Notes 5: Numerical linear algebra

5.1 Systems of linear equations

One of the most basic tasks of scientific computing is to find solutions of sets of linear algebraic equations. In general we can have m equations in n unknowns, $x_0, \ldots x_{n-1}$. We write the linear system in matrix-vector form:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}, \tag{5.1}$$

where

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & \ldots & a_{0,n-1} \\ a_{10} & a_{11} & \ldots & a_{1,n-1} \\ & \ldots & & \\ a_{m-1,0} & a_{m-1,1} & \ldots & a_{m-1,n-1} \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ \ldots \\ x_{n-1} \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} b_0 \\ b_1 \\ \ldots \\ b_{m-1} \end{pmatrix}$$

There are three cases:

1. $m < n$
   If the number of equations, m, is less than the number of unknowns, n, then the system Eq. (5.1) is said to be *under-determined* and there is either no solution for $\mathbf{x}$ or multiple solutions spanning a space of dimension less than n known as the nullspace of $\mathbf{A}$.
2. $m > n$
   If the number of equations, m, is greater than the number of unknowns, n, then the system Eq. (5.1) is said to be *over-determined* and, in general, there is no solution for $\mathbf{x}$. In this case, we are generally interested in solving the linear least squares problem associated with Eq. (5.1) which involves finding the value of $\mathbf{x}$ comes closest to satisfying Eq. (5.1) in the sense that it minimises the sum of the squared errors:
   $$x_* = \arg\min_{\mathbf{x}} |\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|^2 .$$
3. $m = n$
   When the number of equations equals the number of unknowns we generally expect there to exist a single unique solution. This is not guaranteed however unless the rows, or equivalently (since $\mathbf{A}$ is a square matrix) the columns, of $\mathbf{A}$ are linearly independent. If one or more of the m equations is a linear combination of the others, the matrix $\mathbf{A}$ is said to be row-degenerate. If all of the equations contain two or more of the unknowns in exactly the same linear combination, the matrix $\mathbf{A}$ is said to be column-degenerate. Row and column degeneracy are equivalent for square matrices. If $\mathbf{A}$ is degenerate, the linear system Eq. (5.1) is said to be *singular*. Singular systems effectively have fewer equations (meaning linearly independent equations) than unknowns which puts us back in case 1 above.

From a computational point of view, solving Eq. (5.1) can be a highly non-trivial task, even in the case when $n = m$ and $\mathbf{A}$ is known to be non-degenerate. The reason is again rounding error. It can happen that two equations are sufficiently close to being linearly dependent that round-off errors make them become effectively linearly dependent and a solution algorithm which would find a solution in conventional arithmetic will fail when implemented in floating-point arithmetic. Furthermore, since the solution of Eq. (5.1) involves a large number of additions and subtractions when n is large, round off error can accumulate through the calculation so that the "solution", $\mathbf{x}$, which is found is simply wrong in the sense that when substituted back into Eq. (5.1), $\mathbf{A} \cdot \mathbf{x} - \mathbf{b}$ is significantly different from zero. This is particularly the case for linear systems which are close to being singular in some sense. Such systems are said to be ill-conditioned.

5.2 LU decomposition and applications

5.2.1 Triangular matrices and LU decomposition

An $n \times n$ matrix is lower triangular if all of the entries above the diagonal are zero:

$$\mathbf{L} = \begin{pmatrix} l_{00} & 0 & 0 & \dots & 0 \\ l_{10} & l_{11} & 0 & \dots & 0 \\ l_{20} & l_{21} & l_{22} & \dots & 0 \\ & \dots & & & \\ l_{n-1,0} & l_{n-1,1} & l_{n-1,2} & \dots & l_{n-1,n-1} \end{pmatrix}$$

A matrix is upper triangular if all the entries below the diagonal are zero:

$$\mathbf{U} = \begin{pmatrix} u_{00} & u_{01} & u_{02} & \dots & u_{0,n-1} \\ 0 & u_{11} & u_{12} & \dots & u_{1,n-1} \\ 0 & 0 & u_{22} & \dots & u_{2,n-1} \\ & \dots & & & \\ 0 & 0 & 0 & \dots & u_{n-1,n-1} \end{pmatrix}$$

Triangular matrices are important because linear systems involving such matrices are particularly easy to solve. If $\mathbf{U}$ is upper triangular, then the linear system

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{b} \tag{5.2}$$

can be solved by a simple iterative algorithm known as *back substitution*:

```
$x_{n-1} = \frac{b_{n-1}}{u_{n-1,n-1}}$;
for $i = n-2$ to $0$ do
    $x_i = \frac{1}{u_{i,i}} \left[ b_i - \sum_{j=i+1}^{n-1} u_{i,j}\, x_j \right]$;
end
```

Similarly, if $\mathbf{L}$ is lower triangular, the linear system

$$\mathbf{L} \cdot \mathbf{x} = \mathbf{b} \tag{5.3}$$

can be solved by a simple iterative algorithm known as *forward substitution*:

```
$x_0 = \frac{b_0}{l_{0,0}}$;
for $i = 1$ to $n-1$ do
    $x_i = \frac{1}{l_{i,i}} \left[ b_i - \sum_{j=0}^{i-1} l_{i,j}\, x_j \right]$;
end
```

Suppose we can find a way to write a non-triangular matrix, $\mathbf{A}$ in the form

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \tag{5.4}$$

where $\mathbf{L}$ is lower triangular and $\mathbf{U}$ is upper triangular. This is referred to as an LU decomposition of $\mathbf{A}$. We can then solve the linear system Eq. (5.1) as follows:

- Define $\mathbf{y} = \mathbf{U} \cdot \mathbf{x}$
- Solve the linear system $\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$ by forward substitution to obtain $\mathbf{y}$.
- Solve the linear system $\mathbf{U} \cdot \mathbf{x} = \mathbf{y}$ by back substitution to obtain $\mathbf{x}$.

Note that $\mathbf{L}$ and $\mathbf{U}$ as written above each have $\frac{1}{2}n(n+1)$ non-zero entries. The total number of unknowns in the product $\mathbf{L} \cdot \mathbf{U}$ is therefore $n^2 + n$ whereas there are only n^2 equations implied by Eq. (5.4). There is therefore some freedom to introduce additional constraints on the values of the entries of $\mathbf{L}$ and $\mathbf{U}$ to reduce the number of unknowns. It is common to impose that either $\mathbf{L}$ or $\mathbf{U}$ should be unit triangular, meaning that the n diagonal elements are all equal to 1. We shall adopt the convention that $\mathbf{L}$ is unit triangular, i.e. $l_{ii} = 1$ for $i = 0, \dots, n-1$.

It turns out, however, that many non-singular square matrices do not have an LU decomposition. The reason is already evident in a simple example 3×3 example. Let us try to find an LU decomposition of a general 3×3 matrix (assumed to be non-singular) by writing

$$\begin{pmatrix} l_{00} & 0 & 0 \\ l_{10} & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{pmatrix} \begin{pmatrix} u_{00} & u_{01} & u_{02} \\ 0 & u_{11} & u_{12} \\ 0 & 0 & u_{22} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}.$$

Upon multiplication we see that the first entry gives $a_{00} = l_{00}\, u_{00}$. If it so happens that $a_{00} = 0$ then we would have to choose either $l_{00} = 0$ or $u_{00} = 0$. The first choice would result in $\mathbf{L}$ being singular and the second choice would result in $\mathbf{U}$ being singular. The product $\mathbf{L} \cdot \mathbf{U}$ is then necessarily singular but this is impossible since we have assumed that $\mathbf{A}$ is non-singular. Hence the matrix $\mathbf{A}$ does not have an LU decomposition. We could get around this problem by swapping the first row of $\mathbf{A}$ with either the second or the third row so that the upper left entry of the resulting matrix is nonzero. This must be possible because $\mathbf{A}$ is nonsingular so all three of a_{00}, a_{10} and a_{20} cannot simulataneously be zero. If the same problem is encountered in subsequent steps, it can be removed in the same way. We conclude that if the matrix $\mathbf{A}$ does not have an LU decomposition, there is a permutation of the rows of $\mathbf{A}$ which does.

There is a general principle at work here. It turns out that for every nondegenerate square matrix, there exists a permutation of the rows which does have an LU decomposition. Such a permutation of rows corresponds to left multiplication by a permutation matrix, $\mathbf{P}$, which is a matrix obtained by permuting the columns of an $n \times n$ identity matrix. From the point of view of solving linear systems, multiplication by $\mathbf{P}$ makes no difference provided that we also multiply the righthand side, $\mathbf{b}$, by $\mathbf{P}$. We are simply writing the equations in a different order: $\mathbf{P} \cdot \mathbf{A} \cdot \mathbf{x} = \mathbf{P} \cdot \mathbf{b}$. The decomposition

$$\mathbf{P} \cdot \mathbf{A} = \mathbf{L} \cdot \mathbf{U}$$

is known as *LU decomposition with partial pivoting* and can be used to solve general nonsingular systems of linear equations.

5.2.2 Crout's algorithm for LU decomposition with partial pivoting

Having established that the LU decomposition of a matrix is useful, let us now consider how to actually compute it. The basic procedure is known as Crout's algorithm. It works by writing out all n^2 terms in the product in Eq. (5.4) in a special order so that the nonzero values of u_{ij} and l_{ij} can be explicitly solved for using a procedure similar to the back and forward substitution algorithms described above.

Let us first look at the (i, j) entry of the product in Eq. (5.4) which gives a_{ij}. It is obtained by taking the dot product of row i of $\mathbf{L}$ with column j of $\mathbf{U}$. Row i of $\mathbf{L}$ has i nonzero elements:

$$(l_{i0}, l_{i1} \ldots l_{i,i-1}, l_{i,i}, 0, \ldots 0).$$

Column j of $\mathbf{U}$ has j nonzero elements:

$$(u_{0j}, u_{1j} \ldots u_{j-1,j}, u_{j,j}, 0, \ldots 0).$$

When we take the dot product there are three cases:

1. $\mathbf{i} < \mathbf{j}$: In this case we have

   $$a_{ij} = l_{i0}u_{0j} + l_{i1}u_{1j} + \ldots + l_{i,i-1}u_{i-1,j} + l_{ii}u_{ij}$$

   Using the fact that we have adopted the convention that $l_{ii} = 1$ we can write

   $$u_{ij} = a_{ij} - \sum_{k=0}^{i-1} l_{ik}u_{kj}. \tag{5.5}$$

2. $\mathbf{i} = \mathbf{j}$: Actually this is the same as case 1 with $j = i$. We have

   $$a_{ii} = l_{i0}u_{0i} + l_{i1}u_{1i} + \ldots + l_{i,i-1}u_{i-1,i} + l_{ii}u_{ii},$$

   which gives

   $$u_{ii} = a_{ii} - \sum_{k=0}^{i-1} l_{ik}u_{ki}. \tag{5.6}$$

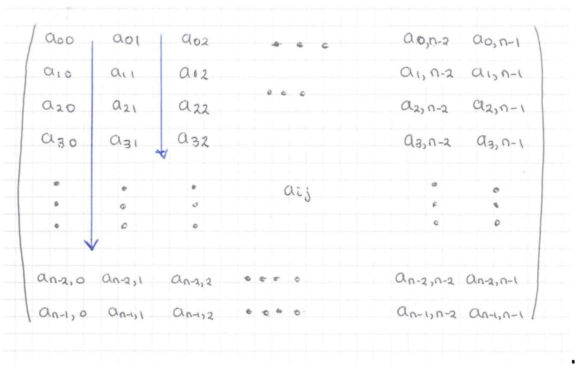
3. $\mathbf{i} > \mathbf{j}$: In this case we have

$$a_{ij} = l_{i0}u_{0j} + l_{i1}u_{1j} + \ldots + l_{i,j-1}u_{j-1,j} + l_{ij}u_{jj}.$$

This gives an equation for l_{ij}:

$$l_{ij} = \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=0}^{j-1} l_{ik}u_{kj}\right). \tag{5.7}$$

At first sight, Eqs. (5.5), (5.6) and (5.7) do not appear to get us any further towards our goal since the unknown l_{ij} and u_{ij} appear on both sides of each. This is where the bit about writing the equations in the correct order comes in. To begin with, let us assume that no pivoting is required. Crout's algorithm works its way through the matrix $\mathbf{A}$ starting at the top left element a_{00}, traversing the row index first and then incrementing the column index:



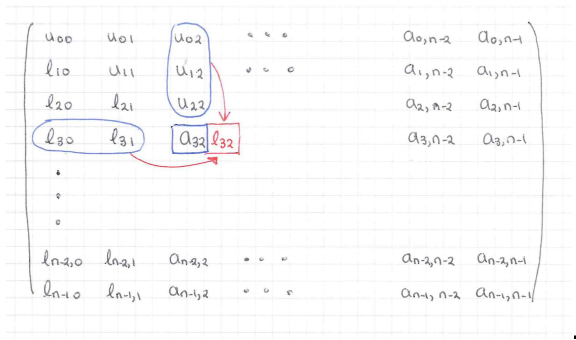
a00 a01 a02 . . . a0,n-2 a0,n-1
a10 a11 a12 . . . a1,n-2 a1,n-1
a20 a21 a22 a2,n-2 a2,n-1
a30 a31 a32 a3,n-2 a3,n-1
aij
an-2,0 an-2,1 an-2,2 . . . an-2,n-2 an-2,n-1
an-1,0 an-1,1 an-1,2 . . . an-1,n-2 an-1,n-1

.

As it progresses it computes as follows:

```
Begin by setting the $l_{ii}$ to 1
**for** $i = 0$ **to** $n-1$ **do**
    $l_{ii} = 0$;
**end**
First loop over columns
**for** $j = 0$ **to** $n-1$ **do**
    Loop over rows in three stages
    Stage 1
    **for** $i = 0$ **to** $j-1$ **do**
        Since $j < i$ we use Eq. (5.5) to compute $u_{ij}$
        $u_{ij} = a_{ij} - \sum_{k=0}^{i-1} l_{ik}u_{kj}$;
        $a_{ij} = u_{ij}$;
    **end**
    Stage 2: i=j;
    Since $i = j$ we use Eq. (5.6) to compute $u_{ii}$
    $u_{ii} = a_{ii} - \sum_{k=0}^{i-1} l_{ik}u_{ki}$;
    $a_{ii} = u_{ii}$;
    Stage 3
    **for** $i = j+1$ **to** $n-1$ **do**
        Since $j > i$ we use Eq. (5.7) to compute $l_{ij}$
        $l_{ij} = \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=0}^{j-1} l_{ik}u_{kj}\right)$;
        $a_{ij} = l_{ij}$;
    **end**
**end**
```

Here is the state of the array $\mathbf{A}$ after this procedure has reached the element a_{32}:


u_{00} u_{01} u_{02} … $a_{0,n-2}$ $a_{0,n-1}$
l_{10} u_{11} u_{12} … $a_{1,n-2}$ $a_{1,n-1}$
l_{20} l_{21} u_{22} $a_{2,n-2}$ $a_{2,n-1}$
l_{30} l_{31} a_{32} l_{32} $a_{3,n-2}$ $a_{3,n-1}$
$l_{n-2,0}$ $l_{n-2,1}$ $a_{n-2,2}$ … $a_{n-2,n-2}$ $a_{n-2,n-1}$
$l_{n-1,0}$ $l_{n-1,1}$ $a_{n-1,2}$ … $a_{n-1,n-2}$ $a_{n-1,n-1}$
.

Notice how the entries of $\mathbf{L}$ and $\mathbf{U}$ required to compute a_{32} have already been computed by the time the algorithm reaches a_{32}. This is true for all entries $a_{i,j}$. Notice also that the value of a_{32} is never needed again since it is only required in the computation of l_{32}. This is also true for all entries $a_{i,j}$. This is why as the algorithm works its way through the array, it can over-write the successive values of a_{ij} with the computed value of l_{ij} or u_{ij} as shown. Crout's algorithm is an example of an *in-place* algorithm. It does not need additional storage to calculate its output and returns the output in the same physical array as its input. After completion, the array $\mathbf{A}$ looks as follows:

$$\mathbf{A} = \begin{pmatrix} u_{00} & u_{01} & u_{02} & \dots & u_{0,n-1} \\ l_{10} & u_{11} & u_{12} & \dots & u_{1,n-1} \\ l_{20} & l_{21} & u_{22} & \dots & u_{2,n-1} \\ & \dots & & & \\ l_{n-1,0} & l_{n-1,1} & l_{n-1,2} & \dots & l_{n-1,n-1} \end{pmatrix}.$$

Remember that the diagonal entries of $\mathbf{L}$ are not stored since we have adopted the convention that $\mathbf{L}$ is unit triangular. The only question which remains is how to incorporate pivoting. The need for pivoting is evident from the equation for l_{ij} in the algorithm. Here there is a trick which is explained in [1, chap. 2] but looks suspiciously like magic to me.

Note that the LU decomposition contains 3 nested loops. It is therefore an $O(n^3)$ algorithm.

5.2.3 Other applications of LU decomposition: matrix inverses and determinants

1. **Computing the inverse of a matrix**
   In the vast majority of cases when a linear system like Eq. (5.1) appears, we are interested in finding $\mathbf{x}$. This is much more efficiently done using the algorithm described above rather than by explicitly computing $\mathbf{A}^{-1}$ and then computing the product $\mathbf{A}^{-1} \cdot \mathbf{b}$. Nevertheless on the rare few occasions when you do really need $\mathbf{A}^{-1}$, the LU decomposition can be used to compute it. To see how to do it note that the LU decomposition can also be used to solve matrix equations of the form
   $$\mathbf{A} \cdot \mathbf{X} = \mathbf{B} \tag{5.8}$$
   where $\mathbf{X}$ and $\mathbf{B}$ are $n \times m$ *matrices*. The case $m = 1$ corresponds to Eq. (5.1). The trick is to realise that once the LU decomposition has been computed, we can use the forward and back substitution algorithm to solve a sequence of m linear systems containing the successive columns of $\mathbf{B}$ as their right hand sides. The results will be the columns of $\mathbf{X}$. To obtain $\mathbf{A}^{-1}$ we take $\mathbf{B}$ to be an $n \times n$ identity matrix. The matrix $\mathbf{X}$ obtained in this case is then the inverse of $\mathbf{A}$.
2. **Computing the determinant of a matrix**
   Determinants of matrices are easy to compute from the LU decomposition by observing that the determinant of a triangular matrix is just the product of the diagonal elements. If we have an LU decomposition $\mathbf{P} \cdot \mathbf{A} = \mathbf{L} \cdot \mathbf{U}$ in which we have arranged for $\mathbf{L}$ to be unit triangular, then
   $$\det(\mathbf{A}) = \det(\mathbf{P}^{-1}) \det(\mathbf{L}) \det(\mathbf{U}) = (-1)^s \left(\prod_{i=0}^{n-1} u_{ii} \right) \tag{5.9}$$
   where s is the number of row exchanges in the permutation matrix $\mathbf{P}$.

5.3 Sparse matrices and conjugate gradient method revisited

5.3.1 Sparse linear systems

A linear system is called *sparse* if only a small number of is matrix elements, a_{ij}, are nonzero. Here "small" usually means $O(n)$. Sparse matrices occur very frequently: some of the most common occurences are in finite difference approximations to differential equations and adjacency matrices for networks with low average degree. An archtypal example of a sparse matrix is a a tri-diagonal matrix in which the only non-zero elements are on the diagonal plus or minus one column:

$$\mathbf{A} = \begin{pmatrix} d_0 & a_0 & 0 & 0 & \dots & & 0 \\ b_1 & d_1 & a_1 & 0 & \dots & & 0 \\ 0 & b_2 & d_2 & a_2 & \dots & & 0 \\ 0 & 0 & b_3 & d_3 & \dots & & 0 \\ & \dots & & & & & \\ 0 & 0 & 0 & \dots & b_{n-2} & d_{n-2} & a_{n-2} \\ 0 & 0 & 0 & \dots & 0 & b_{n-1} & d_{n-1} \end{pmatrix}. \tag{5.10}$$

It is very inefficient to use methods designed for general matrices to solve sparse problems since most of your computer's memory will be taken up with useless zeros and most of the $O(n^3)$ operations required to solve a linear system for example would be trivial additions and multiplications of zeroes. Using the LU decomposition algorithm of Sec. 5.2, dense matrices with n of the order hundreds can be routinely solved (assuming that they are not too close to being singular). As n starts to get of order 1000, computational time starts to become the limiting factor. For sparse matrices, however values of n in the tens of thousands can be done routinely and calculations with n in the millions can be tackled without difficulty on parallel machines. Such matrices could not even be *stored* in dense formats. It is therefore very important to exploit sparsity when it is present in a problem

As an example, calculation of the matrix-vector product $\mathbf{A} \cdot \mathbf{x}$ requires $O(2n^2)$ operations. For the tridiagonal matrix in Eq. (5.10), it is clear that $\mathbf{A} \cdot \mathbf{x}$ can be computed in about $O(3n)$ operations. Storing the matrix also requires only $O(3n)$ doubles as opposed to n^2 for an equivalently sized dense matrix.

5.3.2 The biconjugate gradient algorithm

The biconjugate gradient algorithm is an iterative method for solving Eq. (5.1) which makes use only of matrix-vector multiplications. Since matrix-vector multiplication can be implemented in $O(n)$ time for sparse matrices, the resulting algorithm is much faster than the LU decomposition discussed in Sec. 5.2 (although more efficient implementations of LU decomposition which take advantage of sparsity *do* exist, their stability properties are usually inferior to the biconjugate gradient algorithm). The connection to the regular conjugate gradient algorithm which we met in Sec. 4.4.2 will become clearer in due course. We first describe the algorithm.

Start with pair of vectors $\mathbf{g}_0$ and $\bar{\mathbf{g}}_0$ and set $\mathbf{h}_0 = \mathbf{g}_0$ and $\bar{\mathbf{h}}_0 = \bar{\mathbf{g}}_0$. We then construct four sequences of vectors, $\mathbf{g}_k$, $\bar{\mathbf{g}}_k$, $\mathbf{h}_k$ and $\bar{\mathbf{h}}_k$ from the following recurrence:

$$\begin{aligned} \lambda_k &= \frac{\bar{\mathbf{g}}_k \cdot \mathbf{g}_k}{\bar{\mathbf{h}}_k \cdot \mathbf{A} \cdot \bar{\mathbf{h}}_k} && (5.11) \\ \mathbf{g}_{k+1} &= \mathbf{g}_k - \lambda_k \mathbf{A} \cdot \mathbf{h}_k \quad \bar{\mathbf{g}}_{k+1} = \bar{\mathbf{g}}_k - \lambda_k \mathbf{A}^{\mathrm{T}} \cdot \bar{\mathbf{h}}_k && (5.12) \\ \gamma_k &= \frac{\bar{\mathbf{g}}_{k+1} \cdot \mathbf{g}_{k+1}}{\bar{\mathbf{g}}_k \cdot \bar{\mathbf{g}}_k} && \\ \mathbf{h}_{k+1} &= \mathbf{g}_{k+1} + \gamma_k \mathbf{h}_k \quad \bar{\mathbf{h}}_{k+1} = \bar{\mathbf{g}}_{k+1} + \gamma_k \bar{\mathbf{h}}_k. && (5.13) \end{aligned}$$

The sequences of vectors constructed in this way have the properties of

$$\begin{aligned} \text{bi-orthogonality:} \quad & \bar{\mathbf{g}}_i \cdot \mathbf{g}_j = \mathbf{g}_i \cdot \bar{\mathbf{g}}_j = 0 && (5.14) \\ \text{bi-conjugacy:} \quad & \bar{\mathbf{h}}_i \cdot \mathbf{A} \cdot \bar{\mathbf{h}}_j = \mathbf{h}_i \cdot \mathbf{A}^{\mathrm{T}} \cdot \bar{\mathbf{h}}_j = 0 && (5.15) \\ \text{mutual orthogonality:} \quad & \bar{\mathbf{g}}_i \cdot \mathbf{h}_j = \mathbf{g}_i \cdot \bar{\mathbf{h}}_j = 0, && (5.16) \end{aligned}$$

for any $j < i$. These statements are claimed in [1] to be provable by induction although I haven't tried. Note that this recurrence must terminate after n iterations with $\mathbf{g}_n = \bar{\mathbf{g}}_n = 0$. This is because of the orthogonality conditions: after n steps the n previously constructed $\mathbf{g}_k$ must be orthogonal to $\bar{\mathbf{g}}_n$ but this is impossible since the dimension of the space is n. To use Eqs. (5.11)-(5.13) to solve Eq. (5.1), we take an initial guess $\mathbf{x}_0$ and choose our starting vectors to be

$$\mathbf{g}_0 = \bar{\mathbf{g}}_0 = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_0. \tag{5.17}$$

As we progress through the iteration, we construct a sequence of improved estimates of the solution of Eq. (5.1):

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{h}_k. \tag{5.18}$$

Now we remark that

$$\mathbf{g}_k = \mathbf{b} - \mathbf{A}\cdot\mathbf{x}_k. \tag{5.19}$$

Why? By induction, if we assume that Eq. (5.19) is true for k then

$$\begin{aligned}\mathbf{g}_{k+1} &= \mathbf{g}_k - \lambda_k \mathbf{A}\cdot\mathbf{h}_k \\ &= \mathbf{b} - \mathbf{A}\cdot\mathbf{x}_k - \lambda_k \mathbf{A}\cdot\mathbf{h}_k \\ &= \mathbf{b} - \mathbf{A}\cdot(\mathbf{x}_k + \lambda_k \mathbf{h}_k) \\ &= \mathbf{b} - \mathbf{A}\cdot\mathbf{x}_{k+1}\end{aligned}$$

thus establishing that Eq. (5.19) is true for $k+1$. Since the statement holds for $k=0$ by the definition (5.17) of $\mathbf{g}_0$, the statement is proven for all k. The importance of Eq. (5.19) is that the vectors $\mathbf{g}_k$ constructed by the bi-conjugate gradient algorithm, Eqs. (5.11)-(5.13), have the meaning of a *residual*. They measure the distance of our current estimate, $\mathbf{x}_k$, from the true solution. Since we have established that the terminating residual is $\mathbf{g}_n = 0$, $\mathbf{x}_n$ must be the solution of the original linear system.

Let us suppose that $\mathbf{A}$ is symmetric and (in order to avoid zero denominators) positive definite. If we choose $\mathbf{g}_0 = \bar{\mathbf{g}}_0$ then Eqs. (5.11)-(5.13) become equivalent to the "regular" conjugate gradient algorithm, Eqs.(4.22)-(4.23), which we encountered in our study of multi-dimensional optimisation. To see the connection, note that if $\mathbf{A}$ is symmetric and positive definite, it defines a quadratic form for any vector $\mathbf{b}$ given by

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}\cdot\mathbf{A}\cdot\mathbf{x} - \mathbf{b}\cdot\mathbf{x}, \tag{5.20}$$

which has a single minimum at which the gradient, $\nabla f(\mathbf{x})$, is equal to zero. But

$$\nabla f(\mathbf{x}) = \mathbf{A}\cdot\mathbf{x} - \mathbf{b},$$

so minimising the quadratic form is equivalent to finding the solution of the linear system! Historically, the conjugate gradient algorithm in its regular form was invented first in order to solve symmetric positive definite systems of linear equations. The biconjugate gradient algorithm came later to generalise the method to non-symmetric systems of linear equations but the connection with function optimisation is lost in this generalisation. The use of the conjugate gradient algorithm for nonlinear optimisation of functions which can be approximated by quadratic forms came even later still.

Bibliography

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes: The art of scientific computing (3rd ed.).* Cambridge University Press, New York, NY, USA, 2007.