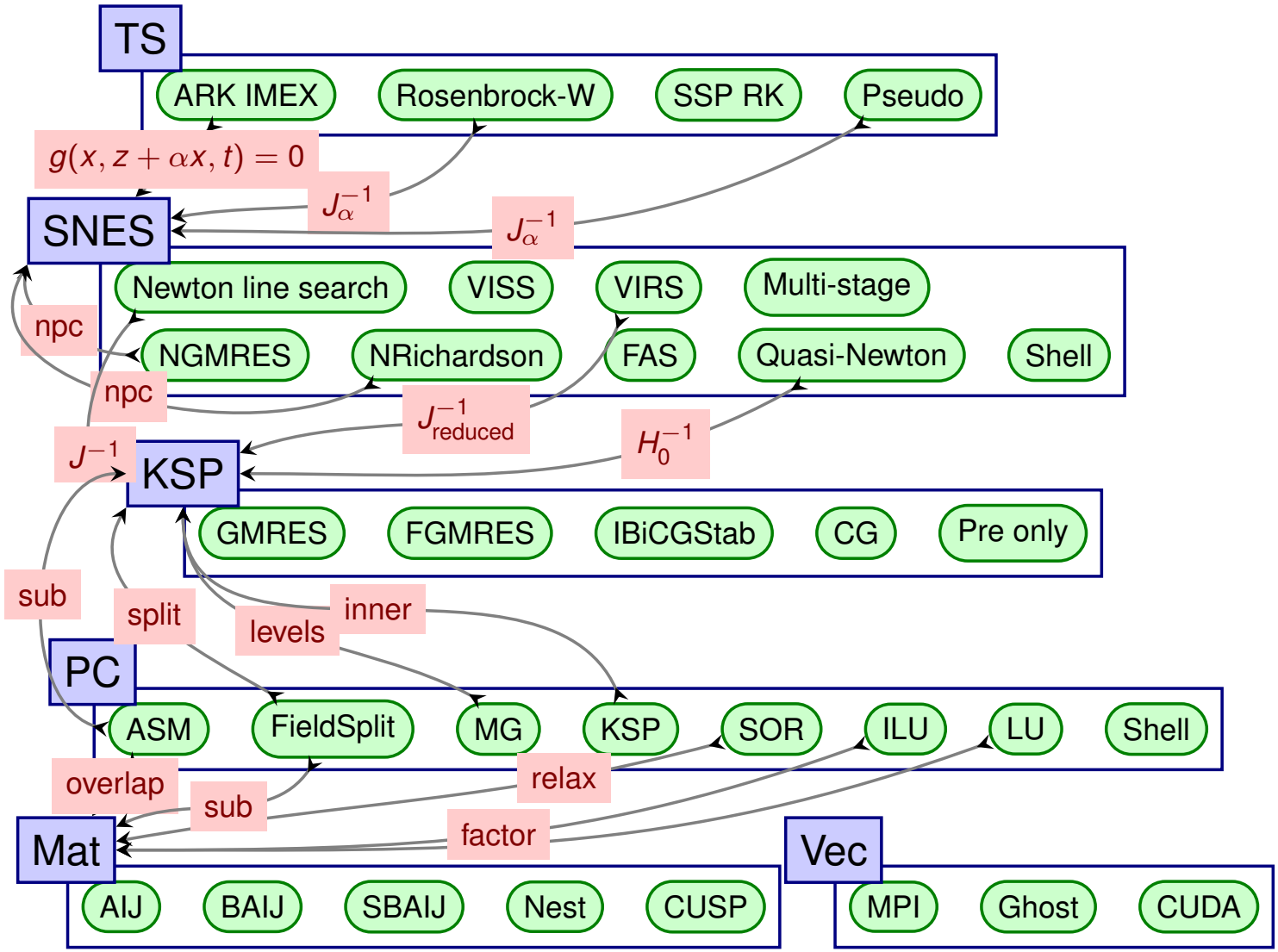
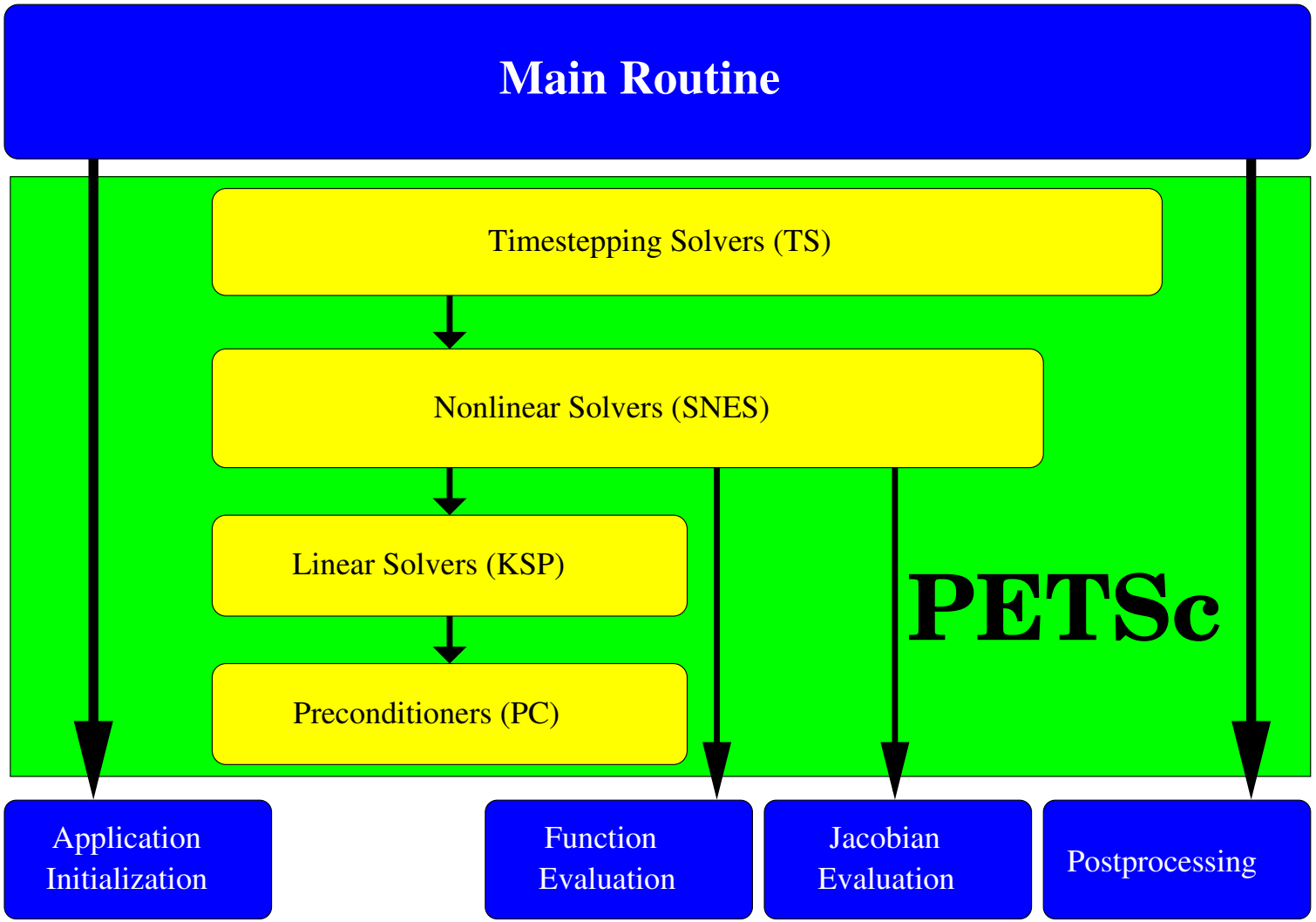
Interactions among composable linear, nonlinear, and timestepping solvers





- IGA used to evaluate nonlinear residuals
- PETSc `DA` used to manage parallelism.
- Adaptive time integration using method of lines.
  - Generalized α method from PETSc `TS`.

Nonlinear solvers in PETSc SNES

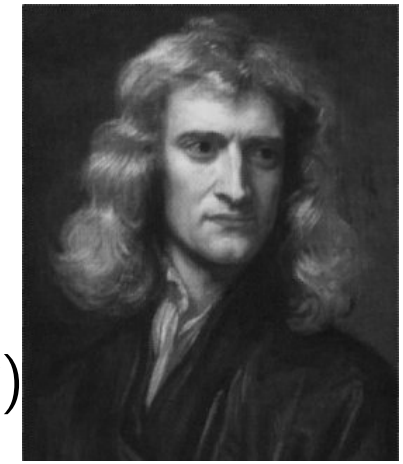
LS, TR	Newton-type with line search and trust region
NRichardson	Nonlinear Richardson, usually preconditioned
VIRS, VIRSAUG, and VISS	reduced space and semi-smooth methods for variational inequalities
QN	Quasi-Newton methods like BFGS
NGMRES	Nonlinear GMRES
NCG	Nonlinear Conjugate Gradients
SORQN	SOR quasi-Newton
GS	Nonlinear Gauss-Seidel sweeps
FAS	Full approximation scheme (nonlinear multigrid)
MS	Multi-stage smoothers, often used with FAS for hyperbolic problems
Shell	Your method, often used as a (nonlinear) preconditioner

Basic Solver Usage

We will illustrate basic solver usage with `SNES`.

- Use `SNESSetFromOptions()` so that everything is set dynamically
  - Use `-snes_type` to set the type or take the default
- Override the tolerances
  - Use `-snes_rtol` and `-snes_atol`
- View the solver to make sure you have the one you expect
  - Use `-snes_view`
- For debugging, monitor the residual decrease
  - Use `-snes_monitor`
  - Use `-ksp_monitor` to see the underlying linear solver

Newton iteration: workhorse of SNES



- Standard form of a nonlinear system

$$F(u) = 0$$

- Iteration

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

- Quadratically convergent near a root:
  $\left|u^{n+1} - u^*\right| \in \mathcal{O}\left(\left|u^n - u^*\right|^2\right)$
- Picard is the same operation with a different $J(u)$

Example (Nonlinear Poisson)

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

SNES Paradigm

The SNES 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

- Using PETSc linear algebra, just add:
  - `SNESSetFunction(SNES snes, Vec r, residualFunc, void *ctx)`
  - `SNESSetJacobian(SNES snes, Mat A, Mat M, jacFunc, void *ctx)`
  - `SNESSolve(SNES snes, Vec b, Vec x)`
- Can access subobjects
  - `SNESGetKSP(SNES snes, KSP *ksp)`
- Can customize subobjects from the cmd line
  - Set the subdomain preconditioner to ILU with `-sub_pc_type ilu`

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

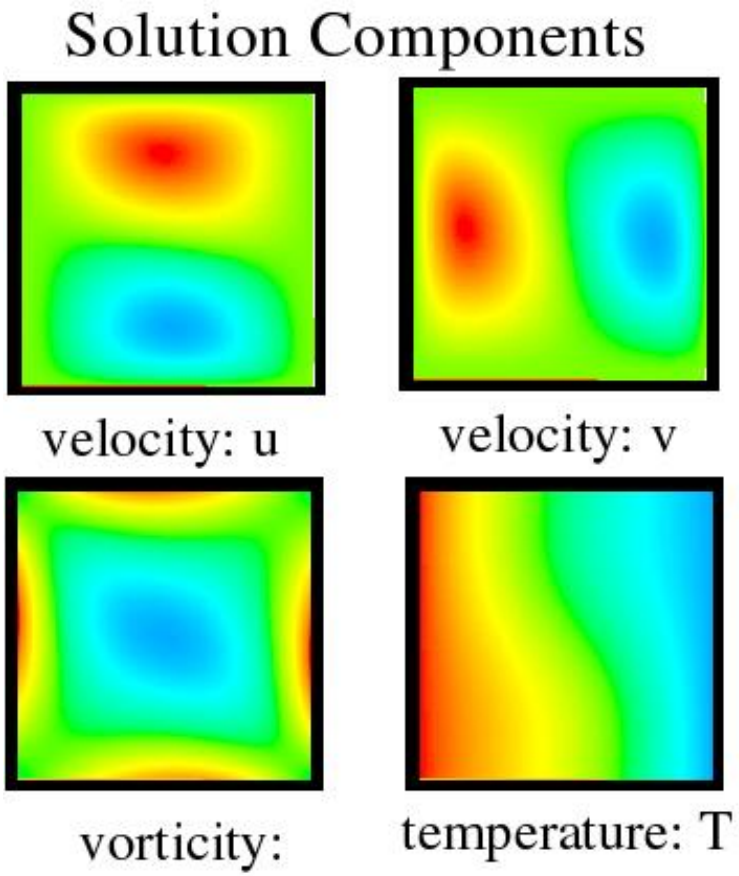
```
PetscErrorCode (*func)(SNES snes,Vec x,Mat J,
                       Mat Jpre,void *ctx)
```

- `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")
- automatic differentiation (ADIC/ADIFOR)

SNES Example
Driven Cavity



- Velocity-vorticity formulation
- Flow driven by lid and/or bouyancy
- Logically regular grid
  - Parallelized with `DMDA`
- Finite difference discretization
- Authored by David Keyes

`src/snes/examples/tutorials/ex19.c`

SNES Example
Driven Cavity Application Context

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

typedef struct {
       /* physical parameters */
   PassiveReal lidvelocity,prandtl,grashof;
       /* color plots of the solution */
   PetscTruth  draw_contours;
} AppCtx;
```

SNES Example

```
DrivenCavityFunction(SNES snes, Vec X, Vec F, void *ptr) {
  AppCtx         *user = (AppCtx *) ptr;
  /* local starting and ending grid points */
  PetscInt       istart, iend, jstart, jend;
  PetscScalar    *f;                /* local vector data */
  PetscReal      grashof = user->grashof;
  PetscReal      prandtl = user->prandtl;
  PetscErrorCode ierr;

  /* Code to communicate nonlocal ghost point data */
  DMDAVecGetArray(da, F, &f);

  /* Loop over local part and assemble into f[idxloc] */
  /* .... */

  DMDAVecRestoreArray(da, F, &f);
  return 0;
}
```

DMDA Local Function

User provided function calculates the nonlinear residual (in 2D)

```
(*lfunc)(DMDALocalInfo *info, PetscScalar **x, PetscScalar **r, void *ctx)
```

`info`: All layout and numbering information

`x`: The current solution

- Notice that it is a multidimensional array

`r`: The residual

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

The local DMDA function is activated by calling

```
SNESSetFunction(snes, r, SNESDAFormFunction, ctx)
```

SNES Example with local evaluation

```
PetscErrorCode DrivenCavityFuncLocal(DMDALocalInfo *info,
                     Field **x,Field **f,void *ctx) {
  /* Handle boundaries ... */
  /* Compute over the interior points */
  for(j = info->ys; j < info->ys+info->ym; j++) {
    for(i = info->xs; i < info->xs+info->xm; 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
      /* V velocity, Omega ... */
      /* Temperature */
      u             = x[j][i].temp;
      uxx           = (2.0*u - x[j][i-1].temp - x[j][i+1].temp)*hy
      uyy           = (2.0*u - x[j-1][i].temp - x[j+1][i].temp)*hx
      f[j][i].temp =  uxx + uyy + prandtl
        * (   (vxp*(u - x[j][i-1].temp) + vxm*(x[j][i+1].temp - u))
            + (vyp*(u - x[j-1][i].temp) + vym*(x[j+1][i].temp - u))
```

DMDA Local Jacobian

User provided function calculates the Jacobian (in 2D)

```
(*lfunc)(DMDALocalInfo *info, PetscScalar **x, Mat J, void *ctx)
```

`info`: All layout and numbering information

`x`: The current solution

`J`: The Jacobian

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

The local DMDA function is activated by calling

```
SNESSetJacobian(snes, J, J, SNESDAComputeJacobian, ctx)
```

Running the driven cavity

- `./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2 -pc_type lu`
- Uh oh, we have convergence problems
- Does `-snes_grid_sequence` help?

Running the driven cavity

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

```
lid velocity = 100, prandtl # = 1, grashof # = 1000
0 SNES Function norm 7.682893957872e+02
1 SNES Function norm 6.574700998832e+02
2 SNES Function norm 5.285205210713e+02
3 SNES Function norm 3.770968117421e+02
4 SNES Function norm 3.030010490879e+02
5 SNES Function norm 2.655764576535e+00
6 SNES Function norm 6.208275817215e-03
7 SNES Function norm 1.191107243692e-07
Number of SNES iterations = 7
```

- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2 -pc_type lu`
- Uh oh, we have convergence problems
- Does `-snes_grid_sequence` help?

Running the driven cavity

- `./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`

```
lid velocity = 100, prandtl # = 1, grashof # = 10000
0 SNES Function norm 7.854040793765e+02
1 SNES Function norm 6.630545177472e+02
2 SNES Function norm 5.195829874590e+02
3 SNES Function norm 3.608696664876e+02
4 SNES Function norm 2.458925075918e+02
5 SNES Function norm 1.811699413098e+00
6 SNES Function norm 4.688284580389e-03
7 SNES Function norm 4.417003604737e-08
Number of SNES iterations = 7
```

- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2 -pc_type lu`
- Uh oh, we have convergence problems
- Does `-snes_grid_sequence` help?

Running the driven cavity

- `./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2 -pc_type lu`

```
lid velocity = 100, prandtl # = 1, grashof # = 100000
0 SNES Function norm 1.809960438828e+03
1 SNES Function norm 1.678372489097e+03
2 SNES Function norm 1.643759853387e+03
3 SNES Function norm 1.559341161485e+03
4 SNES Function norm 1.557604282019e+03
5 SNES Function norm 1.510711246849e+03
6 SNES Function norm 1.500472491343e+03
7 SNES Function norm 1.498930951680e+03
8 SNES Function norm 1.498440256659e+03
...
```

- Uh oh, we have convergence problems
- Does `-snes_grid_sequence` help?

Running the driven cavity

- `./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2`
- `./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -snes_view -da_refine 2 -pc_type lu`
- Uh oh, we have convergence problems
- Does `-snes_grid_sequence` help?

Exercise 5

Run SNES Example 5 using come custom options.

1. `cd $PETSC_DIR/src/snes/examples/tutorials`
2. `make ex5`
3. `mpiexec ./ex5 -snes_monitor -snes_view`
4. `mpiexec ./ex5 -snes_type tr -snes_monitor -snes_view`
5. `mpiexec ./ex5 -ksp_monitor -snes_monitor -snes_view`
6. `mpiexec ./ex5 -pc_type jacobi -ksp_monitor -snes_monitor -snes_view`
7. `mpiexec ./ex5 -ksp_type bicg -ksp_monitor -snes_monitor -snes_view`

Sample output (SNES and KSP)

```
SNES Object: 1 MPI processes
  type: ls
    line search variant: CUBIC
    alpha=1.000000000000e-04, maxstep=1.000000000000e+08, minlambd
    damping factor=1.000000000000e+00
  maximum iterations=50, maximum function evaluations=10000
  tolerances: relative=1e-08, absolute=1e-50, solution=1e-08
  total number of linear solver iterations=5
  total number of function evaluations=6
  KSP Object:   1 MPI processes
    type: gmres
      GMRES: restart=30, using Classical (unmodified) Gram-Schmidt
      GMRES: 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
```

Exercise 6

Create a new code based upon SNES Example 5.

1. Create a new directory
   - `mkdir -p /home/knepley/proj/newsim/src`
2. Copy the source
   - `cp ex5.c /home/knepley/proj/newsim/src`
   - Add `myStuff.c` and `myStuff2.F`
3. Create a PETSc makefile
   - `bin/ex5: src/ex5.o src/myStuff.o src/myStuff2.o`
   - `${CLINKER} -o $@ $^ ${PETSC_SNES_LIB}`
   - `include ${PETSC_DIR}/conf/variables`
   - `include ${PETSC_DIR}/conf/rules`

To get the project ready-made

```
hg clone http://petsc.cs.iit.edu/petsc/tutorials/SimpleTutorial newsim
```

Multiphysics Assembly Code: Residuals

```
FormFunction_Coupled(SNES snes,Vec X,Vec F,void *ctx) {
  struct UserCtx *user = ctx;
  // ...
  SNESGetDM(snes,&pack);
  DMCompositeGetEntries(pack,&dau,&dak);
  DMDAGetLocalInfo(dau,&infou);
  DMDAGetLocalInfo(dak,&infok);
  DMCompositeScatter(pack,X,Uloc,Kloc);
  DMDAVecGetArray(dau,Uloc,&u);
  DMDAVecGetArray(dak,Kloc,&k);
  DMCompositeGetAccess(pack,F,&Fu,&Fk);
  DMDAVecGetArray(dau,Fu,&fu);
  DMDAVecGetArray(dak,Fk,&fk);
  FormFunctionLocal_U(user,&infou,u,k,fu); // u residual with k gi
  FormFunctionLocal_K(user,&infok,u,k,fk); // k residual with u gi
  DMDAVecRestoreArray(dau,Fu,&fu);
  // More restores
```

Multiphysics Assembly Code: Jacobians

```
FormJacobian_Coupled(SNES snes,Vec X,Mat J,Mat B,...) {
  // Access components as for residuals
  MatGetLocalSubMatrix(B,is[0],is[0],&Buu);
  MatGetLocalSubMatrix(B,is[0],is[1],&Buk);
  MatGetLocalSubMatrix(B,is[1],is[0],&Bku);
  MatGetLocalSubMatrix(B,is[1],is[1],&Bkk);
  FormJacobianLocal_U(user,&infou,u,k,Buu);          // single phys
  FormJacobianLocal_UK(user,&infou,&infok,u,k,Buk); // coupling
  FormJacobianLocal_KU(user,&infou,&infok,u,k,Bku); // coupling
  FormJacobianLocal_K(user,&infok,u,k,Bkk);          // single phys
  MatRestoreLocalSubMatrix(B,is[0],is[0],&Buu);
  // More restores
```

- Assembly code is independent of matrix format
- Single-physics code is used unmodified for coupled problem
- No-copy fieldsplit:

  `-pack_dm_mat_type nest -pc_type fieldsplit`
- Coupled direct solve:

  `-pack_dm_mat_type aij -pc_type lu -pc_factor_mat_solver_package mumps`

Finite Difference Jacobians

PETSc can compute and explicitly store a Jacobian via 1st-order FD

- Dense
  - Activated by `-snes_fd`
  - Computed by `SNESDefaultComputeJacobian()`
- Sparse via colorings
  - Activated by `-snes_fd_color` (default when no Jacobian set and using DM)
  - Coloring is created by `MatFDColoringCreate()`
  - Computed by `SNESDefaultComputeJacobianColor()`

Can also use Matrix-free Newton-Krylov via 1st-order FD

- Activated by `-snes_mf` without preconditioning
- Activated by `-snes_mf_operator` with user-defined preconditioning
  - Uses preconditioning matrix from `SNESSetJacobian()`

SNES Variants

- Line search strategies
- Trust region approaches
- Picard iteration
- Variational inequality approaches

Variational Inequalities

- Supports inequality and box constraints on solution variables.
- Solution methods
  - Semismooth Newton
    - reformulate problem as a non-smooth system, Newton on subdifferential
    - Newton step solves diagonally perturbed systems
  - Active set
    - similar linear algebra to solving PDE
    - solve in reduced space by eliminating constrained variables
    - or enforce constraints by Lagrange multipliers
    - sometimes slower convergence or “bouncing”
- composes with multigrid and field-split
- demonstrated optimality for phase-field problems with millions of degrees of freedom

Why isn't SNES converging?

- The Jacobian is wrong (maybe only in parallel)
  - Check with `-snes_type test` and `-snes_mf_operator -pc_type lu`
- The linear system is not solved accurately enough
  - Check with `-pc_type lu`
  - Check `-ksp_monitor_true_residual`, try right preconditioning
- The Jacobian is singular with inconsistent right side
  - Use `MatNullSpace` to inform the `KSP` of a known null space
  - Use a different Krylov method or preconditioner
- The nonlinearity is just really strong
  - Run with `-snes_linesearch_monitor`
  - Try using trust region instead of line search `-snes_type newtontr`
  - Try grid sequencing if possible
  - Use a continuation

SNES Test

- PETSc can compute a finite difference Jacobian and compare it to yours
- `-snes_type test`
  - Is the difference significant?
- `-snes_type test -snes_test_display`
  - Are the entries in the star stencil correct?
- Find which line has the typo
- `$ git checkout 9-newton-correct`
- Check with `-snes_type test`
- and `-snes_mf_operator -pc_type lu`

Nonlinear solvers in PETSc SNES

LS, TR	Newton-type with line search and trust region
NRichardson	Nonlinear Richardson, usually preconditioned
VIRS, VISS	reduced space and semi-smooth methods for variational inequalities
QN	Quasi-Newton methods like BFGS
NGMRES	Nonlinear GMRES
NCG	Nonlinear Conjugate Gradients
GS	Nonlinear Gauss-Seidel/multiplicative Schwarz sweeps
FAS	Full approximation scheme (nonlinear multigrid)
MS	Multi-stage smoothers, often used with FAS for hyperbolic problems
Shell	Your method, often used as a (nonlinear) preconditioner

Taxonomy of implicit solvers

Global linearization: Picard and Newton

- Linear solve "$J(u)w = -F(u)$"
  - (sparse) direct vs. iterative (Krylov) with preconditioning
  - classical relaxation (Jacobi, Gauss-Seidel), incomplete factorization (ILU)
  - domain decomposition and multigrid
- Globalization: "$u_{\text{next}} = u + \alpha w$"
  - Line search, trust region, continuation

Inherently nonlinear methods

- Nonlinear GMRES, Nonlinear CG (can use preconditioning)
- Nonlinear domain decomposition
- Nonlinear multigrid: Full Approximation Scheme (FAS)

Taxonomy of implicit solvers

Global linearization: Picard and Newton

- Linear solve "$J(u)w = -F(u)$"
  - (sparse) direct vs. iterative (Krylov) with preconditioning
  - classical relaxation (Jacobi, Gauss-Seidel), incomplete factorization (ILU)
  - domain decomposition and multigrid
- Globalization: "$u_{\text{next}} = u + \alpha w$"
  - Line search, trust region, continuation

Inherently nonlinear methods

- Nonlinear GMRES, Nonlinear CG (can use preconditioning)
- Nonlinear domain decomposition
- Nonlinear multigrid: Full Approximation Scheme (FAS)

- These methods can be **scalable**.

Taxonomy of implicit solvers

Global linearization: Picard and Newton

- Linear solve "$J(u)w = -F(u)$"
  - (sparse) direct vs. iterative (Krylov) with preconditioning
  - classical relaxation (Jacobi, Gauss-Seidel), incomplete factorization (ILU)
  - domain decomposition and multigrid
- Globalization: "$u_{\text{next}} = u + \alpha w$"
  - Line search, trust region, continuation

Inherently nonlinear methods

- Nonlinear GMRES, Nonlinear CG (can use preconditioning)
- Nonlinear domain decomposition
- Nonlinear multigrid: Full Approximation Scheme (FAS)

- How nonlinear are the scales? How expensive is setup?

Full Approximation Scheme

$$\tilde{u}^h \leftarrow S^h_{\text{pre}} u^h_0 \qquad \text{pre-smooth}$$

$$L^H u^H = I^H_h f^h + \underbrace{L^H \hat{I}^H_h \tilde{u}^h - I^H_h L^h \tilde{u}^h}_{\tau^H_h} \qquad \text{solve coarse problem for } u^H$$

$$u^h \leftarrow S^h_{\text{post}} \left[\tilde{u}^h + I^h_H (u^H - \hat{I}^H_h \tilde{u}^h) \right] \qquad \text{apply correction and post-smooth}$$

- Nonlinearities from spatial discretization fixed locally
- No assembled matrices so better floating point utilization, less memory
- Makes progress on all physical components at once
- FD and DG good, less efficient for continuous finite element methods
- Influence of surface evolution is low rank, no need to visit finest level on each iteration

Nonlinear Multigrid

Most authors just offer an ansatz with nonlinear smoothing

$$x^{new} = S(x^{old}, b) \tag{1}$$

and coarse-grid correction

$$\begin{aligned} F_c(x_c) &= F_c(\tilde{x}_c) + \gamma R(b - F(x^{old})) && (2)\\ x^{new} &= x^{old} + \frac{1}{\gamma} R^T (x_c - \tilde{x}_c) && (3) \end{aligned}$$

where $\tilde{x}$ is an approximate solution.

If F is a linear operator L, the correction reduces to

$$\begin{aligned} L_c(x_c) &= L_c(\tilde{x}_c) + \gamma R(b - L(x^{old})) && (4)\\ L_c(x_c - \tilde{x}_c) &= \gamma R(b - L(x^{old})) && (5)\\ L_c \delta x_c &= \gamma R r && (6) \end{aligned}$$

Nonlinear Multigrid

Most authors just offer an ansatz with nonlinear smoothing

$$x^{new} = S(x^{old}, b) \tag{1}$$

and coarse-grid correction

$$\begin{aligned} F_c(x_c) &= F_c(\tilde{x}_c) + \gamma R(b - F(x^{old})) && (2) \\ x^{new} &= x^{old} + \frac{1}{\gamma} R^T (x_c - \tilde{x}_c) && (3) \end{aligned}$$

where $\tilde{x}$ is an approximate solution.

and the update becomes

$$\begin{aligned} x^{new} &= x^{old} + \frac{1}{\gamma} R^T \delta x_c && (4) \\ x^{new} &= x^{old} + R^T \hat{L}_c^{-1} R r && (5) \end{aligned}$$

Nonlinear Multigrid

It is instructive to look at the alternate derivation of Barry Smith

Begin with the nonlinear generalization $F(u) = 0$, for a correction

$$J_c x_c = R(b - Jx^{old}) \quad (6)$$
$$J_c x_c = -R(F(u) + Jx^{old}) \quad (7)$$

and then using Taylor series

$$F(u^{old}) = F(u) + J(u^{old} - u) + \ldots \quad (8)$$
$$F_c(u_c^{old} + x_c) = F_c(u_c^{old}) + J_c x_c + \ldots \quad (9)$$

we have the correction

$$F_c(u_c^{old} + x_c) - F_c(u_c^{old}) = -RF(u^{old}) \quad (10)$$
$$F_c(u_c^{old} + x_c) = F_c(u_c^{old}) - RF(u^{old}) \quad (11)$$

Nonlinear Multigrid

It is instructive to look at the alternate derivation of Barry Smith

Begin with the nonlinear generalization $F(u) = 0$, for a correction

$$J_c x_c = R(b - Jx^{old}) \tag{6}$$
$$J_c x_c = -R(F(u) + Jx^{old}) \tag{7}$$

and then using Taylor series

$$F(u^{old}) = F(u) + J(u^{old} - u) + \dots \tag{8}$$
$$F_c(u_c^{old} + x_c) = F_c(u_c^{old}) + J_c x_c + \dots \tag{9}$$

and the same update

$$x^{new} = x^{old} + R^T x_c \tag{10}$$

Nonlinear multigrid (full approximation scheme)

- V-cycle structure, but use nonlinear relaxation and skip the matrices
- `./ex19 -da_refine 4 -snes_monitor -snes_type nrichardson -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps 3 -npc_snes_type fas -npc_fas_levels_snes_type gs -npc_snes_max_it 1 -npc_snes_fas_smoothup 6 -npc_snes_fas_smoothdown 6 -lidvelocity 100 -grashof 4e4`
- `./ex19 -da_refine 4 -snes_monitor -snes_type ngmres -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps 3 -npc_snes_type fas -npc_fas_levels_snes_type gs -npc_snes_max_it 1 -npc_snes_fas_smoothup 6 -npc_snes_fas_smoothdown 6 -lidvelocity 100 -grashof 4e4`

Nonlinear multigrid (full approximation scheme)

- V-cycle structure, but use nonlinear relaxation and skip the matrices
- `./ex19 -da_refine 4 -snes_monitor -snes_type nrichardson -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps 3 -npc_snes_type fas -npc_fas_levels_snes_type gs -npc_snes_max_it 1 -npc_snes_fas_smoothup 6 -npc_snes_fas_smoothdown 6 -lidvelocity 100 -grashof 4e4`

```
lid velocity = 100, prandtl # = 1, grashof # = 40000
0 SNES Function norm 1.065744184802e+03
1 SNES Function norm 5.213040454436e+02
2 SNES Function norm 6.416412722900e+01
3 SNES Function norm 1.052500804577e+01
4 SNES Function norm 2.520004680363e+00
5 SNES Function norm 1.183548447702e+00
6 SNES Function norm 2.074605179017e-01
7 SNES Function norm 6.782387771395e-02
8 SNES Function norm 1.421602038667e-02
9 SNES Function norm 9.849816743803e-03
10 SNES Function norm 4.168854365044e-03
11 SNES Function norm 4.392925390996e-04
12 SNES Function norm 1.433224993633e-04
13 SNES Function norm 1.074357347213e-04
14 SNES Function norm 6.107933844115e-05
15 SNES Function norm 1.509756087413e-05
16 SNES Function norm 3.478180386598e-06

Number of SNES iterations = 16
```

- `./ex19 -da_refine 4 -snes_monitor -snes_type ngmres -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps`

Nonlinear multigrid (full approximation scheme)

- V-cycle structure, but use nonlinear relaxation and skip the matrices
- `./ex19 -da_refine 4 -snes_monitor -snes_type nrichardson -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps 3 -npc_snes_type fas -npc_fas_levels_snes_type gs -npc_snes_max_it 1 -npc_snes_fas_smoothup 6 -npc_snes_fas_smoothdown 6 -lidvelocity 100 -grashof 4e4`
- `./ex19 -da_refine 4 -snes_monitor -snes_type ngmres -npc_fas_levels_snes_type gs -npc_fas_levels_snes_gs_sweeps 3 -npc_snes_type fas -npc_fas_levels_snes_type gs -npc_snes_max_it 1 -npc_snes_fas_smoothup 6 -npc_snes_fas_smoothdown 6 -lidvelocity 100 -grashof 4e4`

```
lid velocity = 100, prandtl # = 1, grashof # = 40000
0 SNES Function norm 1.065744184802e+03
1 SNES Function norm 9.413549877567e+01
2 SNES Function norm 2.117533223215e+01
3 SNES Function norm 5.858983768704e+00
4 SNES Function norm 7.303010571089e-01
5 SNES Function norm 1.585498982242e-01
6 SNES Function norm 2.963278257962e-02
7 SNES Function norm 1.152790487670e-02
8 SNES Function norm 2.092161787185e-03
9 SNES Function norm 3.129419807458e-04
10 SNES Function norm 3.503421154426e-05
11 SNES Function norm 2.898344063176e-06

Number of SNES iterations = 11
```

Monolithic approaches

Parallel direct solver

```
-dm_mat_type aij -pc_type lu -pc_factor_mat_solver_package r
```

Coupled nonlinear multigrid accelerated by NGMRES with multi-stage smoothers

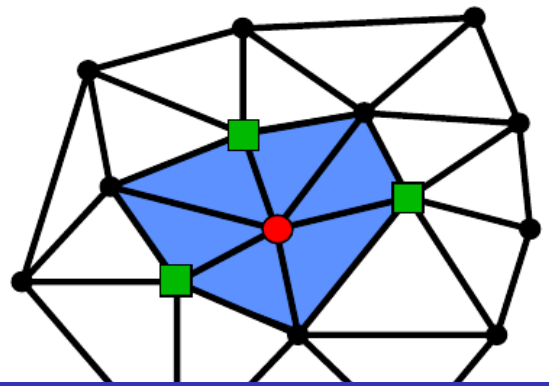
```
-lidvelocity 200 -grashof 1e4
-snes_grid_sequence 5 -snes_monitor -snes_view
-snes_type ngmres
-npc_snes_type fas
-npc_snes_max_it 1
-npc_fas_coarse_snes_type ls
-npc_fas_coarse_ksp_type preonly
-npc_fas_snes_type ms
-npc_fas_snes_max_it 1
-npc_fas_ksp_type preonly
-npc_fas_pc_type pbjacobi
-npc_fas_snes_ms_type m62
-npc_fas_snes_max_it 1
```

Nonlinear and matrix-free smoothing

- matrix-based smoothers require global linearization
- nonlinearity often more efficiently resolved locally
- nonlinear additive or multiplicative Schwarz
- nonlinear/matrix-free is good if

$$C = \frac{(\text{cost to evaluate residual at one “point”}) \cdot N}{(\text{cost of global residual})} \sim 1$$

  - finite difference: $C < 2$
  - finite volume: $C \sim 2$, depends on reconstruction
  - finite element: $C \sim$ number of vertices per cell
- larger block smoothers help reduce C
- additive correction (Jacobi/Chebyshev/multi-stage)
  - global evaluation, as good as $C = 1$
  - but, need to assemble corrector/scaling
  - need spectral estimates or wave speeds



Conclusions

Newton-Multigrid provides

- Good nonlinear solves
- Simple interface for software libraries
- Low computational efficiency

Multigrid-FAS provides

- Good nonlinear solves
- Lower memory bandwidth and storage
- Potentially high computational efficiency
- Requires formation on small systems “on the fly”