Chapter 8

Solutions of Systems of Equations

A brief review

A vector $x \in R^n$ is an ordered n-tuple of real numbers i.e. $x = (x_1, x_2, \ldots, x_n)^T$ where the T denotes this should be considered a column vector.

A matrix, $A \in R^{m \times n}$ is a rectangular array of m rows and n columns.

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ \vdots & & & \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix}$$

Given a square matrix, $A \in R^{n \times n}$ if there exists a second square matrix $B \in R^{n \times n}$ such that $AB = BA = I$ then we say B is the inverse of A. Note that not all sqaure matrices have an inverse. If A has an inverse it is *nonsingular* and if it does not it is *singular*.

The following theorem summarises the conditions under which a matrix is nonsingular and also connects them to the solvability of the linear systems problem.

Theorem

Given a matrix $A \in R^{n \times n}$ the following statements are equivalent:

1. A is nonsingular

2. The columns of A form and independent set of vectors
3. The rows of A form an independent set of vectors
4. The linear system $Ax = b$ has a unique solution for all vectors $b \in R^n$.
5. The homogeneous system $Ax = 0$ has only the trivial solution $x = 0$.
6. The determinant is nonzero.

Corollary

If $A \in R^{n \times n}$ is singular, then there exist infinitely many vectors $x \in R^n$, $x \neq 0$ such that $Ax = 0$.

There are a number of special classes of matrices. In particular, *tridiagonal* matrices and (later) *symmetric positive definite* matrices.

A square matrix is lower (upper) triangular if all the elements above (below) the main diagonal are zero. Thus

$$U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix}$$

is upper tringular, while

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 0 \end{bmatrix}$$

is lower triangular.

Note at this stage you should review concepts of *spanning, basis, dimension and orthogonality* as we will use these ideas soon in a discussion of eignvalues and their computation.

Linear systems and Gaussian elimination

Have seen the solution of tridiagonal systems using Gaussian elimination.
Idea:
Tridiagonal matrix $\longrightarrow$ Triangular matrix $\longrightarrow$ Solve by back substitution.

This transformation is accomplished using only the row operations that preserve the solution set, namely

1. Multiply a row by a nonzero scalar, c.
2. Interchange two rows
3. Multiply a row by a nonzero scalar, c and add the result to another row.

Then for A' the augmented matrix describing the system $Ax = b$ a new matrix A'' which is derived from A' by the operations above is *row equivalent* and has the same solution set.

In this section we will derive a more general algorithm to deal with any matrix (not just tridiagonal). The goal is to use the row operations above to reduce A' to a new augmented matrix $A'' = [U|c]$, where U is upper triangular and $Ux = c$ can be easily solved.

Essential features

Work down each column, eliminating (ie converting to zero) each component below the main diagonal and modifying the rest of the corresponding row appropriately.

Naive Gaussian algorithm for Ax=b

```
for i=1 to n-1
  for j=i+1 to n
    m = a(j,i)/a(i,i)
    for k=i+1 to n
      a(j,k) = a(j,k) - m*a(i,k)
    endfor
    b(j) = b(j) - m*b(i)
  endfor
endfor
```

Programming notes:

The outermost loop (the i loop) ranges over the columns of the matrix; the

last column is skipped because we don't need to perform any eliminations there (since there are no elements below the diagonal). Note that elimination on a non-square matrix would require this last column. The j loop ranges down the i^{th} column, below the diagonal (hence j goes from i+1 to n). First compute the multiplier, m for each row, to eliminate the a_{ji} element. Note that previous values are overwritten and the computation that makes a_{ji} zero isn't done. In this loop also the right-hand side is modified. The k loop ranges across the j^{th} row starting after the i^{th} column. modifying elements to reflect the elimination of a_{ji}.

Note the algorithm doesn't create or store the zeroes in the lower triangular half of A. This is ok because we only need the upper triangular part to solve the equation. What if you have another system of equations with the same coefficient matrix you need a method that saves this information - LU decomposition.

Back-substitution for Ax=b

```
x(n) = b(n)/a(n,n)
for i=n-1 to 1
  sum = 0
  for j=i+1 to n
    sum = sum + a(i,j)*x(j)
  endfor
  x(i) = (b(i) - sum)/a(i,i)
endfor
```

Programming notes:

This algorithm moves back up the diagonal, computing the x_i in turn. I.e.

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right)$$

to solve the triangular system. The j loop is accumulating the summation term in the equation above. Notice that when the multiplier is computed it requires division by the current diagonal element. If this is zero the algorithm fails. But this does not imply the matrix is singular rather that the algorithm is deficient.

In this regard the diagonals are called *pivots* and the solution, swapping rows to avoid a zero is called *pivoting*.

- **Partial pivoting:** only entries in the same column are considered. The most common solution and we will discuss it below.
- **Complete pivoting:** uses the current column but also all other columns. More stable but more expensive.

Partial pivoting

An outline algorithm has the form:

- consider the i^{th} column of the matrix. Search the portion of the i^{th} column including and below the diagonal to find the element with largest absolute value. Let p is the row index of this element.
- Interchange rows i and p.
- Proceed with elimination

Implementing this procedure has an extra benefit - the algorith becomes less susceptible to rounding error.

Example

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{8.1}$$

For $\epsilon \neq 0$ the matrix is nonsingular and a unique solution exists. The exact solution, in terms of ϵ is

$$\begin{aligned} x_1 &= \frac{1}{1-\epsilon} = 1 + \mathcal{O}(\epsilon) \\ x_2 &= \frac{1-2\epsilon}{1-\epsilon} = 1 + \mathcal{O}(\epsilon) \end{aligned}$$

but consider solving this system for ϵ very small.

Without pivoting:

$$\left[\begin{array}{cc|c} \epsilon & 1 & 1 \\ 1 & 1 & 2 \end{array}\right] \sim \left[\begin{array}{cc|c} \epsilon & 1 & 1 \\ 0 & 1-1/\epsilon & 2-1/\epsilon \end{array}\right] \tag{8.2}$$

Suppose that ϵ is so small that, to machine precision, $1 - 1/\epsilon = -\epsilon^{-1}$ and $2 - 1/\epsilon = -\epsilon^{-1}$, then

$$\left[\begin{array}{cc|c} \epsilon & 1 & 1 \\ 0 & 1-1/\epsilon & 2-1/\epsilon \end{array}\right] \equiv \left[\begin{array}{cc|c} \epsilon & 1 & 1 \\ 0 & -1/\epsilon & -1/\epsilon \end{array}\right] \tag{8.3}$$

with the result that $x_2 \equiv 1$ and $x_1 \equiv 0$. You see that while we get the right answer for x_2, x_1 is not correct. The problem is rounding error introduce by the large number $1/\epsilon$.

With pivoting:

$$\left[\begin{array}{cc|c} \epsilon & 1 & 1 \\ 1 & 1 & 2 \end{array}\right] \sim \left[\begin{array}{cc|c} 1 & 1 & 2 \\ \epsilon & 1 & 1 \end{array}\right] \sim \left[\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 1-\epsilon & 1-2\epsilon \end{array}\right] \equiv \left[\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 1 & 1 \end{array}\right]. \tag{8.4}$$

And $x_2 \equiv 1$, $x_1 \equiv 1$. A nice illustration of the benefits of pivoting even without zero diagonal elements.

Operation Counts

Many practical problems use very large matrices it is important to know the computational cost of a specific algorithm. You can do this by counting *operations*. Conventionally one counts multiplication and division.

For the naive Gaussian elimination write the algorithm as a series of sums (corresponding to the loops) so that

$$count = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \left(2 + \sum_{k=i+1}^{n} 1 \right)$$

An answer that depends on the matrix size is of more interest than an exact figure so determine how the largest term in the count depends on the matrix size. Then,

$$\begin{aligned} \sum_{j=i+1}^{n} \left(2 + \sum_{k=i+1}^{n} 1 \right) &= \sum_{j=i+1}^{n} \left(2 + \sum_{m=1}^{n-i} 1 \right) \\ &= \sum_{j=i+1}^{n} (n - i + 2) \\ &= (n - i)(n - i + 2) \end{aligned}$$

Then

$$\begin{aligned} count &= \sum_{i=1}^{n-1} (n - i)(n - i + 2) \\ &= \sum_{m=1}^{n-1} (m^2 + 2m) \\ &= \frac{1}{6} n(n-1)(2n-1) + n(n-1) \\ &= \frac{1}{3} n^3 + \mathcal{O}(n^2) \end{aligned}$$

where the formulae $\sum_{k=1}^{n} k = \frac{1}{2} n(n+1)$ and $\sum_{k=1}^{n} k^2 = \frac{1}{6} n(n+1)(2n+1)$ have been used. Thus we have an operation count of $\frac{1}{3} n^3 + \mathcal{O}(n^2)$. Repeating this analysis for the backsubstitution algorithm gives

$$count = \frac{1}{2} n^2 + \mathcal{O}(n)$$

So then the total cost is

$$totalcost = \frac{1}{3} n^3 + \mathcal{O}(n^2) + \frac{1}{2} n^2 + \mathcal{O}(n)$$

and you see that the solution step is a power of n cheaper than the elimination step.