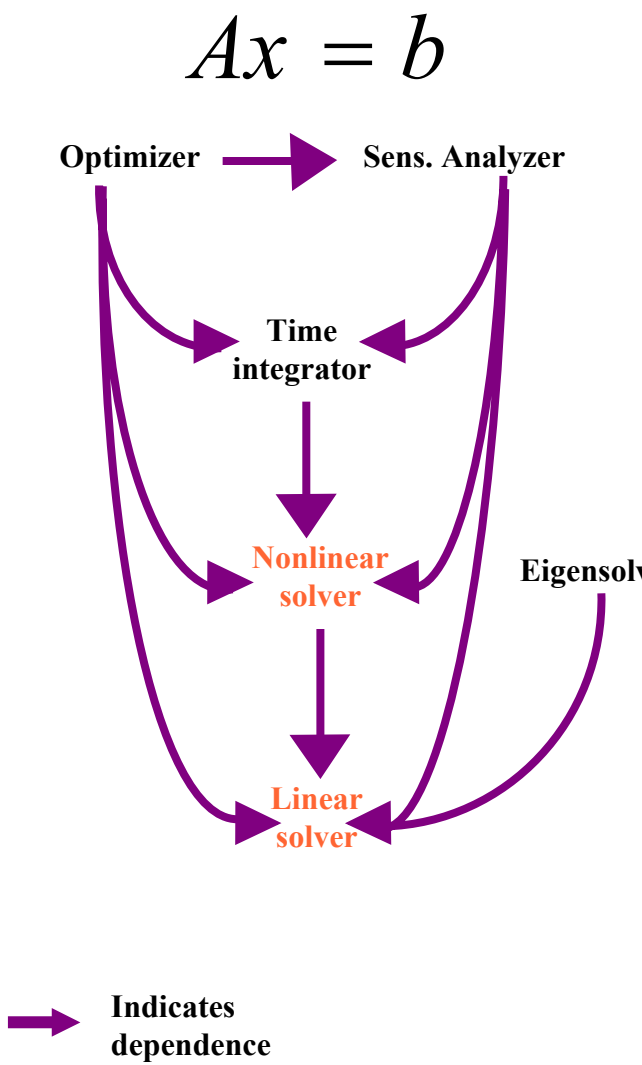
Unifying Solver Frameworks

Barry F. Smith

Argonne National Laboratory

Common software infrastructure for nonlinear PDE solvers

- User codes to the problem they are solving, not the algorithm used to solve the problem
- Implementation of various algorithms reuse common concepts and code when possible, without losing efficiency



Encompassing …

- **Newton's method**
  - **Direct solvers**
  - **Matrix-based preconditioned solvers**
  - **Matrix-free methods**
  - **Multigrid linear solvers (Newton-MG)**
    - **Matrix-based and matrix-free**
- **Nonlinear multigrid**
  - **aka Full approximation scheme (FAS)**
  - **aka MG-Newton**

Software engineering ingredients

- **Standard solver interfaces (SIDL)**
- **Solver libraries (obviously ☺)**
- **Automatic differentiation (AD)**
- **Code generation**

Testbed: 2D Hall MHD sawtooth instability

(Porcelli *et al.*, 1993, 1999)

Model equations:

$$
\begin{aligned}
\frac{\partial F}{\partial t} + [\phi, F] &= \rho_s^2 [U, \psi] \\
\frac{\partial U}{\partial t} + [\phi, U] &= [J, \psi] \\
F &= \psi + d_e^2 J \\
J &= -\nabla^2 \psi \\
U &= \nabla^2 \phi
\end{aligned}
$$

$$\text{with} [A, B] = \hat{z} \cdot \nabla A \times \nabla B$$

$$
\begin{aligned}
\vec{B} &= B_0 \hat{z} + \nabla \psi \times \hat{z} \\
\vec{v} &= \hat{z} \times \nabla \phi
\end{aligned}
$$

Equilibrium:

$$\phi_{eq} = U_{eq} = 0$$

$$\psi_{eq} = J_{eq} = \cos x \quad , \quad F_{eq} = (1 + d_e^2) \cos x$$

Hall Discretization (provided by K. Germaschewski, CMRS)

```
typedef struct {
  double phi,psi,U,F;
} Field;

Function(DALocalInfo *info,Field **x,Field **f,…)
…..
   /* Compute over the local points */
   for (j=info->ys; j<info->ys+info->ym; j++) {
      for (i=info->xs; i<info->xs+info->xm; i++) {
```

Hall Discretization (cont.)

```
  /* Lap(phi) - U */
f[j][i].phi = (Lapl(x,phi,i,j) - x[j][i].U) * hxhy;
  /* psi - d_e^2 * Lap(psi) - F */
f[j][i].psi  =  (x[j][i].psi  -  de2  *  Lapl(x,psi,i,j)  -  x[j][i].F)  *
hxhy;
  /* vx * U_x + vy * U_y - (B_x * F_x + B_y * F_y) / d_e ....
*/
f[j][i].U  = ((vxp * (D_xm(x,U,i,j)) + vxm * (D_xp(x,U,i,j)) +
              vyp * (D_ym(x,U,i,j)) + vym * (D_yp(x,U,i,j))) -
              (Bxp * (D_xm(x,F,i,j) + F_eq_x) + Bxm * ....
 ....
```

Algorithm review

$$F(u) = 0, \quad \textbf{Jacobian } A(u)$$

Newton

$$u \leftarrow \bar{u} - A^{-1}(\bar{u})F(\bar{u})$$

Newton – SOR (1 inner sweep)

$$u_i \leftarrow \bar{u}_i - A_{ii}^{-1}(\bar{u})\{F_i(\bar{u}) - \sum_{j<i} A_{ij}(\bar{u})[\bar{u}_j - u_j]\}$$

SOR-Newton (1 inner sweep)

$$u_i \leftarrow \bar{u}_i - A_{ii}^{-1}(u)F_i(u)$$

Cute observation

SOR-Newton

$$u_i \leftarrow \bar{u}_i - A_{ii}^{-1}(u)F_i(u)$$

With approximations

$$A_{ii}(u) \approx A_{ii}(\bar{u})$$

$$F_i(u) \approx F_i(\bar{u}) + \sum A_{ij}(\bar{u})[u_j - \bar{u}_j]$$

Gives Newton-SOR

$$u_i \leftarrow \bar{u}_i - A_{ii}^{-1}(\bar{u})\{F_i(\bar{u}) - \sum_{j<i} A_{ij}(\bar{u})[\bar{u}_j - u_j]\}$$

$\Rightarrow$ **Matrix-free linear relaxation**

(Gauss-Seidel)

is almost identical to nonlinear relaxation

Function and Jacobian evaluation

- **FAS requires pointwise**
- **Newton desires global**
- **Newton-MG desires both**

Automatic Differentiation

- **Given code for** $F(u)$ **can compute**
  - $A(u)$ **and**
  - $A(u)*w$ **efficiently**
- **Given code for** $F_i(u)$ **can compute**
  - $A_{ii}(u)$ **and**
  - $\sum_j A_{ij}(u)w_j$ **efficiently**

Code generation (in-lining ☺)

- **Inside the small dimensional Newton methods is a user-provided function and (AD) Jacobian**
- **Big performance hit if handled directly with components**

Coarse grid correction is not an issue ☺

- **Newton-MG**

$$A_H(R\bar{u})c_H = RF(\bar{u})$$

$$u \leftarrow u - R^T c_H$$

- **MG-Newton**

$$F_H(R\bar{u} + c_H) - F_H(R\bar{u}) + RF(\bar{u}) = 0$$

Conclusion

The algorithmic/mathematical building blocks for Newton-MG and MG-Newton are essentially the same

Thus the software building blocks should be also (and they will be ☺).