Newton-Krylov Methods

Mech 511

Learning Objectives

- Explain conceptually how Krylov methods aim to produce to a solution to a linear system.
- Describe how the Krylov subspace is computed.
- Describe how GMRES uses the Krylov subspace to compute an approximate solution to the linear system.
- Explain why preconditioning is needed for GMRES and how to apply it.
- Understand why Newton-like methods are an efficient way to reach steady state for CFD problems.
- Describe how to use line search methods to improve the robustness of a Newton-Krylov solver.
- Describe techniques that are useful for accelerating the steady-state convergence of a Newton-Krylov solver.

1 Introduction

CFD codes operating in steady-state mode have as their goal the most efficient possible calculation of the steady-state solution to a non-linear system of algebraic equations:

$$R(U) = 0 \tag{1}$$

where U is the solution in all control volumes, and R is the (non-linear) residual. In principle, we could solve Eq. 1 using Newton's method, but we require a good starting guess to get the problem to converge. In practice, therefore, we solve the unsteady problem instead:

$$\frac{dU}{dt} = R\left(U\right)$$

and discretize this in time using implicit time advance, to get:

$$\left(\frac{I}{\Delta t} + \frac{\partial R}{\partial U}\right)\delta U = R\left(U^n\right) \tag{2}$$

We solve Eq. 1 by integrating Eq. 2 to large time. This is completely standard stuff from a first course in CFD.

If we can use a sufficiently large time step Δt, then Eq. 2 approaches Newton's method; this is a good thing, because we know that Newton's method converges quadratically. To make this all work, we need two things: a robust, efficient solver for the system of linear equations we have to solve at every time step to get δU from Eq. 2; and a scheme for deciding how to increase Δt as we go along without having the whole solver break down.

2 Linear System Solution

The linear system that we're trying to solve, $A\boldsymbol{x} = \boldsymbol{b}$, can be massive. In computational aerodynamics, for instance, it's currently[1] normal to see computations with more than 1 billion (10^9!) equations, especially for high-lift configurations. These systems are far too large to solve by direct inversion. For structured meshes, alternating direction approximate factorization schemes are reasonably effective. For unstructured meshes, on the other hand, approximate factorization into terms associated with the coordinate directions fails, and purely numerical factorization schemes like incomplete lower-upper (ILU) factorization don't typically work well at this large scale.

Krylov subspace methods work by finding the point in a small dimensional subspace (30-500 dimensions) of that massive 10^9-dimensional space that is the closest to the true solution. Essentially, we trade the difficult problem of solving the massive linear system exactly for the much easier problem of finding the linear combination of a small number of vectors that comes closest to solving that massive linear system.

[1] As of 2020.

Now, if you pick that subspace randomly (by choosing the same number of vectors at random), you'll get a lousy answer. Yet Krylov subspace methods — including the conjugate gradient (CG) method, the generalized minimal residual method (GMRES), and a number of others — work, and work well. The question is, how?

2.1 Computing a Krylov Subspace

For our subspace to include a good approximation to the solution of the linear system, we need the vectors to include information about the matrix. To do this, we start with a series of products of some starting vector times the matrix. That is, for some vector $\boldsymbol{v}$ and matrix A, we compute $A\boldsymbol{v}$, $A^2\boldsymbol{v}$, $A^3\boldsymbol{v}$, and so on, by repeated matrix vector multiplication. These vectors aren't, in general orthogonal to each other[2], and we'd like them to be, because that will make it easier to find the right linear combination of these vectors later on. So we'll compute a *span* of ($\boldsymbol{v}$, $A\boldsymbol{v}$, $A^2\boldsymbol{v}$, $A^3\boldsymbol{v}$, ...), which will give us a collection of orthogonal unit vectors that are all linear combinations of these repeated products. In fact, we'll orthogonalize after every matrix-vector product. The vectors we produce this way still span the same vector space, but less orthogonalization error.

Along the way, we'll accumulate information about the dot products of the vectors we produce in a rectangular matrix H. This is called the *Arnoldi process.* See Algorithm 1; note that in this and all other algorithms in this document, "=" is used in the computer science sense of assignment, not in the mathematical sense. In addition to computing the matrix-vector products required to build the subspace, this algorithm also performs the *modified Gram-Schmidt* orthogonalization, and sets up an $(m+1) \times m$ matrix containing the dot products computed during the orthogonalization. This matrix, we'll show later, is the one we need for that small matrix problem.

Notice that

$$\begin{aligned} h_{i,j} &= (A\boldsymbol{v}_j) \cdot \boldsymbol{v}_i \\ &= \boldsymbol{v}_i^T A \boldsymbol{v}_j \end{aligned}$$

Therefore, after step k, we can write the matrix H as

$$H_k = V_{k+1}^T A V_k. \tag{3}$$

[2]In fact, this series of products will eventually converge on the eigenvector of A that has the eigenvalue with largest magnitude. There's a whole family of eigensolver techniques that exploit this behavior.

Algorithm 1 Arnoldi Process for computing a Krylov vector subspace.

Given: a matrix A, a vector $\boldsymbol{v}_1$, and a subspace size m.
for j = 1,...,m do
  compute $\boldsymbol{w} = A\boldsymbol{v}_j$
  for (i = 1,...,j) do
    $h_{i,j} = \boldsymbol{w} \cdot \boldsymbol{v}_i$
    $\boldsymbol{w} = \boldsymbol{w} - h_{i,j}\boldsymbol{v}_i$
  end for
  $h_{j+1,j} = \|\boldsymbol{w}\|_2$
  $\boldsymbol{v}_{j+1} = \frac{1}{h_{j+1,j}}\boldsymbol{w}$
end for

Here, V_k is a rectangular $(N \times k)$ matrix whose columns are the first k vectors from the Arnoldi process; by construction, these vectors are orthonormal. H_k is a rectangular $((k+1) \times k)$ matrix whose non-zero entries are the $h_{i,j}$ computed in the Arnoldi process. The fill in this matrix looks like:

$$\begin{bmatrix} x & x & x \\ x & x & x \\ 0 & x & x \\ 0 & 0 & x \end{bmatrix};$$

this is what's called an *upper Hessenberg* matrix.

Because the vectors $\boldsymbol{v}_j$ are orthonormal, we can multiply Equation 3 from the left by V_{k+1} and get

$$AV_k = V_{k+1}H_k.$$

2.2 The Generalized Minimum Residual Method

The Arnoldi process is use in several different Krylov subspace algorithms for solving linear systems and for eigenvalue problems. In CFD, the most commonly used of these methods is the generalized minimum residual method (GMRES).

2.2.1 Getting Started

GMRES aims to solve the linear system

$$A\boldsymbol{x} = \boldsymbol{b}$$

with some initial guess $\boldsymbol{x}_0$, which is often the zero vector. This implies an initial residual for the linear system

$$\boldsymbol{r}_0 \equiv \boldsymbol{b} - A\boldsymbol{x}_0.$$

GMRES defines

$$\beta \equiv \|\boldsymbol{r}_0\|_2$$

and sets $\boldsymbol{v}_1 = \frac{1}{\beta}\boldsymbol{r}_0$.

2.2.2 Approximately Solving the Linear System

With that starting point, GMRES runs through the Arnoldi process, building a vector subspace K_k spanned by the vectors V_k, and the Hessenberg matrix H_k. The goal is to choose a solution update $\boldsymbol{u}$ to minimize the error in solving the linear system. In math,

$$\min_{\boldsymbol{u}\epsilon K_k} \|\boldsymbol{b} - A\left[\boldsymbol{x}_0 + \boldsymbol{u}\right]\|$$

Now we can replace $\boldsymbol{r}_0 = \beta\boldsymbol{v}_1$ and $\boldsymbol{u} = V_k\boldsymbol{y}$. Notice that here we're starting towards the lower-dimensional problem, because $\boldsymbol{y}$ is the vector of length k that tells the weights of the linear combination of the k vectors V_k. This gives us

$$\begin{aligned}\min_{\boldsymbol{z}\epsilon K_k} \|\boldsymbol{r}_0 - A\boldsymbol{u}\| &= \min_{\boldsymbol{z}\epsilon K_k} \|\beta\boldsymbol{v}_1 - AV_k\boldsymbol{y}\| \\ &= \min_{\boldsymbol{z}\epsilon K_k} \|V_{k+1}\left(\beta\boldsymbol{e}_1 - H_k\boldsymbol{y}\right)\|\end{aligned}$$

where $\boldsymbol{e}_1$ is a vector of length $k+1$ whose only non-zero entry is a one in the first position. Finally, because V_{k+1} has orthonormal columns, we can write [3]

$$\begin{aligned}\min_{\boldsymbol{z}\epsilon K_k} \|\boldsymbol{r}_0 - A\boldsymbol{u}\| &= \min_{\boldsymbol{z}\epsilon K_k} \|V_{k+1}\left(\beta\boldsymbol{e}_1 - H_k\boldsymbol{y}\right)\| \\ &= \min_{\boldsymbol{z}\epsilon K_k} \left\{\left(\beta\boldsymbol{e}_1 - H_k\boldsymbol{y}\right)^T \cancelto{I}{\left(V_{k+1}^T V_{k+1}\right)}\left(\beta\boldsymbol{e}_1 - H_k\boldsymbol{y}\right)\right\} \\ &= \min_{\boldsymbol{z}\epsilon K_k} \|\left(\beta\boldsymbol{e}_1 - H_k\boldsymbol{y}\right)\| \qquad (4)\end{aligned}$$

This result is critically important: we just showed that we can find the vector $\boldsymbol{u}$ in the Krylov subspace that gives us the smallest error in solving the original $N \times N$

[3] To see this, remember that the norm of that vector (call it $\boldsymbol{p}$) can be written as $\boldsymbol{p}^T\boldsymbol{p}$, which contains in the middle $V_{k+1}^T V_{k+1}$, which is a $(k+1) \times (k+1)$ identity matrix.

linear system (that's the left-hand side) by solving a $(k+1)\times k$ least-squares problem on the right, which will tell us the weights in the linear combination

$$\boldsymbol{u} = \sum_{j=1}^{k} \boldsymbol{v}_k y_k$$

What's more, as we'll see, the optimal way to solve that least-squares problem will automatically give us information about the final residual norm $\|\boldsymbol{b} - A(\boldsymbol{x}_0 + \boldsymbol{u})\|$. That's actually fairly remarkable.

The right-hand side of Eq. 4 is a least squares problem, with $k+1$ equations in k unknowns. Specifically, we're trying to solve a system that looks like this (for $k = 4$):

$$\begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} \\ h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} \\ & h_{3,2} & h_{3,3} & h_{3,4} \\ & & h_{4,3} & h_{4,4} \\ & & & h_{5,4} \end{bmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} \beta \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{5}$$

Because the matrix H is upper Hessenberg[4], this is a particularly easy least-squares problem to solve.

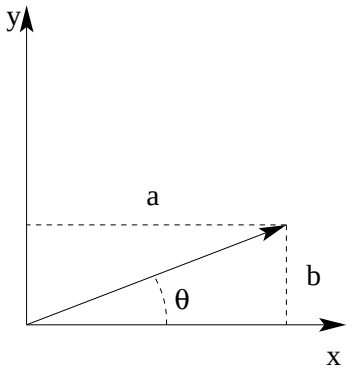

Figure 1: Givens rotation

Consider a vector $\begin{pmatrix} a & b \end{pmatrix}^T$, as shown in Figure 1. We can rotate this vector by an angle $-\theta$ so that the result is $\begin{pmatrix} \sqrt{a^2+b^2} & 0 \end{pmatrix}^T$:

$$\begin{aligned} \begin{pmatrix} \sqrt{a^2+b^2} \\ 0 \end{pmatrix} &= \frac{1}{\sqrt{a^2+b^2}} \begin{bmatrix} a & b \\ -b & a \end{bmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \\ &= \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \end{aligned}$$

where $\theta = \arctan(b/a)$. How does this help? Because if we do this to the first column of H, we eliminate the entry below the main diagonal. To apply this to Eq. 5, we

[4]That is, all its entries $h_{i,j}$ with $i \geq j+2$ are zero.

need to have additional rows added to our rotation so that we leave the third and subsequent rows alone. That is, we need to left multiply both sides of Eq. 5 by

$$\begin{bmatrix} \cos\theta & \sin\theta & & & \\ -\sin\theta & \cos\theta & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} \\ h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} \\ & h_{3,2} & h_{3,3} & h_{3,4} \\ & & h_{4,3} & h_{4,4} \\ & & & h_{5,4} \end{bmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & & & \\ -\sin\theta & \cos\theta & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \begin{pmatrix} \beta \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{6}$$

to get something that looks like this:

$$\begin{bmatrix} h'_{1,1} & h'_{1,2} & h'_{1,3} & h'_{1,4} \\ 0 & h'_{2,2} & h'_{2,3} & h'_{2,4} \\ & h_{3,2} & h_{3,3} & h_{3,4} \\ & & h_{4,3} & h_{4,4} \\ & & & h_{5,4} \end{bmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} \beta\cos\theta \\ -\beta\sin\theta \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

where the $'$ markings are there to remind us that these values have all been updated as we did the matrix-matrix multiplication on the LHS of Eq. 6.

Now we can treat the last k rows and $k-1$ columns as a problem of the same form, but smaller, and repeat the process.

After applying k rotations, we end up with a system that looks like this:

$$\begin{bmatrix} h'_{1,1} & h'_{1,2} & h'_{1,3} & h'_{1,4} \\ 0 & h'_{2,2} & h'_{2,3} & h'_{2,4} \\ & 0 & h'_{3,3} & h'_{3,4} \\ & & 0 & h'_{4,4} \\ & & & 0 \end{bmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \rho_4 \\ \rho_5 \end{pmatrix}$$

We can't do anything about the last row, but we can certainly solve the upper-triangular system in the first k rows for the y_i by back-substitution. Also, note that at each step, we're multiplying the residual on the RHS (originally β) by a quantity strictly less than 1 in magnitude to populate the next row. This implies that GMRES will improve the approximation to the solution of the original linear system with each iteration. The final residual for the least-squares problem (in this case ρ_5) is the same as the residual for the big linear system. [5]

Once we solve this for $\boldsymbol{y}$, then the solution update $\boldsymbol{u}$ is just $V_k\boldsymbol{y}$, a linear combination of our basis vectors.

[5] To make matters even better, we can (at the cost of a very small amount of memory, to store each of the Givens rotations) do this process incrementally, so that we track the residual as we build the subspace (without even knowing the y_j's yet), and stop GMRES when that residual is small enough. That's not the way the algorithm is laid out above, but it's a small variation.

2.2.3 Overall Algorithm

The overall algorithm is summarized in Algorithm 2. One additional wrinkle in this version is the possibility of a restart. In principle, we can run GMRES forever, but the cost is quadratic: at step j, we have to compute j dot products. Combined with the numerical difficulties of really having the $\boldsymbol{v}_j$ stay orthogonal to each other and memory usage limits, we typically limit subspace sizes to about 100. If the error in the solution is still too large, we restart GMRES from the updated solution. This amounts to doing a second GMRES solve with a different starting guess for the solution. There are senior ninja techniques for carrying over part of the subspace from one restart to the next, which helps convergence.

Algorithm 2 GMRES with restart

Start: Choose $\boldsymbol{x}_0$ and compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$, and $\boldsymbol{v}_1 = \boldsymbol{r}_0 / \|\boldsymbol{r}_0\|$
Iterate: For $j = 1, 2, \ldots, m$ do
    $\boldsymbol{w}_j = A\boldsymbol{v}_j$
    $h_{i,j} = \boldsymbol{w}_j \cdot \boldsymbol{v}_i, \quad i = 1, 2, \ldots, j$
    $\hat{\boldsymbol{v}}_{j+1} = \boldsymbol{w}_j - \sum_{i=1}^{j} h_{i,j}\boldsymbol{v}_i$
    $h_{j+1,j} = \|\hat{\boldsymbol{v}}_{j+1}\|$
    $\boldsymbol{v}_{j+1} = \hat{\boldsymbol{v}}_{j+1}/h_{j+1,j}$
end for
Update solution: $\boldsymbol{x}_m = \boldsymbol{x}_0 + \sum_{i=1}^{m} y_i\boldsymbol{v}_i$ where the y_i solve Eq. 5
Restart: If $\boldsymbol{r}_m = \boldsymbol{b} - A\boldsymbol{x}_m$ is small enough, stop. Otherwise, restart with $\boldsymbol{x}_0$ replaced by $\boldsymbol{x}_m$.

2.2.4 Convergence Behavior

The solution update $\boldsymbol{u}$ is a linear combination of basis vectors for the Krylov space, each of which can be written a polynomial in A multiplied by the starting vector $\mathbf{v}_1$. In general, then, we can write the update as

$$\boldsymbol{u} = P(A)\,\boldsymbol{v}_1$$

where $P(A)$ is a polynomial of degree k. In the simplest possible terms, GMRES converges well if there is a polynomial P so that $P(\lambda_i)$ is small for all λ_i. Saad and Schultz [5] contains a much more detailed description and bounds on the convergence rate based on the eigenvalue spectrum of A.

2.3 Preconditioning

GMRES often converges abysmally without preconditioning, because the eigenvalues aren't clustered well enough. With (right) preconditioning, we solve:

$$AP^{-1}P\boldsymbol{x} = \boldsymbol{b}$$
$$\text{or} \quad AP^{-1}\boldsymbol{\xi} = \boldsymbol{b}$$

with $P\boldsymbol{x} = \boldsymbol{\xi}$. So every iteration, in place of just multiplying $A\boldsymbol{v}$, we have to be able to solve efficiently a linear system $P\boldsymbol{z} = \boldsymbol{v}$, after which we do much the same things for the rest of the iteration. At the end, we get $\boldsymbol{\xi} = V_k\boldsymbol{y}$, and we have to apply the preconditioner again to get $\boldsymbol{x} = P^{-1}\boldsymbol{\xi}$. See Saad [4] for more details.

Our requirements for P is that P must be a reasonable approximation to A, and it must be easy to invert. Factorizations of various sorts are often used, as are matrices derived from similar physics but simpler discretizations (for instance, a first-order matrix for a second-order problem, or a linearization of a cheap flux function for a problem with an expensive flux).

2.4 An Example

Let's take a small system, at 10×10:

$$A = \begin{bmatrix} -2 & 1 & & & & & & & & \\ 1 & -2 & 1 & & & & & & & \\ & 1 & -2 & 1 & & & & & & \\ & & 1 & -2 & 1 & & & & & \\ & & & 1 & -2 & 1 & & & & \\ & & & & 1 & -2 & 1 & & & \\ & & & & & 1 & -2 & 1 & & \\ & & & & & & 1 & -2 & 1 & \\ & & & & & & & 1 & -2 & 1 \\ & & & & & & & & 1 & -2 \end{bmatrix}$$

$$\boldsymbol{b} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 5 & 1 & 0 & 0 & 0 \end{pmatrix}^T$$

We can solve this exactly, and find:

$$\boldsymbol{x} = -\frac{7}{11}\begin{pmatrix} 5 & 10 & 15 & 20 & 25 & \frac{199}{7} & 24 & 18 & 12 & 6 \end{pmatrix}^T$$

Let's apply GMRES to this, and see how it goes. Our initial guess is $\boldsymbol{x}_0 = 0$.

2.4.1 Without Preconditioning

Before we start, we have $\boldsymbol{r}_0 = \boldsymbol{b}$, so $\|\boldsymbol{r}_0\| = 3\sqrt{3}$, and

$$\boldsymbol{v}_1 = \frac{\sqrt{3}}{9}\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 5 & 1 & 0 & 0 & 0 \end{pmatrix}^T$$

One-dimensional subspace For a one-dimensional sub-space, we do one loop of the Arnoldi process. First, we compute

$$\begin{aligned} \mathbf{w}_2 = A\boldsymbol{v}_1 &= \frac{\sqrt{3}}{9}\begin{pmatrix} 0 & 0 & 0 & 1 & 3 & -8 & 3 & 1 & 0 & 0 \end{pmatrix}^T \\ h_{1,1} &= \mathbf{w}_2 \cdot (A\mathbf{v}_1) \\ \hat{\mathbf{v}}_2 &= A\boldsymbol{v}_1 - h_{1,1}\boldsymbol{v}_1 \end{aligned}$$

$$H = \begin{bmatrix} h_{1,1} \\ \\ h_{2,1} \end{bmatrix} = \frac{1}{27}\begin{bmatrix} -34 \\ \\ 2\sqrt{278} \end{bmatrix}$$

$$\boldsymbol{v}_2 = \frac{A\boldsymbol{v}_1 - h_{1,1}\boldsymbol{v}_1}{h_{2,1}} = \frac{\sqrt{3}\sqrt{278}}{5004}\begin{pmatrix} 0 & 0 & 0 & 27 & 115 & -46 & 115 & 27 & 0 & 0 \end{pmatrix}^T$$

At this point, the least-squares problem we have to solve is

$$\frac{1}{27}\begin{bmatrix} -34 \\ 2\sqrt{278} \end{bmatrix}(y_1) = \begin{pmatrix} 3\sqrt{3} \\ 0 \end{pmatrix}$$

$$y_1 = -\frac{17}{14}\sqrt{3}$$

so

$$\boldsymbol{u} = \frac{-17}{42}\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 5 & 1 & 0 & 0 & 0 \end{pmatrix}^T$$

and

$$\boldsymbol{r}_1 = \frac{1}{42}\begin{pmatrix} 0 & 0 & 0 & 17 & 93 & 74 & 93 & 17 & 0 & 0 \end{pmatrix}^T$$

$$\|\boldsymbol{r}_1\| = \frac{\sqrt{5838}}{21} \approx 3.638$$

We could also have gotten this value without calculating $\mathbf{u}$ or $\mathbf{r}_1$ if we'd used a Givens rotation on the least-squares problem (I used the normal equations).

So the solution is in fact better after one iteration. It's easy to verify that this is the optimal value of y_1; that is, if you make y_1 slightly larger or smaller, $\|\boldsymbol{r}_1\|$ increases.

Two-dimensional subspace For a two-dimensional sub-space, we do another loop of the Arnoldi process. First, we compute

$$A\boldsymbol{v}_2 = \frac{\sqrt{3}\sqrt{278}}{5004}\begin{pmatrix} 0 & 0 & 27 & 61 & -249 & 322 & -249 & 61 & 27 & 0 \end{pmatrix}^T$$

Then we fill in the values in H:

$$H = \begin{bmatrix} -\frac{34}{27} & \frac{2\sqrt{278}}{27} \\ \frac{2\sqrt{278}}{27} & -\frac{17197}{7506} \\ 0 & \frac{3\sqrt{2373}}{139} \end{bmatrix}$$

and compute $\boldsymbol{v}_3$.

$$\boldsymbol{v}_3 = \frac{\sqrt{278}\sqrt{791}}{5879592}\begin{pmatrix} 0 & 0 & 278 & 1265 & -275 & 110 & -275 & 1265 & 278 & 0 \end{pmatrix}^T$$

At this point, the least-squares problem we have to solve is

$$\begin{bmatrix} -\frac{34}{27} & \frac{2\sqrt{278}}{27} \\ \frac{2\sqrt{278}}{27} & -\frac{17197}{7506} \\ 0 & \frac{3\sqrt{2373}}{139} \end{bmatrix}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 3\sqrt{3} \\ 0 \\ 0 \end{pmatrix}$$

$$\text{Solution:} \qquad \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -\frac{\sqrt{3}}{70}\begin{pmatrix} 179 \\ 4\sqrt{278} \end{pmatrix}$$

so

$$\boldsymbol{u} = \frac{-1}{210}\begin{pmatrix} 0 & 0 & 0 & 54 & 409 & 803 & 409 & 54 & 0 & 0 \end{pmatrix}^T$$

and

$$\|\boldsymbol{r}_2\| = \frac{2\sqrt{23730}}{105} \approx 2.934$$

Again, the solution is better, although it isn't improving very fast. Note that there's only one $\boldsymbol{u}$ for the whole solve, not one for each iteration.

2.4.2 With Preconditioning

In this case, it happens that if you take these two matrices:

$$L = \begin{bmatrix} 1 & & & & & & & & & \\ -1 & 1 & & & & & & & & \\ & -1 & 1 & & & & & & & \\ & & -1 & 1 & & & & & & \\ & & & -1 & 1 & & & & & \\ & & & & -1 & 1 & & & & \\ & & & & & -1 & 1 & & & \\ & & & & & & -1 & 1 & & \\ & & & & & & & -1 & 1 & \\ & & & & & & & & -1 & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} -1 & 1 & & & & & & & & \\ & -1 & 1 & & & & & & & \\ & & -1 & 1 & & & & & & \\ & & & -1 & 1 & & & & & \\ & & & & -1 & 1 & & & & \\ & & & & & -1 & 1 & & & \\ & & & & & & -1 & 1 & & \\ & & & & & & & -1 & 1 & \\ & & & & & & & & -1 & 1 \\ & & & & & & & & & -1 \end{bmatrix}$$

then

$$LR = \begin{bmatrix} -1 & 1 & & & & & & & & \\ 1 & -2 & 1 & & & & & & & \\ & 1 & -2 & 1 & & & & & & \\ & & 1 & -2 & 1 & & & & & \\ & & & 1 & -2 & 1 & & & & \\ & & & & 1 & -2 & 1 & & & \\ & & & & & 1 & -2 & 1 & & \\ & & & & & & 1 & -2 & 1 & \\ & & & & & & & 1 & -2 & 1 \\ & & & & & & & & 1 & -2 \end{bmatrix}$$

which is almost identical to A. Also, inverting L and R (more precisely, solving $L\boldsymbol{p} = \boldsymbol{q}$ and $R\boldsymbol{p} = \boldsymbol{q}$ is almost trivial. So LR should be a great preconditioner for

this problem. In fact, Maple is happy to compute AP^{-1} for this case:

$$A\left(LR\right)^{-1} = \begin{bmatrix} 11 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ & 1 & & & & & & & & \\ & & 1 & & & & & & & \\ & & & 1 & & & & & & \\ & & & & 1 & & & & & \\ & & & & & 1 & & & & \\ & & & & & & 1 & & & \\ & & & & & & & 1 & & \\ & & & & & & & & 1 & \\ & & & & & & & & & 1 \end{bmatrix}$$

Physically, L and R are both one-sided first derivative approximations (with no boundary conditions), and A is a Laplacian approximation (again with no boundary conditions). So it's not that surprising that $A \approx LR$.

We'll start with the same initial data.

One Dimensional Subspace We have the same $\boldsymbol{v}_1$ as before, but now we calculate

$$\begin{aligned} \boldsymbol{z}_1 &= \left(LR\right)^{-1} \boldsymbol{v}_1 \\ &= -\frac{7\sqrt{3}}{9} \begin{pmatrix} 5 & 5 & 5 & 5 & 5 & \frac{34}{7} & 4 & 3 & 2 & 1 \end{pmatrix}^T \end{aligned}$$

$$H = \begin{bmatrix} 1 \\ \frac{35\sqrt{3}}{9} \end{bmatrix}$$

$$\begin{aligned} \boldsymbol{w}_2 &= A\boldsymbol{z}_1 - H_{1,1}\boldsymbol{v}_1 \\ &= \frac{35}{9}\sqrt{3} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^T \\ \boldsymbol{v}_2 &= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^T \end{aligned}$$

Solving, we get $y_1 = \frac{81\sqrt{3}}{1252}$ and

$$\boldsymbol{\xi}_1 = \frac{27}{1252} \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 5 & 1 & 0 & 0 & 0 \end{pmatrix}^T$$

$$\boldsymbol{u} = -\frac{189}{1252} \begin{pmatrix} 5 & 5 & 5 & 5 & 5 & \frac{34}{7} & 4 & 3 & 2 & 1 \end{pmatrix}^T$$

It's no surprise that this is a multiple of $\boldsymbol{z}_1$, since both start out with $(LR)^{-1}\boldsymbol{v}_1$. The norm of the residual is $\frac{105}{626}\sqrt{939} \approx 5.1398$. This is a terrible start, but wait for it!

Two Dimensional Subspace When we add another vector, we get:

$$\begin{aligned}
\boldsymbol{z}_2 &= (LR)^{-1}\boldsymbol{v}_2 \\
&= -\begin{pmatrix} 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix}^T \\
A\boldsymbol{z}_2 &= \begin{pmatrix} 11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^T \\
H &= \begin{bmatrix} 1 & 0 \\ \frac{35\sqrt{3}}{9} & 11 \\ 0 & 0 \end{bmatrix} \\
\boldsymbol{v}_3 &= \mathbf{0}
\end{aligned}$$

This is the "happy breakdown" case: we can't produce another entry in the subspace, and we have a square system instead of a rectangular one. When we solve, we get

$$y_1 = 3\sqrt{3} \qquad y_2 = -\frac{35}{11}$$

This gives us

$$\boldsymbol{\xi}_2 = \begin{pmatrix} -\frac{35}{11} & 0 & 0 & 0 & 1 & 5 & 1 & 0 & 0 & 0 \end{pmatrix}^T$$

and

$$\boldsymbol{u} = \boldsymbol{x}_{exact}$$

Comments This was a particularly special case, because $A(LR)^{-1}$ has exactly two eigenvalues. This means that a second-degree polynomial in the eigenvalues is enough to get $P(\lambda) = 0$ at both eigenvalues, and so only two members are required in the subspace. This shows, for an extreme, contrived case, the potential power of a good preconditioner.

3 Quasi-Newton Methods for the Non-linear Problem

Efficient solution of the linear system is necessary but not sufficient to efficiently solve a non-linear problem like a steady-state flow problem. For that, we can use a

Newton-like method, though we have to be careful to make it robust enough that we will actually get convergence.

There are four components that we can combine to make this process efficient and robust for moderately complex flows.

1. Line search. The solution update that comes out of the linear solver may, in some circumstances lead to an aphysical solution state, or "just" to a solution that has a larger residual than the one from the previous time step. A line search allows us to avoid this by taking only part of that update when necessary.

2. Time step modification. We want to be running as close as we can to a full Newton scheme for the non-linear steady-state problem, but we can't start out there. We start with an implicit Euler time advance with a modest CFL number, and then increase the CFL number as we go. That increase can be tied to the status of the line search.

3. Local time-stepping. Most steady-state simulations operate using local time-stepping: a different time step for each control volume. Since we don't care about time accuracy, a scheme that allows large time steps where that's feasible should help convergence, and it does.

4. A safe state. It's possible that we can get a solution at some point in the convergence process that satisfied our line search criteria, but from which we can't make further progress towards the solution. The details of why this happens aren't well understood, but the solution is: keep an old solution to fall back to.

3.1 Line Search and CFL Modification

At each time step, we solve a problem of the form

$$\left(\frac{I}{\Delta t} + \frac{\partial R}{\partial U}\right)\left(\overline{U}^{n+1} - \overline{U}^{n}\right) = -R\left(\overline{U}^{n}\right)$$

to get δU. This is actually a linearized version of the non-linear time advance problem:

$$\frac{I}{\Delta t}\left(\overline{U}^{n+1} - \overline{U}^{n}\right) = -R\left(\overline{U}^{n+1}\right)$$

We can define an *unsteady residual* as:

$$\widetilde{R}\left(\overline{U}^{n+1}\right) \equiv R\left(\overline{U}^{n+1}\right) + \frac{I}{\Delta t}\left(\overline{U}^{n+1} - \overline{U}^{n}\right)$$

When we're at steady state, this quantity is zero (both $R(U)$ and δU are zero).

Our goal during time advance is to reduce the norm of $\widetilde{R}$. That is, we seek to minimize

$$z\left(\overline{U}\right) = \frac{1}{2}\left\|\widetilde{R}\left(\overline{U}\right)\right\|_2^2$$

From optimization theory, we know that we can guarantee convergence of this optimization problem if we satisfy the Armijo sufficient decrease condition [1]:

$$\begin{aligned} z\left(\overline{U}^{n+1}\right) &\leq z\left(\overline{U}^{n}\right) + c_1 \alpha \nabla z^T \delta\overline{U} \\ &\leq z\left(\overline{U}^{n}\right) + c_1 \alpha \widetilde{R}\left(\overline{U}^{n}\right)^T \frac{\partial \widetilde{R}}{\partial \overline{U}}_n \delta\overline{U} \end{aligned} \tag{7}$$

where the constant c_1 can be taken to be very small; 10^{-4} is a common choice.

Because our original search direction $\delta\overline{U}$ has been chosen to decrease the residual, we should be able to find a value of α that reflects that decrease. We start with $\alpha = 1$. While we could search for the value of α that decreases $\widetilde{R}$ the most, this isn't necessary, nor even efficient. Instead, we check for sufficient decrease. If we don't meet that criterion, we decrease α (easy choice: multiply by 1/2 each time) and check again, recomputing $\widetilde{R}$ to evaluate the LHS of Ineq. 7. Then we can update the solution using

$$\overline{U}^{n+1} = \overline{U}^{n} + \alpha\,\delta\overline{U}$$

Finally, we can update the CFL number using [2, 3]:

$$CFL^{n+1} = \begin{cases} \beta \cdot CFL^n & \alpha = 1 \\ CFL^n & \alpha_{\min} < \alpha < 1 \\ \kappa \cdot CFL^n & \alpha < \alpha_{\min} \end{cases}$$

In our research code, we use $\beta = 1.5$, $\kappa = 0.1$, and $\alpha_{\min} = 0.01$.[6] We start with an initial CFL that's pretty small (0.01 for turbulent flows; 1 for inviscid flows; in between for laminar flows).

[6] Actually, in recent work, we're even more aggressive than that, but that scheme has a lot of moving parts to get working together properly.

3.2 Local Time Stepping

Up until now, we've talked about using a single time step for all the cells in the mesh, and picking that time step to be the largest that we can use for stability. This limits our time step based on the size of the smallest cell and/or the maximum velocity in the flow.

Local time-stepping schemes are based on the notion that we should run near the maximum stable time step in *each* cell, even though that means that the time step varies between cells. Clearly, this isn't time accurate. We don't care, when we're going for a steady-state solution. So we can pick, for instance

$$\Delta t_{i,j} = CFL_{\max} \frac{u_{i,j}}{\Delta x_{i,j}}$$

Here, the maximum CFL number is a global value, but the time step differs between cells. Now $\Delta t_{i,j}$ is one more field variable to carry around in your code, so that you can use the proper value each time you would have used Δt.

3.3 Safe State

In spite of careful checks at the line search stage, it's possible to get a solution state from which you can't continue to make progress towards a solution: line search backtracks you all the way to $\alpha \ll 1$, and reducing your global CFL number doesn't change this. At this point, you're stuck. The way out of this is to have a solution from one or more time steps before. When you detect that you can't progress from the state you're in, you go back to that previous stored state, and try again from there with a smaller time step.

References

[1] L. Armijo. Minimization of functions having lipschitz continuous first partial derivatives. *Pac. J. Math.*, 16:1–3, 1966.

[2] Marco Ceze and K. J. Fidkowski. Constrained pseudo-transient continuation. *International Journal for Numerical Methods in Engineering*, 2015.

[3] Alireza Jalali. *An adaptive higher-order unstructured finite volume solver for turbulent compressible flows.* Ph.D. thesis, The University of British Columbia, Department of Mechanical Engineering, 2017.

[4] Youcef Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal of Scientific Computing*, 14(2):461–469, 1993.

[5] Youcef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 7(3):856–869, July 1986.