x, b) - x)$	$\mathcal{M} + \mathcal{N}$
Multiplicative	$*$	$\mathcal{M}(\mathcal{F}, \mathcal{N}(\mathcal{F}, x, b), b)$	$\mathcal{M} * \mathcal{N}$
Left Prec.	$-_L$	$\mathcal{M}(x - \mathcal{N}(\mathcal{F}, x, b), x, b)$	$\mathcal{M} -_L \mathcal{N}$
Right Prec.	$-_R$	$\mathcal{M}(\mathcal{F}(\mathcal{N}(\mathcal{F}, x, b)), x, b)$	$\mathcal{M} -_R \mathcal{N}$
Inner Lin. Inv.	$\backslash$	$y = J(x)^{-1} r(x) = \mathrm{K}(J(x), y_0, b)$	$\mathcal{N} \backslash \mathrm{K}$

Composing Scalable Nonlinear Algebraic Solvers,
Brune, Knepley, Smith, and Tu, SIAM Review, 2015.

For details on using nonlinear composition and preconditioning, see manual pages for `SNESCOMPOSITE` and `SNESGetNPC()`.

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type
    newtonls -npc_snes_max_it 4 -npc_pc_type mg
```

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type
    newtonls -npc_snes_max_it 4 -npc_pc_type mg
```

```
lid velocity = 100., prandtl # = 1., grashof # = 13373.
Nonlinear solve did not converge due to DIVERGED_INNER iterations 0
```

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type
    newtonls -npc_snes_max_it 4 -npc_pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type
    newtonls -npc_snes_max_it 4 -npc_pc_type lu
```

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type
    newtonls -npc_snes_max_it 4 -npc_pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type
    newtonls -npc_snes_max_it 4 -npc_pc_type lu
```

```
lid velocity = 100., prandtl # = 1., grashof # = 13373.
  0 SNES Function norm 7.987708558131e+02
  1 SNES Function norm 8.467169687854e+02
  2 SNES Function norm 7.300096001529e+02
  3 SNES Function norm 5.587232361127e+02
  4 SNES Function norm 3.071143076019e+03
  5 SNES Function norm 3.347748537471e+02
  6 SNES Function norm 1.383297972324e+01
  7 SNES Function norm 1.209841384629e-02
  8 SNES Function norm 8.660606193428e-09
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 8
```

So nonlinear Richardson preconditioned with Newton has let us go further than Newton alone.

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type
    newtonls -npc_snes_max_it 4 -npc_pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type
    newtonls -npc_snes_max_it 4 -npc_pc_type lu
./ex19 -da_refine 2 -grashof 1.4e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type lu
```

Hands-on: Nonlinear Richardson Preconditioned with Newton

```
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type
    newtonls -npc_snes_max_it 4 -npc_pc_type mg
./ex19 -da_refine 2 -grashof 1.3373e4 -snes_type nrichardson -npc_snes_type
    newtonls -npc_snes_max_it 4 -npc_pc_type lu
./ex19 -da_refine 2 -grashof 1.4e4 -snes_type nrichardson -npc_snes_type newtonls
    -npc_snes_max_it 4 -npc_pc_type lu
```

```
lid velocity = 100., prandtl # = 1., grashof # = 14000.
...
 37 SNES Function norm 5.992348444448e+02
 38 SNES Function norm 5.992348444290e+02
Nonlinear solve did not converge due to DIVERGED_INNER iterations 38
```

We've hit another barrier. What about switching things up?
Let's try preconditioning Newton with nonlinear Richardson.

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000
```

```
...
352 SNES Function norm 2.145588832260e-02
  Linear solve converged due to CONVERGED_RTOL iterations 7
353 SNES Function norm 1.288292314235e-05
  Linear solve converged due to CONVERGED_RTOL iterations 8
354 SNES Function norm 3.219155715396e-10
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 354
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
```

```
...
 23 SNES Function norm 4.796734188970e+00
  Linear solve converged due to CONVERGED_RTOL iterations 7
 24 SNES Function norm 2.083806106198e-01
  Linear solve converged due to CONVERGED_RTOL iterations 8
 25 SNES Function norm 1.368771861149e-04
  Linear solve converged due to CONVERGED_RTOL iterations 8
 26 SNES Function norm 1.065794992653e-08
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 26
```


EXASCALE COMPUTING PROJECT

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 4
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 5
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 6
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 7
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 4
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 5
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 6
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 7
```

```
lid velocity = 100., prandtl # = 1., grashof # = 14000.
  0 SNES Function norm 8.016512665033e+02
  Linear solve converged due to CONVERGED_RTOL iterations 11
  1 SNES Function norm 7.961475922316e+03
  Linear solve converged due to CONVERGED_RTOL iterations 10
  2 SNES Function norm 3.238304139699e+03
  Linear solve converged due to CONVERGED_RTOL iterations 10
  3 SNES Function norm 4.425107973263e+02
  Linear solve converged due to CONVERGED_RTOL iterations 9
  4 SNES Function norm 2.010474128858e+02
  Linear solve converged due to CONVERGED_RTOL iterations 8
  5 SNES Function norm 2.936958163548e+01
  Linear solve converged due to CONVERGED_RTOL iterations 8
  6 SNES Function norm 1.183847022611e+00
  Linear solve converged due to CONVERGED_RTOL iterations 8
  7 SNES Function norm 6.662829301594e-03
  Linear solve converged due to CONVERGED_RTOL iterations 7
  8 SNES Function norm 6.170083332176e-07
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 8
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 4
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 5
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 6
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 7
```

Newton preconditioned with nonlinear Richardson can be pushed quite far! Try

```
./ex19 -da_refine 2 -grashof 1e6 -pc_type lu -npc_snes_type nrichardson -npc_snes_max_it 7
     -snes_max_it 1000
```

Hands-on: Newton Preconditioned with Nonlinear Richardson

```
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 1
     -snes_max_it 1000
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 3
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 4
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 5
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 6
./ex19 -da_refine 2 -grashof 1.4e4 -pc_type mg -npc_snes_type nrichardson -npc_snes_max_it 7
```

Newton preconditioned with nonlinear Richardson can be pushed quite far! Try

```
./ex19 -da_refine 2 -grashof 1e6 -pc_type lu -npc_snes_type nrichardson -npc_snes_max_it 7
     -snes_max_it 1000
```

```
lid velocity = 100., prandtl # = 1., grashof # = 1e+06
...
 69 SNES Function norm 4.241700887134e+00
  Linear solve converged due to CONVERGED_RTOL iterations 1
 70 SNES Function norm 3.238739735055e+00
  Linear solve converged due to CONVERGED_RTOL iterations 1
 71 SNES Function norm 1.781881532852e+00
  Linear solve converged due to CONVERGED_RTOL iterations 1
 72 SNES Function norm 1.677710773493e-05
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 72
```

Takeaways

PETSc provides a wide assortment of nonlinear solvers through the (`SNES`) component

Users can build sophisticated solvers from composable algorithmic components:

- Inner, linear solves can employ full range of solvers and preconditioners provided by PETSc `KSP` and `PC`
  - Multigrid solvers particularly important for mesh size-independent convergence
- Composite nonlinear solvers can be built analogously, using building blocks from PETSc `SNES`

Newton-Krylov dominates, but large design space of "composed" nonlinear solvers to explore:

- Not well-explored theoretically or experimentally (interesting research opportunities!)
- Composed nonlinear solvers can be very powerful, though frustratingly fragile
  - Nonlinear Richardson, Gauss-Seidel, or NGMRES with Newton often improves robustness

Further items to explore include

- Nonlinear domain decomposition (`SNESASPIN/SNESNASM`) and nonlinear multigrid (or Full Approximation Scheme, `SNESFAS`) methods
- PETSc `TS` timesteppers use `SNES` to solve nonlinear problems at each time step
  - Pseudo-transient continuation (`TSPSEUDO`) can solve highly nonlinear steady-state problems