Notes for 2015-02-18 – 2015-02-20

Tricky tridiagonals

Consider the tridiagonal matrix

$$T = \begin{bmatrix} a_1 & b_1 & & & & \\ c_2 & a_2 & b_2 & & & \\ & c_3 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & & c_n & a_n \end{bmatrix}$$

What happens if we factor using Gaussian elimination? We'll consider first the case of no pivoting. At the first step of elimination, we subtract a multiple of the first row from the second row, introducing a zero in the $(2,1)$ position. Then we subtract a multiple of the second row from the third row, and so on. At each step, we only work with two rows of the matrix. Let's write the first two steps in pictures, assuming an implementation where we are systematically overwriting the original matrix with lower and upper triangular factors. At each step, we color the entries of the matrix that are transformed. In the first step, we have

$$\begin{bmatrix} a_1 & b_1 & & & & \\ c_2 & a_2 & b_2 & & & \\ & c_3 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & & c_n & a_n \end{bmatrix} \rightarrow \begin{bmatrix} a_1 & b_1 & & & & \\ \mathbf{l_2} & \mathbf{\hat{a}_2} & b_2 & & & \\ & c_3 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & & c_n & a_n \end{bmatrix}$$

At the next step, we have

$$\begin{bmatrix} a_1 & b_1 & & & & \\ c_2 & a_2 & b_2 & & & \\ & c_3 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & & c_n & a_n \end{bmatrix} \rightarrow \begin{bmatrix} a_1 & b_1 & & & & \\ l_2 & \hat{a}_2 & b_2 & & & \\ & \mathbf{l_3} & \mathbf{\hat{a}_3} & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & & c_n & a_n \end{bmatrix}$$

At the end of the iteration, we have

$$\begin{bmatrix} a_1 & b_1 & & & \\ c_2 & a_2 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & c_n & a_n \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ l_2 & 1 & & & \\ & \ddots & \ddots & & \\ & & l_{n-1} & 1 & \\ & & & l_n & 1 \end{bmatrix} \begin{bmatrix} a_1 & b_1 & & & \\ & \hat{a}_2 & b_2 & & \\ & & \ddots & \ddots & \\ & & & \hat{a}_{n-1} & b_{n-1} \\ & & & & \hat{a}_n \end{bmatrix}$$

In code, we go from the standard Gaussian elimination loop

```
% Overwrite A with L and U
for j = 1:n-1
  A(j+1:n,j) = A(j+1:n,j)/A(j,j);
  A(j+1:n,j+1:n) = A(j+1:n,j+1:n) - A(j+1:n,j)*A(j,j+1:n);
end
```

to the simplified loop

```
% Overwrite T with L and U
for j = 1:n-1
  T(j+1,j) = T(j+1,j)/T(j,j);
  T(j+1,j+1) = T(j+1,j+1) - T(j+1,j)*T(j,j+1);
end
```

The former loop does $O(n^2)$ work per step, while the latter does $O(1)$ work per step; that is, factoring a tridiagonal costs $O(n)$ work per time, not the $O(n^3)$ cost of factoring an ordinary matrix.

Of course, since most of the entries in a tridiagonal are zero, we waste space by storing every entry explicitly, just as we waste time if we use the general-purpose Gaussian elimination procedure. In general, it is better to store only the nonzero entries of the matrix. For example, in the case of the tridiagonal matrix, we might store the a, b, and c coefficients as vectors, in which case Gaussian elimination looks like

```
% Overwrite a with the L entries, and b and c with the U entries
for j = 1:n-1
  a(j+1) = a(j+1)/b(j);
  b(j+1) = b(j+1)-a(j+1)*c(j);
end
```

We can similarly write $O(n)$ time routines for forward and backward substitution with the (bidiagonal) triangular factors of T.

Consider sparsity

Tridiagonal matrices (and more generally banded matrices) belong to a more general family of *sparse* matrices, or matrices in which the vast majority of the entries are zero.

Sparse matrices are a bit different from dense matrices. In a dense matrix, we store the elements of the matrix in a fixed order. There is no need to specify that a value at a specific location in memory corresponds to a particular matrix element; that information is implicit. For special sparse matrices, such as banded matrices, one can again come up with storage formats that have this property. But in a general sparse matrix format, though we only store the nonzero entries, we also have to store the location of those entries. This has several consequences. One is that memory access for sparse matrix computations is less regular than in the dense case, and so they run at a lower flop rate than dense computations (though they typically do far fewer flops than corresponding dense computations might, so it's still a win). Also, operations that are trivial in the dense case become more complicated in the sparse case. For example, adding two dense matrices is easy, but adding two sparse matrices can be rather complicated, depending on the data structure that is used to store them.

The one operation on sparse matrices that is really simple is matrix-vector multiplication. This costs $O(\text{nnz})$ time, where nnz is the number of nonzeros in the matrix. We will see that we can build iterative solvers that just use this operation, but that's a topic for another day. For now, the question is: can we do fast Gaussian elimination on sparse matrices? The answer is yes, but whether it's a good idea or not depends a lot on the matrix. We turn to why this is the case in the next section.

General sparse direct methods

Suppose A is a sparse matrix, and $PA = LU$. Will L and U also be sparse? The answer depends in a somewhat complicated way on the structure of the graph associated with the matrix A, the pivot order, and the order in which variables are eliminated. Except in very special circumstances, there will generally be more nonzeros in L and U than there are in A; these extra nonzeros are referred to as *fill*. There are two standard ideas for minimizing fill:

1. Apply a *fill-reducing ordering* to the variables; that is, use a factorization
$$PAQ = LU,$$
where Q is a column permutation chosen to approximately minimize the fill in L and U, and P is the row permutation used for stability.

   The problem of finding an elimination order that minimizes fill is NP-hard, so it is hard to say that any ordering strategy is really optimal. But there is canned software for some heuristic orderings that tend to work well in practice. From a practical perspective, then, the important thing is to remember that a fill-reducing elimination order tends to be critical to using sparse Gaussian elimination in practice.

2. Relax the standard partial pivoting condition, choosing the row permutation P to balance the desire for numerical stability against the desire to minimize fill.

For the rest of this lecture, we will consider the simplified case of *structurally* symmetric matrices and factorization without pivoting (which is stable for diagonally dominant systems and positive definite systems).

Sparse matrices, graphs, and tree elimination

Consider the following illustrative example of how factoring a sparse matrix can lead to more or less dense factors depending on the order of elimination. Putting in $\times$ to indicate a nonzero element, we have

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & & \\ \times & & & \times & \\ \times & & & & \times \end{bmatrix} = \begin{bmatrix} \times & & & & \\ \times & \times & & & \\ \times & \times & \times & & \\ \times & \times & \times & \times & \\ \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \end{bmatrix}.$$

That is, L and U have many more nonzeros than A. These nonzero locations that appear in L and U and not in A are called *fill-in*. On the other hand,

if we cyclically permute the rows and columns of A, we have

$$\begin{bmatrix} \times & & & & \times \\ & \times & & & \times \\ & & \times & & \times \\ & & & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} = \begin{bmatrix} \times & & & & \\ & \times & & & \\ & & \times & & \\ & & & \times & \\ \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times & & & & \times \\ & \times & & & \times \\ & & \times & & \times \\ & & & \times & \times \\ & & & & \times \end{bmatrix}.$$

That is, the factorization of PAP^T has *no* fill-in.

A sparse matrix A can be viewed as an *adjacency matrices* for an associated graphs: make one node for each row, and connect node i to node j if $A_{ij} \neq 0$. The graphs for the two "arrow" matrices above are:

These graphs of both our example matrices are *trees*, and they differ only in how the nodes are labeled. In the original matrix, the root node is assigned the first label; in the second matrix, the root node is labeled after all the children. Clearly, the latter label order is superior for Gaussian elimination. This turns out to be a general fact: if the graph for a (structurally symmetric) sparse matrix S is a tree, and if the labels are ordered so that each node appears after any children it may have, then there is no fill-in: that is, L and U have nonzeros only where S has nonzeros.

Why should we have no fill when factoring a matrix for a tree ordered from the leaves up? To answer this, we think about what happens in the first step of Gaussian elimination. Our original matrix has the form

$$S = \begin{bmatrix} \alpha & w^T \\ v & S_{22} \end{bmatrix}$$

The first row of U is identical to the first row of S, and the first column of L has the same nonzero structure as the first column of A, so we are fine there. The only question is about the nonzero structure of the Schur complement $S_{22} - vw^T/\alpha$. Note that the update vw^T/α has nonzeros only where v_i and w_j are both nonzero — that is, only when nodes i and j are both connected to node 1. But node 1 is a leaf node; the only thing it connects to is its

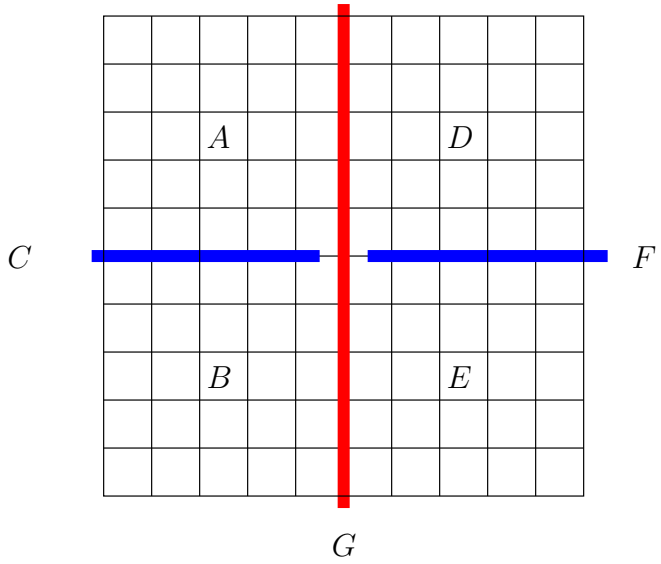
parent! So if p is the index of the parent of node 1 in the tree, then we only change the (p, p) entry of the trailing submatrix during the update — and we assume that entry is already nonzero. Thus, the graph associated with the Schur complement is the same as the graph of the original matrix, but with one leaf trimmed off.

Nested dissection

Tree-structured matrices are marvelous because we can do everything in $O(n)$ time: we process the tree from the leaves to the root in order to compute L and U, then recurse from the root to the leaves in order to do back substitution with U, and then go back from the leaves to the root in order to do forward substitution with L. Sadly, many of the graphs we encounter in practice do not look like trees. However, we can often profitably think of clustering nodes so that we get a *block* structure associated with a tree.

For illustrative purposes, let us consider Gaussian elimination on a matrix whose graph is a regular $n \times n$ mesh. Such a matrix might arise, for example, if we were solving Poisson's equation using a standard five-point stencil to discretize the Laplacian operator. We then think of cutting the mesh in half by removing a set of separator nodes, cutting the halves in half, and so forth. This yields a block structure of a tree consisting of a root (the separator nodes) and two children (the blocks on either side of the separator). We can now dissect each of the sub-blocks with a smaller separator, and continue on in this fashion until we have cut the mesh into blocks containing only a few nodes each. Figure 1 illustrates the first two steps in this process of *nested dissection*.

We can get a lower bound on the cost of the factorization by figuring out the cost of factoring the Schur complement associated with G, C, F, etc. After we eliminate everything except the nodes associated with G, we pay about $2n^3/3$ flops to factor the remaining (dense) n-by-n Schur complement matrix G. Similarly, we pay about $2(n/2)^3/3$ time to factor the dense $(n/2)$-by-$(n/2)$ complements associated with the separators C and F. Eliminating all four separators then costs a total of $\approx 10n^3/12$ flops. Now, think of applying nested dissection to blocks A, B, D, and E; eliminating the Shur complements associated with separators inside each of these blocks will take about $5(n/2)^3/6$ flops; all four together cost a total of $4(5(n/2)^3/6) = (1/2)(5n^3/6)$ flops to factor. If we keep recursing, we find that the cost of factoring Schur



$$S = \begin{bmatrix} S_{AA} & & S_{AC} & & & & S_{AG} \\ & S_{BB} & S_{BC} & & & & S_{BG} \\ S_{CA} & S_{CB} & S_{CC} & & & & S_{CG} \\ & & & S_{DD} & & S_{DF} & S_{DG} \\ & & & & S_{EE} & S_{EF} & S_{EG} \\ & & & S_{FD} & S_{FE} & S_{FF} & S_{FG} \\ S_{GA} & S_{GB} & S_{GC} & S_{GD} & S_{GE} & S_{GF} & S_{GG} \end{bmatrix}$$

Figure 1: Nested dissection on a square mesh. We first cut the graph in half with the red separator G, then further dissect the halves with the blue separators C and F. Nodes in A, B, D, and F are only connected through these separator nodes, which is reflected in the sparsity pattern of the adjacency matrix S when it is ordered so that separators appear after the things they separate.

complements associated with all the separators looks like

$$\frac{5}{6}n^3\left(1+\frac{1}{2}+\frac{1}{4}+\ldots\right) \approx \frac{5}{3}n^3.$$

It turns out that forming each Schur complement is asymptotically not more expensive than eliminating it, so that the overall cost of doing nested dissection on an $n \times n$ mesh with $N = n^2$ unknown is also $O(n^3) = O(N^{1.5})$. It also turns out that the fill-in is $O(N \log N)$[1].

Now think about doing the same thing with a three-dimensional mesh. In this case, the top-level separators for an $n \times n \times n$ mesh with $N = n^3$ unknowns would involve n^2 unknowns, and we would take $O(n^6) = O(N^2)$ time to do the elimination, and $O(N^{4/3})$ fill. This relatively poor scaling explains why sparse direct methods are attractive for solving 2D PDEs, but are less popular for 3D problems.

E pluribus unum

So far, we have described a few ideas about how to perform Gaussian elimination for linear systems. But you might find yourself asking a reasonable — if a bit lazy — question: "why bother?" After all, the MATLAB backslash operator is a marvelous invention, and the simple expression

```
x = A\b;
```

mostly "does the right thing" for a variety of types of matrices. If A is square and dense, this line causes MATLAB to factor the matrix A using Gaussian elimination, then carry out forward and backward substitution. If A is triangular, MATLAB detects that and just does forward or backward substitution without a factorization step. If A is a sparse matrix, MATLAB uses the UMFPACK sparse LU package, including applying a reasonable column permutation. If MATLAB can do all this automatically for you, why do you need to know the details?

There's a deeper answer to this question than the superficial "because you're taking a numerical methods class." It even goes beyond needing to

[1] The explanation of why is not so hard, at least for regular 2D meshes, but it requires more drawing than I feel like at the moment. The paper "Nested Dissection of a Regular Finite Element Mesh" by Alan George (SIAM J. Numer. Anal. 10(2), April 1973) gives a fairly readable explanation for the curious.

understand things like when a sparse system is best solved by a direct method vs. an iteration. One very important reason to understand the factorizations that are being computed behind the scenes is that those factorizations can be *reused* when you are solving more than one linear system at a go. And solving more than one problem at a time, as it turns out, is often what we want to do.

Multiple right hand sides

The simplest case of solving multiple problems is when the matrix is fixed, but there are several right hand sides. That is, we want to solve

$$Ax^{(k)} = b^{(k)}$$

for $k = 1, \ldots, m$. In the simple case where all the right hand sides are known in advance, we can still accomplish this by using the magic of MATLAB's backslash:

```
X = A\B;
```

But in some cases, we might not know the kth right hand side until we have learned the answer to the $k-1$th question. For example, suppose we wanted to run the iterative refinement process

$$x^{(k+1)} = x^{(k)} + \hat{A}^{-1}(b - Ax^{(k)})$$

that was mentioned in a previous lecture. In MATLAB, if we had already computed the factorization

```
[L,U,P] = lu(A);
```

we might run the iteration

```
x = U\(L\(P*b));
for k = 1:niter
  r = b-A*x;
  x = x + U\(L\(P*r));
end
```

Note that we *never* form the inverse of A, explicitly or implicitly. Rather, we apply A^{-1} to vectors through triangular solves involving the factors computed through Gaussian elimination. Using only triangular solves is good for

performance (we take $O(n^2)$ time per solve after the original factorization, rather than $O(n^3)$ time); and it is good for numerical stability.

The admonition against inverses sometimes causes a certain amount of confusion, and it bears repeating: *we want to only do permutations and triangular solves* applied to vectors. Specifically, in MATLAB, we have

```
% Generally good
x = U\(L\(P*b));

% Generally bad
x = inv(A)*b;     % Code that calls 'inv' deserves skepticism
x = U\(L\(P'\b)); % Silly, since P^{-%} = P
x = U\L\P*b;      % Order of operations means we form U\L!
x = (U\(L\P)*b;   % U\(L\P) is an explicit inverse
```

Multiple small-norm changes

Sometimes, we want to solve a sequence of linear systems of the form

$$(A + \Delta A)x = b$$

where $\|\Delta A\| \ll \|A\|$. One way to approach this problem is to linearize about A, i.e.

$$(A + \Delta A)^{-1} = A^{-1} - A^{-1}(\Delta A)A^{-1} + O(\|\Delta A\|^2).$$

Though it may not be obvious at the outset, you've seen this approach before! Consider what happens if we use A^{-1} as an approximation for $(A + \Delta A)^{-1}$ and apply one step of iterative refinement:

$$\begin{aligned}
x^{(0)} &= A^{-1}b \\
x^{(1)} &= x^{(0)} + A^{-1}\left(b - (A + \Delta A)x^{(0)}\right) \\
&= x^{(0)} + A^{-1}\left(b - (b + \Delta A)x^{(0)}\right) \\
&= x^{(0)} - A^{-1}\Delta A x^{(0)} \\
&= A^{-1}b - A^{-1}\Delta A b
\end{aligned}$$

We have already seen how to re-use the LU factorization for one or more steps of iterative refinement; no need to repeat here.

Multiple low-rank changes

Sometimes, we want to solve a sequence of linear systems of the form

$$(A + uv^T)x = b$$

where u and v are vectors. Suppose A and b remain fixed, but u and v change from problem. How do we solve this type of problem without having to do a new LU factorization for each choice of u and v?

There is a formula, the *Sherman-Morrison* formula, for the inverse of a matrix plus a rank-1 update. It's sometimes a useful thing to know about, and the derivation of the formula provides some nice examples of things we've described so far, like fast multiplication by low-rank matrices or Gaussian elimination on block matrices. The starting point for the argument is to think about how you would quickly *multiply* by the matrix $A + uv^T$. In MATLAB, you would write

```
y = A*x + u*(v'*x);
```

Note that this is *not* forming uv^T at any point! Equivalently, we might write

```
z = v'*x;
y = A*x + u*z;
```

That is, we introduce a *new variable* z to represent a convenient intermediate step in the computation. This new variable turns out to be the key to thinking about how to solve quickly with $A + uv^T$, too, since we can think of the problem

$$(A + uv^T)x = b$$

as equivalent to the extended problem

$$\begin{bmatrix} A & u \\ v^T & -1 \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}.$$

Now, suppose we have already computed $PA = LU$, and we want to use this to quickly solve the extended system matrix. Block Gaussian elimination gives

$$\begin{bmatrix} A & u \\ v^T & -1 \end{bmatrix} = \begin{bmatrix} I & 0 \\ v^T A^{-1} & 1 \end{bmatrix} \begin{bmatrix} A & u \\ 0 & -1 - v^T A^{-1} \end{bmatrix}$$

and forward and backward substitution on the extended system yields

$$\begin{aligned} z &= v^T A^{-1} b / (1 + v^T A^{-1} u) \\ x &= A^{-1}(b - uz) \end{aligned}$$

Putting this altogether gives the Sherman-Morrison formula

$$(A + uv^T)^{-1}b = A^{-1}b - \frac{A^{-1}uv^TA^{-1}b}{1 + v^TA^{-1}u}.$$

But remember our admonition above: *don't form explicit inverses!* In MATLAB, assuming we had already computed $PA = LU$, the natural way to apply the Sherman-Morrison formula would be with code like

```
x0 = U\(L\(P*b));
invAu = U\(L\(P*u));
x = x0 - invAu*( (v'*x0)/(1+v'*invAu) );
```

Forming $A^{-1}b$ and $A^{-1}u$ as in this code requires $O(n^2)$ time given the LU factorization; and the final step of the computation requires only $O(n)$ additional work.

The Sherman-Morrison formula tells us what to do with a rank-1 update. For a solve with a more general rank-k update

$$x = (A + UCV^T)^{-1}b, \quad U, V \in \mathbb{R}^{n\times k}, C \in \mathbb{R}^{k\times k},$$

the slightly more general Sherman-Morrison-Woodbury formula allows us to do a similarly fast computation.

Parametric families of matrices

This last section is a bit farther out, but is something that I've been playing with recently. So I'll tell you about it.

The hard case is not when we have multiple right hand sides, or matrices that vary in a structured way. It's when we have a general parameter-dependent linear system

$$A(s)x(s) = b$$

where A varies according to a small number of parameters s.

Sometimes, we can make progress even in this case, though, based on the empirical observation that the solutions often are well approximated from a low-dimensional subspace. That is, we can find $U \in \mathbb{R}^{n\times k}$ such that

$$x(s) \approx Uy(s).$$

We can often find a good U by sampling X for various parameter values and then using a truncated SVD; this is variously called Proper Orthogonal

Decomposition (POD), Principal Components Analysis (PCA), or truncated Karhunen-Loève (K-L) expansion.

Once we have a basis, though, there's a question of how to find $y(s)$ for a given s. There are several options. One simple approach is to choose y to minimize the residual norm, but there are others. One approach, sometimes called the *Bubnov-Galerkin* approach is to force the residual to be orthogonal to U:

$$U^T \left(A(s)Uy(s) - b \right) = 0.$$

Equivalently, we solve

$$\tilde{A}(s)y(s) = \tilde{b}, \quad \tilde{A} = U^T A U, \quad \tilde{b} = U^T b.$$

But why would we expect anything good to come from this? Let's assume for the moment that U is well-chosen, i.e. the best possible error

$$\|e_{\min}\| = \min_z \|Uz - x(s)\|$$

is small. How close does this approach get to that optimum? The answer to this question involves the same ingredients we've seen before: residuals, norm bounds, and something much like condition numbers.

The optimal error $e_{\min}$ satisfies

$$x = Uz + e_{\min}.$$

Therefore, plugging into the Bubnov-Galerkin ansatz, we have

$$\begin{aligned} 0 = U^T(AUy - b) &= U^T \left[AUy - A(uz + e_{\min}) \right] \\ &= U^T AU(y - z) - U^T A e_{\min} \end{aligned}$$

That is,

$$y - z = \tilde{A}^{-1} U^T A e_{\min}.$$

Multiply by U to get

$$U(y - z) = U\tilde{A}^{-1} U^T A e_{\min}.$$

On the other hand, $0 = x - x = (Uy + e_{\min}) - (Uz + e)$; therefore,

$$U(y - z) = e_{\min} - e.$$

Putting it all together,

$$e = (I - U\tilde{A}^{-1}U^T A)e_{\min}.$$

Taking norms (and assuming an operator norm), we have

$$\|e\| \leq (1 + \kappa_{BG})\|e_{\min}\|$$

where

$$\kappa_{BG} = \|U\|\|\tilde{A}^{-1}\|\|U^T A\|.$$

If U has orthonormal columns, then $\|U\| = \|U^T\| = 1$, and another round of norm bounds gives

$$\kappa_{BG} \leq \|\tilde{A}^{-1}\|\|A\|.$$

This is almost the same as the ordinary condition number, except that instead of the norm of the inverse of A, we have the norm of the inverse of the projected system $\tilde{A}$.