SECTION 5
Matrix Algebra

Outline of Section

- Linear Algebraic Equations
- Direct Method - LU Decomposition
- Iterative Methods
- Eigensystems

In this section we will look at solving simultaneous linear algebraic equations of the form

$$\begin{aligned} a_{11}\,x_1 + a_{12}\,x_2 + a_{13}\,x_3 + ... + a_{1N}\,x_N &= b_1\,, \\ a_{21}\,x_1 + a_{22}\,x_2 + a_{23}\,x_3 + ... + a_{2N}\,x_N &= b_2\,, \\ ... &= ...\,, \\ a_{M1}\,x_1 + a_{M2}\,x_2 + a_{M3}\,x_3 + ... + a_{MN}\,x_N &= b_M\,, \end{aligned} \tag{5.1}$$

which can be written as the matrix operation

$$\mathbf{A}\cdot\vec{x} = \vec{b}\,, \tag{5.2}$$

where $\mathbf{A}$ is an $M \times N$ matrix, $\vec{x}$ is a vector with N components and $\vec{b}$ is a vector with M components. A given coefficient a_{ij} in the above simultaneous equations becomes the element of the matrix located at row i (i.e. the first subscript) and column j (the 2nd subscript). We want to solve for the unknown variables $\vec{x}$ for a given linear operator $\mathbf{A}$ and right-hand side $\vec{b}$. M is the number of equations in the system and N is the number of unknowns we have to solve for.

The need to solve a matrix equation such as (5.2) arises frequently in many numerical methods. In section 4.4 we saw them in implicit finite difference methods for solving sets of coupled linear ODEs. In section 9 we will encounter them in solving PDEs via finite difference methods. The $N \times N$ matrices there will be very large, with N equal to the number of spatial grid points! Matrix equations will also occur in solving boundary value problems (section 6) and minimisation of functions (section 7). Of course matrix equations arise directly in many physical problems too, such as quantum mechanics and

classical mechanics. The need to find eigenvalues of a matrix is also a common task both in numerical methods and in physics problems. In the former case, think back to the stability analysis of FDMs for coupled ODEs in section 3.4.

Recap: direct methods – You will already know some *direct methods* for solving $\mathbf{A}\cdot\vec{x}=\vec{b}$, such as **Gaussian elimination** and possibly **Gauss-Jordan elimination**. These direct methods involve manipulating the matrix to eliminate elements so that the system can be easily solved. The manipulation involves, e.g., swapping rows, adding a multiple of one row to another, etc. For Gaussian elimination one manipulates the augmented matrix formed by $\left(\mathbf{A}|\vec{b}\right)$, in an effort to end up with $\mathbf{A}$ in upper triangular form. For Gauss-Jordan elimination one manipulates the augmented matrix formed by $(\mathbf{A}|\mathbf{I})$, until one has $(\mathbf{I}|\mathbf{C})$, which yields the inverse of $\mathbf{A}$ since it turns out that $\mathbf{C}=\mathbf{A^{-1}}$. In both methods, $\mathbf{A}$ dictates the manipulations chosen with $\vec{b}$ or $\mathbf{I}$ 'shadowing' these moves and thereby becoming transformed. This section of the course concentrates mainly on **iterative methods** for solving $\mathbf{A}\cdot\vec{x}=\vec{b}$ and finding eigenvalues. These turn out to be more suitable than direct methods for large matrices.

General Considerations

Before diving into various new methods it is worth discussing some general points about solving matrix equations. Consider the inhomogeneous matrix equation $\mathbf{A}\cdot\vec{x}=\vec{b}$. If $M=N$ and all equations are **linearly independent** then there should exist a unique solution for $\vec{x}$. However if some equations are degenerate (i.e. can be written as linear combination of other equations) then effectively the system has less equations than unknowns $M<N$ and there may not be a solution (or a unique solution). This will be evident in the matrix $\mathbf{A}$ being singular (some eigenvalues are zero) and having a vanishing determinant. An approximate solution can be found using Singular Value Decomposition (SVD), not covered here.

Conversely if $M>N$ the system is over-determined. In general no solution exists in this case but an approximate one can usually be found by a linear least squares search for the solution fitting the equations most closely.

In particular for the $M=N$ case the system can be extended to N unknown solutions for N right-hand sides i.e.

$$\mathbf{A}\cdot\mathbf{X}=\mathbf{B}\,, \tag{5.3}$$

where $\mathbf{X}$ and $\mathbf{B}$ are now $N\times N$ matrices too. Each column of $\mathbf{X}$ is one of the vectors $\vec{x}_i$ of the N equations (where $1\leq i\leq N$). Similarly, each column of $\mathbf{B}$ is the right hand-side vector $\vec{b}_i$ of one of the equations.

If we can solve for $\mathbf{X}$ then we can also use the methods to find the inverse of a matrix since

$$\mathbf{A}\cdot\mathbf{A}^{-1}=\mathbf{I}\ , \tag{5.4}$$

so setting $\mathbf{B}$ to the identity matrix so the matrix equations becomes $\mathbf{A}\cdot\mathbf{X}=\mathbf{I}$ and solving yields solution $\mathbf{X}=\mathbf{A}^{-1}$. As we will see we can also get the determinant of a matrix for free.

Finally, if the matrix equation is homogeneous, i.e., $\mathbf{A}\cdot\vec{x}=\vec{0}$ (where $\mathbf{A}$ is an $N\times N$ matrix), then the solution is non-trivial only if $\det\mathbf{A}=0$.

5.1. LU Decomposition

This is a direct method. We will focus on the case where $M=N$, i.e., $\mathbf{A}$ is a square matrix. We will assume that the matrix can be factorised into upper and lower triangular matrices

$$\mathbf{A}\equiv\mathbf{L}\cdot\mathbf{U} \tag{5.5}$$

where $\mathbf{L}$ and $\mathbf{U}$ are of the form

$$\begin{pmatrix}\alpha_{11}&0&0\\ \alpha_{21}&\alpha_{22}&0\\ \alpha_{31}&\alpha_{32}&\alpha_{33}\end{pmatrix}\quad\text{and}\quad\begin{pmatrix}\beta_{11}&\beta_{12}&\beta_{13}\\ 0&\beta_{22}&\beta_{23}\\ 0&0&\beta_{33}\end{pmatrix}, \tag{5.6}$$

respectively. The algorithm to carry out LU decomposition is related to Gaussian elimination.

In this case we can write the system

$$\boxed{\mathbf{A}\cdot\vec{x}=\mathbf{L}\cdot\mathbf{U}\cdot\vec{x}=\mathbf{L}\cdot\vec{y}=\vec{b}\ ,} \tag{5.7}$$

where $\vec{y}=\mathbf{U}\cdot\vec{x}$.

The trick here is that it is trivial to solve a triangular system, i.e., one where each equation is a function of one more unknown than the previous one. To do this we first carry out a **forward substitution** to solve for $\vec{y}$.

$$y_1 = \frac{b_1}{\alpha_{11}} \tag{5.8}$$

$$y_i = \frac{1}{\alpha_{ii}}\left(b_i-\sum_{j=1}^{i-1}\alpha_{ij}y_j\right)\quad i=2,3,...,N\quad\text{(in this order)}\ . \tag{5.9}$$

We then solve for $\vec{x}$ by carrying out a **backwards substitution**

$$x_N = \frac{y_N}{\beta_{NN}} \tag{5.10}$$

$$x_i = \frac{1}{\beta_{ii}}\left(y_i-\sum_{j=i+1}^{N}\beta_{ij}x_j\right)\quad i=N-1,N-2,...,1\ , \tag{5.11}$$

where the calculation in (5.11) must be carried out in the order shown. We will not look into the details of how the decomposition is carried out (see e.g. *Numerical Recipes*). It `Python` it can be carried out using the `SciPy` linear algebra function `scipy.linalg.lu(...)` . For other languages, black-box routines in Linear Algebra packages (e.g. `LINPACK` and `LAPACK`) can be used. The decomposition requires $\mathcal{O}(N^3)$ operations.

If we want to find the inverse of $\mathbf{A}$ then we can decompose the matrix ($\mathcal{O}(N^3)$) once and solve equation (5.4) for each column of $\mathbf{A}^{-1}$ which requires $N\times\mathcal{O}(N^2)$ operations i.e. also $\mathcal{O}(N^3)$. Thus taking an inverse costs $\mathcal{O}(N^3)$ operations. This can become very slow even on the fastest computers when $N>\mathcal{O}(10^3)$.

Once the matrix is LU decomposed the determinant is easy to calculate since

$$\det \mathbf{A} = \prod_{j=1}^{N} \beta_{jj} \tag{5.12}$$

(quoted here without proof). However this will quickly lead to overflows so it is usually calculated as

$$\ln(\det \mathbf{A}) = \sum_{j=1}^{N} \ln \beta_{jj} \quad . \tag{5.13}$$

5.2. Iterative Solution – Jacobi Method

Given that inverting matrices explicitly can be prohibitive even on the fastest computers we can use iterative methods to converge onto a solution from an initial guess. This is a method which is *guaranteed* to work if the matrix $\mathbf{A}$ is strictly **diagonally dominant**, i.e.,

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \quad , \tag{5.14}$$

for all rows i of the matrix. The Jacobi method can converge for a matrix which is not diagonally dominant, if the eigenvalues of its associated update matrix are suitable, as will be shown shortly. [1]

Sparse systems, where only a few variables appear in each equation, can often be rearranged into a diagonally dominant form such as a band diagonal matrix which looks like

$$\begin{pmatrix} \cdot & \cdot & & & & & & \\ \cdot & \cdot & \cdot & & & & & \\ & \cdot & \cdot & \cdot & & & 0 & \\ & & \cdot & \cdot & \cdot & & & \\ & 0 & & \cdot & \cdot & \cdot & & \\ & & & & \cdot & \cdot & \cdot & \\ & & & & & \cdot & \cdot & \cdot \end{pmatrix} . \tag{5.15}$$

We can rearrange $\mathbf{A}$ as the sum of its diagonal elements and its lower and upper triangles

$$\boxed{\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U} \quad .} \tag{5.16}$$

Note that the $\mathbf{L}$ and $\mathbf{U}$ matrices here are nothing to do with those from LU decomposition! (For a start, we are adding rather than multiplying to compose $\mathbf{A}$.) We then have

$$\mathbf{A} \cdot \vec{x} = (\mathbf{L} + \mathbf{D} + \mathbf{U}) \cdot \vec{x} = \vec{b},$$

giving

$$\mathbf{D} \cdot \vec{x} = -(\mathbf{L} + \mathbf{U}) \cdot \vec{x} + \vec{b}.$$

[1] In particular, matrices with $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$, i.e., some (but not all) rows not *strictly* diagonally dominant, will often work with the Jacobi method.

By multiplying by $\mathbf{D}^{-1}$ we can rearrange this to get

$$\vec{x} = -\mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U}) \cdot \vec{x} + \mathbf{D}^{-1} \cdot \vec{b}\,.$$

This looks like an iterative equation if we identify $\vec{x}$ on the left and right-hand sides with a new and old version respectively

$$\boxed{\vec{x}_{n+1} = -\mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U}) \cdot \vec{x}_n + \mathbf{D}^{-1} \cdot \vec{b}} \tag{5.17}$$

which we can use to find $\vec{x}$ starting from a guess $\vec{x}_0$. Equation (5.17) is the Jacobi method. In general this method will converge from any starting guess if all the eigenvalues of the update matrix $\mathbf{T} \equiv \mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U})$ satisfy $|\lambda^i| < 1$. [2] Introducing some new terminology, the **spectral radius** of $\mathbf{T}$ must be less than unity. The spectral radius ρ of a matrix $\mathbf{M}$ is defined as the largest modulus of any of its eigenvalues, i.e., $\rho(\mathbf{M}) = \max\{|\lambda^i|\}$. The condition $\rho(\mathbf{T}) < 1$ is guaranteed for any matrix $\mathbf{A}$ that is strictly diagonally dominant. [3] Note that the inverse of $\mathbf{D}$ is easy to calculate since it is a diagonal matrix. For any diagonal matrix $\mathbf{D} = \mathrm{diag}(d_1, d_2, \ldots\, d_N)$ the inverse is

$$\mathbf{D}^{-1} = \mathrm{diag}\left(\frac{1}{d_1}, \frac{1}{d_2}, \ldots, \frac{1}{d_N}\right) \quad . \tag{5.18}$$

Proof of Jacobi convergence condition – The condition $|\lambda^i| < 1$ is simple to prove, using similar principles to those used previously in section 3.4 for the stability analysis of finite difference solution of a set of coupled linear ODEs. As before, we consider the errors at each iteration

$$\vec{\epsilon}_n = \vec{x}_n - \vec{x} \qquad \text{and} \qquad \vec{\epsilon}_{n+1} = \vec{x}_{n+1} - \vec{x}\,, \tag{5.19}$$

where $\vec{x}$ is the exact solution. For convergence, we need $\vec{\epsilon}_n$ to vanish as $n \to \infty$. Using these expressions for the errors and the Jacobi iteration formula, we can show that

$$\begin{aligned} \vec{\epsilon}_{n+1} = \vec{x}_{n+1} - \vec{x} &= -\mathbf{T} \cdot \vec{x}_n + \mathbf{D}^{-1} \cdot \vec{b} - \left(-\mathbf{T} \cdot \vec{x} + \mathbf{D}^{-1} \cdot \vec{b}\right) \quad , \\ &= -\mathbf{T} \cdot \vec{x}_n + \mathbf{T} \cdot \vec{x} \quad , \\ &= -\mathbf{T} \cdot (\vec{x}_n - \vec{x}) \quad , \\ \therefore \qquad \vec{\epsilon}_{n+1} &= -\mathbf{T} \cdot \vec{\epsilon}_n \quad . \end{aligned} \tag{5.20}$$

If $\mathbf{T}$ is a diagonalisable matrix so that it can be factorised using the eigen-decomposition

$$\mathbf{T} = \mathbf{R} \cdot \mathbf{\Lambda} \cdot \mathbf{R}^{-1} \quad \text{with} \quad \mathbf{\Lambda} \equiv \mathrm{diag}(\lambda^1, \lambda^2, ..., \lambda^N)\,, \tag{5.21}$$

then we can map to a 'rotated' error $\vec{\zeta}_n = \mathbf{R}^{-1} \cdot \vec{\epsilon}_n$ such that

$$\vec{\zeta}_{n+1} = \mathbf{\Lambda} \cdot \vec{\zeta}_n \quad , \tag{5.22}$$

[2] Note that the eigenvalues of a real valued matrix can be complex.

[3] If $\mathbf{A}$ is not diagonally dominant then the spectral radius of its Jacobi update matrix will need to be checked, to determine whether or not the Jacobi method will converge with $\mathbf{A}$.

and therefore the components of ζ are uncoupled. This allows us to impose a convergence criterion on each eigenvalue individually since

$$\left|\frac{\zeta^i_{n+1}}{\zeta^i_n}\right| \equiv \left|\lambda^i\right| < 1 \quad . \tag{5.23}$$

(Note that eigen-decomposition is just used here to prove the requirements for Jacobi iteration to converge; we don't actually have to explicitly eigen-decompose $\mathbf{T}$ (or anything else) to implement the Jacobi method!) What we see is that repeated application of the Jacobi iteration 'erodes' away the error vector (present in our initial guessed solution $\vec{x}_0$). The slowest possible rate of erosion, and therefore the slowest rate of convergence to the solution of $\mathbf{A}\cdot\vec{x} = \vec{b}$, is governed by the spectral radius of the Jacobi update matrix.

Benefits of iterative methods

Firstly, they are faster than the direct inversion methods (such as LU decomposition), especially for sparse matrix equations. This is because we only need to carry out a matrix–vector multiplication (i.e. $\mathbf{T}\cdot\vec{x}_n$) at each step. (The $\mathbf{D}^{-1}\cdot\vec{b}$ calculation need only be done once, then added on at each step which carries little extra computational cost in comparison to the multiplication.) The total number of operations needed to obtain the solution will be $\mathcal{O}(N^k \times N_{\text{iter}})$ where N_{iter} is the number of iterations required to reach a chosen level of convergence (fractional change in solution is less than a chosen tolerance) and N^k accounts for multiplication. For a sparse matrix the number of non-zero elements can be a few N (i.e. $3N-2$ for a tri-diagonal matrix) so that $k=1$, i.e., $\mathbf{T}\cdot\vec{x}_n$ takes $\mathcal{O}(N)$ operations. For a full matrix, $k=2$. As long as N_{iter} is substantially less than N, iterative solution scales much better than $\mathcal{O}(N^3)$ needed for direct inversion methods, even for full matrix. (To be fair, LU decomposition can be carried out faster then $\mathcal{O}(N^3)$ for a band diagonal matrix by using a bespoke algorithm tailored to the specific structure of that particular matrix.) The key thing with iterative methods is to get quick convergence so that N_{iter} is as small as possible.

A second advantage of iterative methods is that in practice, they often yield more accurate solutions than exact (i.e. direct) methods because they are much less susceptible to round-off error. This is especially true the larger the matrix.

Checking for convergence

Typically, one checks that the fractional change in the norm of the solution vector

$$\varepsilon = \left|\frac{\|\vec{x}_{n+1}\| - \|\vec{x}_n\|}{\|\vec{x}_n\|}\right| \quad , \tag{5.24}$$

is below a specified tolerance; something a few orders of magnitude above the fractional round-off error due to finite precision arithmetic (e.g. $\varepsilon_{tol} \sim 100\times 10^{-16} \sim 10^{-14}$). There are many measures of the norm of a vector; the general ℓ-norm is

$$\|\vec{x}\|_\ell \equiv \left(\sum_{i=1}^{n} |x_i|^\ell\right)^{\frac{1}{\ell}} \quad . \tag{5.25}$$

Commonly the $\ell = 1$ (sum of $|x_i|$), the $\ell = 2$ (the familiar Euclidean sum of squares) or the $\ell = \infty$ (the largest component of the vector!) are used.

Another way to check for convergence is to monitor the **residual** which is $\vec{r} = \mathbf{A} \cdot \vec{x}_{n+1} - \vec{b}$ which gives us $\mathbf{A} \cdot \vec{\epsilon}_{n+1}$, i.e., the matrix acting on the error vector. One would then monitor $\varepsilon = \|\vec{r}\|/\|\vec{b}\|$. This is more costly to do every iteration.

5.3. Iterative Gauss-Seidel Method

An alternative method which converges faster than the Jacobi method is to take

$$\boxed{\vec{x}_{n+1} = -(\mathbf{L}+\mathbf{D})^{-1} \cdot \mathbf{U} \cdot \vec{x}_n + (\mathbf{L}+\mathbf{D})^{-1} \cdot \vec{b} \ ,} \tag{5.26}$$

where the update matrix is now $\mathbf{T} = (\mathbf{L}+\mathbf{D})^{-1} \cdot \mathbf{U}$. This algorithm is derived similarly to the Jacobi method, except that $\mathbf{U} \cdot \vec{x}$ (rather than $(\mathbf{L}+\mathbf{U}) \cdot \vec{x}$) is moved over to the RHS to be with $\vec{b}$. The improved convergence speed stems from the fact that the spectral radius of the Gauss-Seidel update matrix is less than that of the Jacobi update matrix (for a given $\mathbf{A}$), i.e., $\rho(\mathbf{T}_{GS}) < \rho(\mathbf{T}_J)$.

At first glance it looks like we have to calculate $(\mathbf{L}+\mathbf{D})^{-1}$, so that Gauss-Seidel (G-S) would seem to be more computationally costly than Jacobi. However it turns out that the G-S iteration can be carried out without this inverse if we use **forward substitution**, thanks to the shape of the matrix $\mathbf{L}$. We rewrite (5.26) as

$$\mathbf{D} \cdot \vec{x}_{n+1} = -\mathbf{L} \cdot \vec{x}_{n+1} - \mathbf{U} \cdot \vec{x}_n + \vec{b} \ .$$

We then solve for each component $x_k^{(n+1)}$ of $\vec{x}_{n+1}$ in turn, starting with $k = 1$ and sequentially increasing until $k = N$. This is the same approach used to solve $\mathbf{L} \cdot \vec{y} = \vec{b}$ when using LU decomposition. (See equation (5.9) .) In this form G-S looks like

$$x_k^{(n+1)} = \frac{1}{D_{kk}} \left(-\sum_{j=1}^{k-1} L_{kj} x_j^{(n+1)} - \sum_{j=k+1}^{N} U_{kj} x_j^{(n)} + b_k \right) \quad , \quad k = 1, 2, \ldots, N \ . \tag{5.27}$$

The components of $\vec{x}_{n+1}$ must be calculated in the order shown.

5.4. Iterative Successive Over-Relaxation Method

Abbreviated to SOR, this can be thought of as a modified version of Gauss-Seidel with superior speed of convergence

$$\boxed{\vec{x}_{n+1} = (\omega\mathbf{L}+\mathbf{D})^{-1} \cdot \left(-[\omega\mathbf{U} + (\omega-1)\mathbf{D}] \cdot \vec{x}_n + \omega\vec{b} \right) \ ,} \tag{5.28}$$

where ω is the **relaxation parameter**. A high speed of convergence can be achieved by tuning ω, which affects the spectral radius of SOR's update matrix. With $\omega = 1$ SOR reverts to Gauss-Seidel. With $1 < \omega \leq 2$ SOR gives faster convergence than G-S. However, the optimum value of ω is problem specific. In simple cases it can be determined analytically (not covered in this course!), otherwise it must be found by trial and error. For $\omega > 2$ SOR fails. $\omega < 1$ corresponds to under-relaxation, which is not beneficial to speed of convergence.

The above SOR iteration can be implemented using forward substitution in a similar way to Gauss-Seidel, avoiding the need to explicitly compute $(\omega\mathbf{L}+\mathbf{D})^{-1}$.

There are even faster converging iterative matrix solvers than SOR, which are based on iterative optimisation methods. These will be briefly discussed in section 7.3.

5.5. Largest Eigenvalue of a Matrix

We have seen how finding the largest eigenvalue of a matrix would be useful in checking convergence criteria. The **method of powers** is a simple method to approximate the largest eigenvalue of any non-singular $N \times N$ matrix with N linearly independent eigenvectors. It also yields the associated eigenvector.

Consider a system of the type

$$\mathbf{A}\cdot\vec{x} = \lambda\,\vec{x}\,, \tag{5.29}$$

with $\mathbf{A}$ an $N \times N$ matrix. In general if $\mathbf{A}$ is non-singular (i.e. $\det\mathbf{A} \neq 0$) it will have N distinct eigenvalues (λ_i) [4] with associated linearly independent eigenvectors $(\vec{e}_i)$

$$\mathbf{A}\cdot\vec{e}_i = \lambda_i\,\vec{e}_i\,. \tag{5.30}$$

The linearly independent eigenvectors form a basis in which any vector (say $\vec{v}$) can be expanded

$$\vec{v} = \sum_{i=1}^{N} c_i\,\vec{e}_i\,. \tag{5.31}$$

We start with a random choice of vector $\vec{v}$ and act on it with $\mathbf{A}$ a number of times

$$\begin{aligned}
\mathbf{A}\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\mathbf{A}\cdot\vec{e}_i = \sum_{i=1}^{N} c_i\,\lambda_i\,\vec{e}_i \ , \\
\mathbf{A}\cdot\mathbf{A}\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\lambda_i^2\,\vec{e}_i \ , \\
\mathbf{A}^n\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\lambda_i^n\,\vec{e}_i \ , \\
\therefore \quad \lim_{n\to\infty}\ \mathbf{A}^n\cdot\vec{v} &\to c_j\,\lambda_j^n\,\vec{e}_j \ ,
\end{aligned} \tag{5.32}$$

where λ_j is the largest eigenvalue. Since any multiple of an eigenvector is still an eigenvector itself we have found the largest eigenvector. We can normalise the vector we have just found as

$$\boxed{\vec{\omega}_j = \frac{\mathbf{A}^n\cdot\vec{v}}{|\mathbf{A}^n\cdot\vec{v}|} \equiv \frac{\vec{e}_j}{|\vec{e}_j|} \ ,} \tag{5.33}$$

such that $\vec{\omega}_j^{\,T}\cdot\vec{\omega}_j = 1$. (Here $\vec{\omega}_j^{\,T}$ is the transpose of vector $\vec{\omega}_j$.)

[4] Note the change in notation here for eigenvalues! λ_i is now being used in place of λ^i, to accommodate the need to raise eigenvalues to powers.

Then it is simple to calculate the eigenvalue itself since

$$\boxed{\vec{\omega}_j^{\,T} \cdot \mathbf{A} \cdot \vec{\omega}_j = \vec{\omega}_j^{\,T} \cdot (\lambda_j \vec{\omega}_j) = \lambda_j \ .} \tag{5.34}$$

Smallest eigenvalue

We can also use the method of powers to find the smallest eigenvalue of the system, by using $\mathbf{A}^{-1}$ instead of $\mathbf{A}$ in equations (5.33) and (5.34). This works because, the inverse of a matrix has the same eigenvectors but with the inverse eigenvalues. This can be shown since

$$\vec{e}_i = \mathbf{A}^{-1} \cdot \mathbf{A} \cdot \vec{e}_i = \mathbf{A}^{-1} \cdot (\lambda_i \, \vec{e}_i) = \lambda_i \, \mathbf{A}^{-1} \cdot \vec{e}_i \quad ,$$

thus

$$\mathbf{A}^{-1} \cdot \vec{e}_i = (\lambda_i)^{-1} \, \vec{e}_i \ .$$

So using the power method on $\mathbf{A}^{-1}$ finds its largest eigenvalue which will be the smallest eigenvalue of the original $\mathbf{A}$.

5.6. Other Eigenvalues

There are many methods for finding the remaining eigenvalues of a matrix but most use the **shift method** by finding the eigenvalue closest to a given value α. To see this define the shifted matrix

$$\mathbf{A}' \equiv \mathbf{A} - \alpha \, \mathbf{I} \, , \tag{5.35}$$

such that

$$\mathbf{A}' \cdot \vec{e}_i = \mathbf{A} \cdot \vec{e}_i - \alpha \, \mathbf{I} \cdot \vec{e}_i = \lambda_i \, \vec{e}_i - \alpha \, \vec{e}_i = (\lambda_i - \alpha) \, \vec{e}_i \, . \tag{5.36}$$

So the shifted matrix has the same eigenvectors but shifted eigenvalues, $\lambda'_i = \lambda_i - \alpha$. Thus finding the largest eigenvalue of $(\mathbf{A}')^{-1}$ (e.g. by the power method) will give the the smallest of $\mathbf{A}'$ and thus the eigenvalue of $\mathbf{A}$ closest to the value α. But how to get $(\mathbf{A}')^{-1}$? In principle, we can solve equation $\mathbf{A}' \cdot \mathbf{X} = \mathbf{I}$ for $\mathbf{X}$ using an efficient method like Jacobi or Gauss-Seidel (or LU decomposition for a relatively small matrix). Then $\mathbf{X}$ will be $(\mathbf{A}')^{-1}$ that we seek.

A crucial point is then to chose a suitable value for α. It turns out that the values of the diagonal elements are good choices, particularly if $\mathbf{A}$ is diagonally dominant. This follows from **Gerschgorin's Theorem** which states that for any eigenvalue λ_i the following inequality is satisfied

$$|\lambda_i - a_{ii}| \leq \sum_{j \neq i} |a_{ij}| \ , \tag{5.37}$$

i.e. the eigenvalue lies within a circle/disk in the complex plane of radius $\sum_{j \neq i} |a_{ij}|$ centred on the value of the diagonal element a_{ii}. [5] Equivalently, each 'Gerschgorin disc' will have an eigenvalue in it (and possibly more than one if discs overlap and if eigenvalues happen to fall on such overlaps). This is particularly useful if $\mathbf{A}$ is diagonally

[5] The role of 'i' in equation (5.37) needs careful qualification. 'i' specifies a certain row of the matrix as would be expected, but which of the N eigenvalues is λ_i? It turns out that λ_i belongs to the eigenvector with element i being the largest in magnitude (in that eigenvector).

dominant since the radius will be small and therefore the values of the diagonal elements will be close to the eigenvalues, allowing the algorithm to work quickly and efficiently. For example take the following matrix [6]

$$\mathbf{A} \equiv \begin{pmatrix} 1 & 0.1 & 0 \\ 0.1 & 5 & 0.2 \\ 0.1 & 0.3 & 10 \end{pmatrix} . \tag{5.38}$$

We then have that $0.9 \leq \lambda_1 \leq 1.1$, $4.7 \leq \lambda_2 \leq 5.3$, or $9.6 \leq \lambda_3 \leq 10.4$. We can then find the exact values by setting α to 1, 5, or 10 in the shift method.

Finally, it is worth noting that Gerschgorin's Theorem provides a convenient way of assessing upper bounds for the spectral radius of the update matrices of the iterative solvers seen earlier (sections 5.2, 5.3, 5.4), and thus determining the suitability of a matrix $\mathbf{A}$ for solution by, e.g., Jacobi iteration.

[6] To make life easier I have chosen one where the ranges of each row do not overlap.

SECTION 6
Boundary Value Problems

Outline of Section

- Boundary Value Problems
- Shooting Method
- Finite differences
- Eigensystems

Consider the general, linear, second order equation

$$\alpha \frac{d^2y}{d\eta^2} + \beta \frac{dy}{d\eta} + \gamma\, y = k\,. \tag{6.1}$$

Normally we would require initial conditions specifying the value of the solution at some initial point $\eta = a$, i.e., $y(a)$ and its derivative $y'(a)$, to integrate the system to a final point.

But what if we want to find the solution that matches an initial and final condition, e.g., $y(a)$ and $y(b)$? Since we are giving two conditions and we have two degrees of freedom (second order ODE) we should be able to find a consistent solution.

6.1. Shooting Method

A first method we can use is the **shooting method** where we start at a with $y(a) = A$, make a guess for $y'(a) = C_1$ and find a solution $y_1(\eta)$. This can be obtained via a finite difference method (from section 4), or algebraically if we are lucky. In general the solution will not end at the desired point $y(b) = B$ say, but will end instead at $y_1(b) = B_1$.

We then make another guess starting from the same point $y(a) = A$ but with $y'(a) = C_2$ and find a new solution $y_2(\eta)$ which ends at $y_2(b) = B_2$. Since the system is **linear** a sum of two solutions will itself be a solution of the system so we can introduce a weighted average of y_1 and y_2 as a new solution

$$\boxed{y_c(\eta) = c\, y_1(\eta) + (1-c)\, y_2(\eta)\ \ ,} \tag{6.2}$$

where c is an unknown constant which makes y_c the required solution which ends at $y(b) = B$.

We can check that

$$y_c(a) = c\,A + (1-c)\,A = A\,,$$

as required and the condition that

$$y_c(b) = c\,B_1 + (1-c)\,B_2 = B\,,$$

gives us a solution for c

$$\boxed{c = \frac{B - B_2}{B_1 - B_2}\ .} \tag{6.3}$$

So finding the required solution to a linear system is relatively straightforward using the shooting method. However it still requires solving the system twice to get a single solution. Next we will look at another method which solves the system explicitly via matrix methods.

6.2. Finite Difference Method

Consider the system

$$\frac{d^2y}{d\eta^2} = k\,. \tag{6.4}$$

We can re-write the second derivative as a central difference and discretise the solution with m intervals of size h as in Section 2.2 (see also Problem Sheet 1)

$$\frac{y_{i-1} - 2\,y_i + y_{i+1}}{h^2} = k\,. \tag{6.5}$$

Then, since we have $y_0 = A$ and $y_m = B$, the system will look like

$$\begin{aligned} A - 2\,y_1 + y_2 &= k\,h^2\,,\\ y_1 - 2\,y_2 + y_3 &= k\,h^2\,,\\ &\ \ \vdots\\ y_{m-2} - 2\,y_{m-1} + B &= k\,h^2\,, \end{aligned} \tag{6.6}$$

i.e. $m-1$ linear equations. The system can be written in matrix form as

$$\begin{pmatrix} -2 & 1 & 0 & 0 & .... & 0 & 0\\ 1 & -2 & 1 & 0 & .... & 0 & 0\\ & & & \vdots & & & \\ 0 & 0 & 0 & 0 & .... & 1 & -2 \end{pmatrix} \begin{pmatrix} y_1\\ y_2\\ \vdots\\ y_{m-1} \end{pmatrix} = \begin{pmatrix} k\,h^2 - A\\ k\,h^2\\ \vdots\\ k\,h^2 - B \end{pmatrix}, \tag{6.7}$$

i.e. $\mathbf{M}\cdot\tilde{\vec{y}} = \vec{b}$. The '$\sim$' is used to denote that the solution is approximate (compared to the true solution of eqn (6.4) at the sample points), because of the use of finite difference approximation to derivatives. The matrix $\mathbf{M}$ is close to being diagonally dominated (since it has $|m_{ii}| \geq \sum_{j\neq i} |m_{ij}|$) and is in fact suitable for solving using the Jacobi, Gauss-Seidel and SOR methods introduced in Sections 5.2, 5.3 and 5.4. This is because

the spectral radius of the associated update matrix (for each method) is less than unity. Figure 6.1 illustrates the finite difference method applied to solving a boundary value problem.

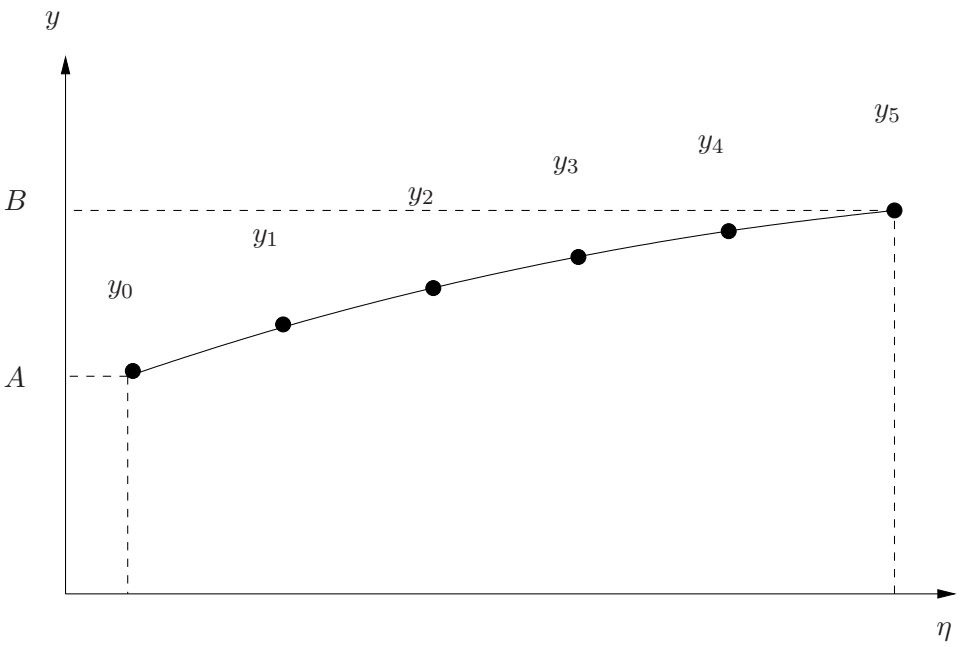

Figure 6.1. Finite difference method, for solution of boundary value problem. Example of discrete points with $m = 5$.

6.3. Derivative Boundary Conditions

On occasion we know the derivative at one of the boundaries but not the initial condition for the variable. Considering the same system as in (6.4) we could have boundary conditions

$$y'(a) = C \quad \text{and} \quad y(b) = B\,. \tag{6.8}$$

We can adapt the finite difference method by taking a central difference for the derivative at the boundary

$$y'_0(a) \approx \frac{y_1 - y_{-1}}{2h} = C\,, \tag{6.9}$$

and extending the system by one interval so that the first equation reads

$$y_{-1} - 2\,y_0 + y_1 = k\,h^2\,, \tag{6.10}$$

which, given (6.9), is

$$-\,2\,y_0 + 2y_1 = k\,h^2 + 2\,h\,C\,. \tag{6.11}$$

The system can then be written as m linear equations

$$\begin{pmatrix} -2 & 2 & 0 & 0 & .... & 0 & 0 \\ 1 & -2 & 1 & 0 & .... & 0 & 0 \\ & & & . & & & \\ & & & . & & & \\ & & & . & & & \\ 0 & 0 & 0 & 0 & .... & 1 & -2 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ . \\ . \\ . \\ y_{m-1} \end{pmatrix} = \begin{pmatrix} k\,h^2 + 2\,h\,C \\ k\,h^2 \\ . \\ . \\ . \\ k\,h^2 - B \end{pmatrix} . \tag{6.12}$$

6.4. Eigenvalue Problems

For the particular case where the differential equations are homogeneous and linear we can view the problem as an eigensystem. For example, consider the wave equation

$$\frac{d^2y}{dx^2} + k^2 y = 0\,, \tag{6.13}$$

with boundary conditions $y(0) = 0$ and $y(1) = 0$. This describes the vibrations on a string of length 1 and fixed at the endpoints. The general solution to this system is easily found to be

$$y(x) = A\,\sin(k\,x) + B\,\cos(k\,x)\,. \tag{6.14}$$

The boundary conditions imply that $B = 0$ and that $k = \pm n\,\pi$. The solution describes fundamental modes of vibrations on the string where $n = 1, 2, 3, \ldots$, etc.

The system (and more complicated ones in particular) can be solved using finite differences

$$\frac{y_{i-1} - 2\,y_i + y_{i+1}}{h^2} + k^2\,y_i = 0\,, \tag{6.15}$$

which gives the eigensystem

$$\begin{pmatrix} +2 & -1 & 0 & 0 & .... & 0 & 0 \\ -1 & +2 & -1 & 0 & .... & 0 & 0 \\ & & & . & & & \\ & & & . & & & \\ & & & . & & & \\ 0 & 0 & 0 & 0 & .... & -1 & +2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_{m-1} \end{pmatrix} = h^2\,k^2 \begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_{m-1} \end{pmatrix} , \tag{6.16}$$

i.e.

$$\mathbf{A} \cdot \tilde{\vec{y}} = \lambda \tilde{\vec{y}} \tag{6.17}$$

where $\lambda = h^2\,k^2$ are the eigenvalues of the system. One then finds $\mathbf{A}$'s eigenvalues λ_i and eigenvectors $\vec{e}_i$ using a suitable numerical eigen-solver. These should be close numerical approximations to the eigenvalues and eigenfunctions of the original ODE eigen-problem. It is often useful to find just the eigenvector corresponding to the largest (or smallest) eigenvalues, as these define the fundamental modes of the system. The power method (section 5.5) can be used to readily find these.

SECTION 7
Minimisation and Maximisation of Functions

Outline of Section

- One–dimensional minimisations
- Multi-dimensional minimisations
- Using minimisation to solve matrix equations

This section concerns **iterative methods** for finding local or global minima (maxima) of a function of one variable $f(x)$ and of many independent variables $f(x_1, x_2, \ldots, x_N)$, written more succinctly as $f(\vec{x})$ where $\vec{x}$ is an N-dimensional vector. Both the position of a minimum $\vec{x}_\star$ and the value of the function there $f(\vec{x}_\star)$ may be of interest. The function can be non-linear. The methods covered will find minima; maxima can be found by applying these methods to $F(\vec{x}) = -f(\vec{x})$. Root-finding of non-linear functions $f(x)$ was introduced in section 2.5. In principle roots of a non-linear function involving many variables $f(\vec{x})$ can be found by finding minima $\vec{x}_\star$ of $|f(\vec{x})|^2$ where $f(\vec{x}_\star) = 0$.

Interestingly, it turns out that solving a matrix equation can be formulated as a minimisation problem, as we shall see later. Some of the most efficient iterative matrix solvers are based on this approach and converge much faster than Jacobi, Gauss-Seidel and SOR.

We will deal with **unconstrained** minimisation here. ***Constrained*** optimisation exists, but will not be covered. We will also limit ourselves to real valued, scalar functions $f(\vec{x}) \in \mathbb{R}$ and real valued variables; $x \in \mathbb{R}$ and $\vec{x} \in \mathbb{R}^N$.

One setting where finding the minimum of a complicated non-linear function occurs is the optimisation of parameters in a model when fitting to data. Consider the observed data $d_i^{\rm obs}$ with errors σ_i in figure 7.1. The model we would like to fit is a linear one

$$d_i^{\rm model} = a + \alpha\, X_i \,. \tag{7.1}$$

The normal procedure is to find the minimum χ^2 with respect to the parameters a and α

$$\chi^2(a,\alpha) = \sum_{i=1}^{N^{\rm data}} \frac{\left(d_i^{\rm obs} - d_i^{\rm model}\right)^2}{\sigma_i^2} \,. \tag{7.2}$$

The assumption here is that the data is distributed as independent Gaussian random numbers and we are **maximising the likelihood**

$$L(a,\alpha) \propto e^{-\frac{1}{2}\chi^2(a,\alpha)}\,, \tag{7.3}$$

which is equivalent to finding the minimum in the χ^2. The result of the minimisation may look something like figure 7.2.

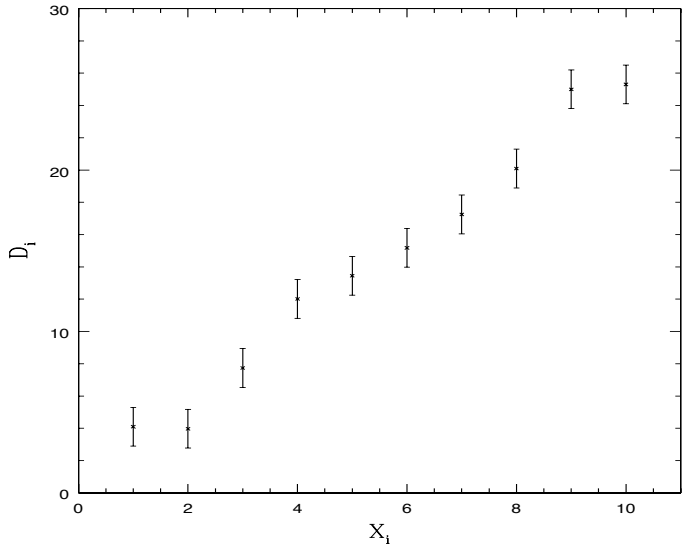

Figure 7.1. Some data with errors.

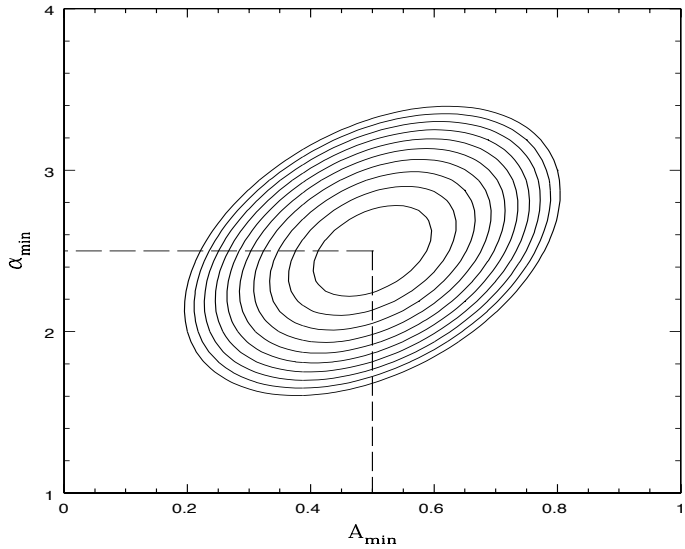

Figure 7.2. Contour plot of the χ^2 around the minimum. $a_{\min}$ and $\alpha_{\min}$ are the maximum likelihood values for the parameters in the model. The curvature around the minimum is related to the error in the parameters.

7.1. One–dimensional minimisation

Naively we might think that finding the minima of a function is trivial. All we need to do is differentiate the function and find the roots of the resulting equation. However there are many cases where the analytical method is not applicable, e.g., where the derivative in the function is discontinuous or the function has a regular (possibly infinite) set of minima. For a function of many, many variables, it can be too computationally costly to evaluate $\partial f/\partial x_i$ for all variables or simply too laborious to implement them all in code!

In practice this is often the case when evaluating a function of some scattered data where the analytical dependence of the likelihood with respect to the parameters is not known *a priori* and must be evaluated numerically. In this case we have to choose between a method that searches for minima/maxima using just the function values or (more optimal) ones that also use approximations for the derivatives of the function.

Parabolic Method

Functions almost always approximate a parabola near a minimum (except for some pathological cases). A very simple method to find a local minimum of a function exploits this. The method works when the curvature of $f(x)$ is positive everywhere in the interval we are looking at. The parabolic method consists of selecting three points along the function $f(x)$, e.g., x_0, x_1, and x_2 where

$$f(x_0) = y_0, \quad f(x_1) = y_1, \quad \text{and} \quad f(x_2) = y_2 \tag{7.4}$$

and then fitting $P_2(x)$, the 2nd-order Lagrange polynomial, through the points (see section 2.6). The location of the minimum of the interpolating parabola $P_2(x)$ is found using equation (7.6) below; this value is x_3. This procedure is illustrated in fig. 7.3. We then keep the three lowest points out of $f(x_0)$, $f(x_1)$, $f(x_2)$, and $f(x_3)$ and repeat the procedure. At each successive iteration the point x_3 will converge towards $x_\star$ where the minimum of the function $f(x)$ is located. The iterations can be stopped once the change in x_3 is less than a desired value.

The quadratic interpolating the 3 points is given by

$$P_2(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}y_0 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}y_1 + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}y_2 \ , \tag{7.5}$$

[c.f. equation (2.37)]. We want to find the minimum of $P_2(x)$, which will be a better estimate of the minimum's position (than the middle point of our trio). Differentiating $P_2(x)$ gives

$$\frac{dP_2}{dx} = \frac{[(x-x_1)+(x-x_2)]\,(x_2-x_1)}{d}y_0 + \frac{[(x-x_0)+(x-x_2)]\,(x_0-x_2)}{d}y_1 + \frac{[(x-x_0)+(x-x_1)]\,(x_1-x_0)}{d}y_2 \ ,$$

where $d = (x_1-x_0)(x_2-x_0)(x_2-x_1)$. Then setting $dP_2/dx = 0$ and solving for x gives (after some algebra) a new estimate of the minimum x_3

$$\boxed{x_3 = \frac{1}{2}\,\frac{(x_2^2-x_1^2)y_0 + (x_0^2-x_2^2)y_1 + (x_1^2-x_0^2)y_2}{(x_2-x_1)y_0 + (x_0-x_2)y_1 + (x_1-x_0)y_2}} \ . \tag{7.6}$$

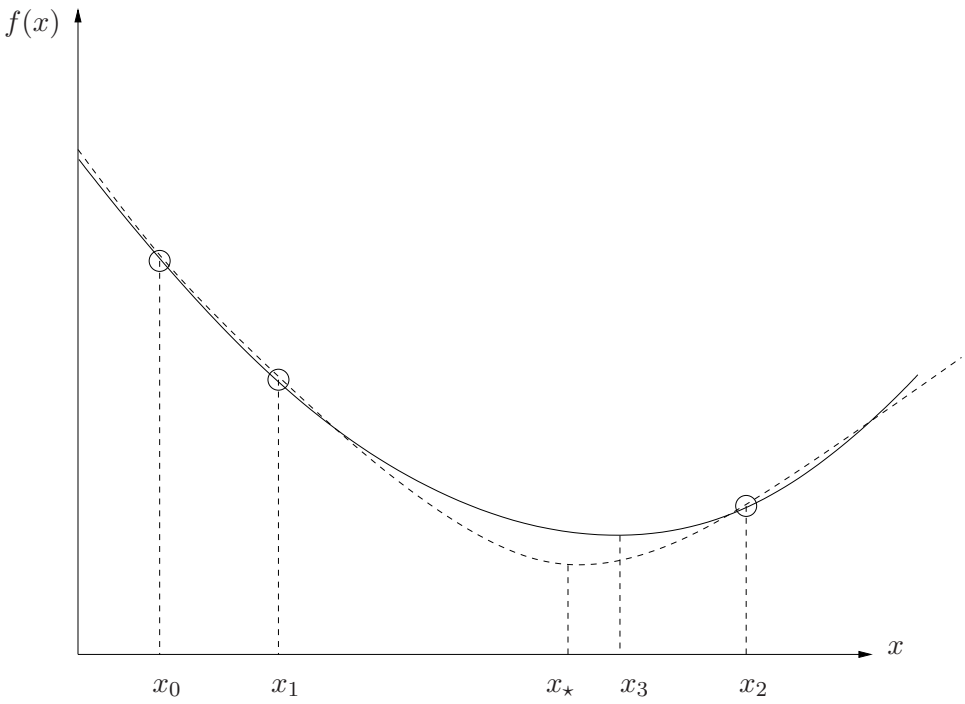

Figure 7.3. The parabolic minimum search. The function being minimised $f(x)$ is the dashed curve. A parabola (solid curve) is fit to the three points at x_0, x_1, x_2 and the minimum $x = x_3$ is found. Then the highest of the four points x_0, x_1, x_2, and x_3 is discarded and the method repeated. The minima of successive parabola approximations converge to the minimum of the function $(x_\star, f(x_\star))$.

This method works because any minimum can be approximated by a parabola and the approximation becomes more and more accurate as you get closer to the minimum.

For the case where the curvature is not positive throughout the initial interval we can first evaluate the function at two points inside the interval. We then keep the interval which includes the lowest point and then use the parabolic method to find the minimum within the new interval.

7.2. Multi-Dimensional methods

Univariate method

For the case where the function is multi-dimensional, i.e., $f(\vec{x})$, we can extend the one-dimensional parabolic method. This can be done by searching for the minimum in each direction successively and iterating. However this is a very inefficient method which is not useful in most cases. Univariate search is illustrated by the black arrows in figure 7.4. Contours are shown for a 2D function $f(x, y)$. The univariate search spirals into the minimum, converging slowly.

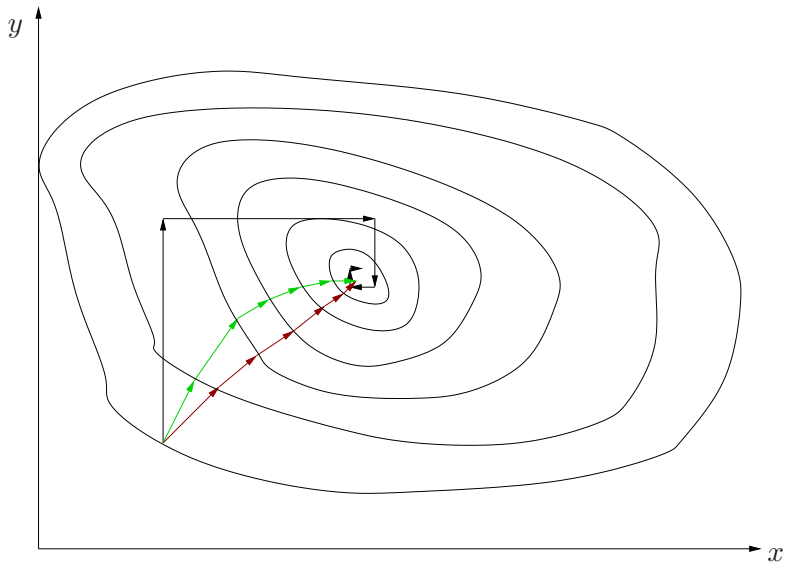

Figure 7.4. Multi-dimensional minimisation - for a function $f(x, y)$. Three methods are depicted: univariate method (black arrows), gradient method (green arrows) and Newton's method (red arrows). [Nb: this is not quite accurate; the green arrows are all supposed to be perpendicular to the contours!]

Gradient method

We can follow the steepest descent towards the minimum. This is achieved by calculating the local gradient and following its (negative) direction. The gradient of $f(\vec{x})$ at point $\vec{x}_n$ is given by the vector

$$\vec{d}(\vec{x}_n) = \vec{\nabla} f(\vec{x}_n) \qquad \text{where} \qquad \vec{\nabla} f = \begin{bmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_2 \\ \vdots \\ \partial f/\partial x_N \end{bmatrix} \tag{7.7}$$

and is perpendicular to the local contour line. Note that the notation $\vec{\nabla} f(\vec{x}_n)$ is equivalent to $\vec{\nabla} f|_{\vec{x}_0}$, i.e., $\vec{\nabla} f$ evaluated at $\vec{x} = \vec{x}_0$. We can take a small step in the direction **opposite** to gradient, i.e.,

$$\boxed{\vec{x}_{n+1} = \vec{x}_n - \alpha\, \vec{d}(\vec{x}_n) \ ,} \tag{7.8}$$

where $\alpha \ll 1$ and positive. If α is small enough then

$$f(\vec{x}_{n+1}) < f(\vec{x}_n) \ .$$

We can iterate this to find the minimum. (See green arrows in fig. 7.4.)

The gradient $\vec{\nabla} f$ can be found analytically or using a finite difference approximation; e.g. via a FDS applied in each variable x_i for $i = 1, \ldots, N$,

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, x_2, \ldots, x_i + \Delta, \ldots, x_N) - f(x_1, x_2, \ldots, x_i, \ldots, x_N)}{\Delta} \quad . \tag{7.9}$$

Newton's method

A more efficient method to find the minimum is achieved by including the local curvature at each step. Consider starting at a location $\vec{x}_0$. The minimum is displaced $\vec{\delta}$ away, at position $\vec{x_0} + \vec{\delta}$. How to estimate $\vec{\delta}$?

We use the Taylor series for the function about $\vec{x}$. (Later, $\vec{x}$ will be set to $\vec{x}_0$.)

$$f(\vec{x} + \vec{\delta}) = f(\vec{x}) + [\vec{\nabla} f(\vec{x})]^T \cdot \vec{\delta} + \frac{1}{2} \vec{\delta}^{\,T} \cdot \mathbf{H}(\vec{x}) \cdot \vec{\delta} + \mathcal{O}(|\vec{\delta}|^3) \quad , \tag{7.10}$$

where **H** is the **Hessian** or **Curvature** matrix (an $N \times N$ matrix)

$$H_{ij}(\vec{x}) = \frac{\partial^2 f(\vec{x})}{\partial x_i \partial x_j} \quad . \tag{7.11}$$

(c.f. equation (2.4) in section 2.1.) Since by definition $\vec{x} + \vec{\delta}$ is the location of the minimum, we must have $\vec{\nabla} f(\vec{x} + \vec{\delta}) = \vec{0}$. To actually be a minimum (rather than a maximum) the Hessian needs to be **positive definite**, i.e.,

$$\vec{\delta}^{\,T} \cdot \mathbf{H} \cdot \vec{\delta} > 0 \qquad (\text{for } \textit{all } \vec{\delta} \neq \vec{0}\,) \quad . \tag{7.12}$$

To determine $\vec{\delta}$ we can use the Taylor expansion (7.10) to 1st-order in $\vec{\delta}$ and take its gradient yielding

$$\begin{aligned} \vec{\nabla} f(\vec{x} + \vec{\delta}) &= \vec{\nabla} f(\vec{x}) + \vec{\nabla} \left([\vec{\nabla} f(\vec{x})]^T \cdot \vec{\delta} \right) = \vec{0} \quad , \\ &= \vec{\nabla} f(\vec{x}) + \mathbf{H}(\vec{x}) \cdot \vec{\delta} = \vec{0} \quad . \end{aligned}$$

Since we start at $\vec{x} = \vec{x}_0$ we can solve the following matrix-equation for $\vec{\delta}$,

$$\boxed{\mathbf{H}(\vec{x}_0) \cdot \vec{\delta} = -\vec{\nabla} f(\vec{x}_0) \quad .} \tag{7.13}$$

The notation $\mathbf{H}(\vec{x}_0)$ means (7.11) evaluated at $\vec{x} = \vec{x}_0$. If $f(\vec{x})$ is exactly 'parabolic', i.e.,

$$f(\vec{x}) = \vec{x}^{\,T} \cdot \mathbf{A} \cdot \vec{x} + \vec{b}^{\,T} \cdot \vec{x} + c \quad , \tag{7.14}$$

then there are no terms $\mathcal{O}(|\vec{\delta}|^3)$ in the Taylor expansion, and $\vec{x}_1 = \vec{x}_0 + \vec{\delta}$ is the exact position of the minimum.

If the function is not well approximated by a quadratic then the estimate of $\vec{\delta}$ obtained from solving (7.13) will not take us directly to the minimum so we iterate this until we get arbitrarily close to it, i.e., $\vec{x}_{n+1} = \vec{x}_n + \vec{\delta}$ which can be written as

$$\boxed{\vec{x}_{n+1} = \vec{x}_n - [\mathbf{H}(\vec{x}_n)]^{-1} \cdot \vec{\nabla} f(\vec{x}_n) \quad .} \tag{7.15}$$

This is illustrated by the red arrows in fig. 7.4. This method can be very efficient with **quadratic convergence**, $|\vec{\delta}|_{n+1} \sim |\vec{\delta}|_n^2$, however in problems with large dimensions the calculation of the inverse Hessian can become very slow. In addition calculating the Hessian using finite differences often yields a matrix which is not strictly positive definite.

Example – Consider the 2D parabolic function

$$f(x,y) = (x - x_\star)^2 + ay^2$$

with minimum at $\vec{x_\star} = (x_\star, 0)$. The gradient and Hessian are given by

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 2(x - x_\star) \\ 2ay \end{bmatrix} \quad , \qquad \mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2a \end{bmatrix} \quad .$$

It is easy to find $\mathbf{H}^{-1}$. Hence starting at $\vec{x} = \vec{x}_0 = (x_0, y_0)$ equation (7.13) yields

$$\vec{\delta} = -\begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2a} \end{bmatrix} \cdot \begin{bmatrix} 2(x_0 - x_\star) \\ 2ay_0 \end{bmatrix} = -\begin{bmatrix} (x_0 - x_\star) \\ y_0 \end{bmatrix} = \vec{x_\star} - \vec{x}_0 \ ,$$

so that $\vec{\delta}$ takes us directly to the minimum (i.e. $\vec{x}_0 + \vec{\delta} = \vec{x_\star}$) in this case of a parabolic function.

Aside – What we have done in obtaining equation (7.13) is completely analogous to what would be done for finding the extremum of a simple quadratic $f(x) = ax^2 + bx + c$ if we know its gradient and curvature at a point x_0. The turning point $f' = 2ax + b = 0$ occurs at $x = -b/2a \equiv x_\star$. Also $f'' = 2a$. The exact Taylor expansion is, $f(x_0 + h) = f(x_0) + hf'|_{x_0} + \frac{1}{2}h^2 f''|_{x_0} = (ax_0^2 + bx_0 + c) + h(2ax_0 + b) + \frac{1}{2}h^2(2a)$. The gradient of the Taylor expansion is

$$f'|_{x_0+h} = f'|_{x_0} + h\, f''|_{x_0} = (2x_0 + b) + h(2a) + h^2(0) \ .$$

From this we get the value of h to take us from x_0 to the turning point ($f'|_{x_0+h} = 0$);

$$h = -f'|_{x_0}/f''|_{x_0} = -x_0 - b/2a \ .$$

This can be re-expressed as $h = x_\star - x_0$, demonstrating that getting h from the gradient of the Taylor expansion does indeed work for a parabola. For the multi-dimensional case equation (7.13) corresponds to the expression for h above.

Quasi-Newton Method (non-examinable)

A disadvantage of Newton's method is that we need to calculate both first and second derivatives of the multi-dimensional function, and the inverse of the curvature matrix which takes $\mathcal{O}(N^3)$.

The Quasi-Newton method gets around this by approximating the inverse Hessian using the local gradient. The basic iteration step is

$$\boxed{\vec{x}_{n+1} = \vec{x}_n - \alpha\, \mathbf{G}_n \cdot \vec{\nabla} f(\vec{x}_n) \ ,} \tag{7.16}$$

where $\mathbf{G}_n$ is an approximation of $\mathbf{H}^{-1}(\vec{x}_n)$ and $\alpha \ll 1$.

For the first iteration $\mathbf{G}_0$ is set to the identity matrix $\mathbf{I}$, which makes this iteration the same as a gradient search. To update $\mathbf{G}_n \to \mathbf{G}_{n+1}$ we use the updates in $\vec{x}$ and $\vec{\nabla} f$;

$$\vec{\delta}_n = \vec{x}_{n+1} - \vec{x}_n \qquad , \qquad \vec{\gamma}_n = \vec{\nabla} f(\vec{x}_{n+1}) - \vec{\nabla} f(\vec{x}_n) \ .$$

Then comparing the above expression for the gradient update $\vec{\gamma}_n$ with the gradient of the Taylor expansion of $f(\vec{x})$, to linear order,

$$\vec{\nabla} f(\vec{x}_{n+1}) \approx \vec{\nabla} f(\vec{x}_n) + \vec{\nabla} \left(\vec{\nabla} f(\vec{x}_n) \cdot \vec{\delta}_n \right)$$

we see that to linear order

$$\mathbf{H}_n^{-1} \cdot \vec{\gamma}_n = \vec{\delta}_n \ .$$

(This is equivalent to equation (7.13) when $\nabla f(\vec{x}_0 + \vec{\delta}) \neq 0$.) The trick is then to update $\mathbf{G}$ to satisfy

$$\mathbf{G}_n \cdot \vec{\gamma}_n = \vec{\delta}_n \, . \tag{7.17}$$

so that $\mathbf{G}_n$ mimics the inverse Hessian. A number of methods exist for this update, each with different efficiency and convergence characteristics. One of the most common is the DFP (Davidon–Fletcher–Power) algorithm where the update is

$$\mathbf{G}_{n+1} = \mathbf{G}_n + \frac{(\vec{\delta}_n \otimes \vec{\delta}_n)}{\vec{\gamma}_n \cdot \vec{\delta}_n} - \frac{\mathbf{G}_n \cdot (\vec{\delta}_n \otimes \vec{\delta}_n) \cdot \mathbf{G}_n}{\vec{\gamma}_n \cdot \mathbf{G}_n \cdot \vec{\gamma}_n} \, , \tag{7.18}$$

where $(\vec{u} \otimes \vec{v})_{ij} \equiv u_i v_j$ is the outer product of $\vec{u}$ and $\vec{v}$.

The advantage of this scheme is that it only involves (at worst) matrix multiplications, i.e., $\mathcal{O}(N^k)$ operations at each iteration and the resulting (approximated) Hessian is positive definite by construction. For full matrices (such as $\mathbf{G}_n$ and $(\vec{\delta}_n \otimes \vec{\delta}_n)$ here), naively one would think that the index k is 3. However it can be shown that matrix multiplication can be done with $k < 3$, in particular the Coppersmith & Winograd (1990) method scales with $k = 2.376$ and the commonly used `BLAS` routine scales with $k = 2.807$. Although these may seem close to $k = 3$ they make a big difference in computation time!

Global minimum search

These methods do not allow for the fact that the function may have multiple minima (local minima) in the interval we are interested in. There is no fool-proof method to find the global minimum (lowest minimum) of a function and we often need to restart algorithms from different starting points to check we have not found a local minimum.

7.3. Using minimisation to solve matrix equations

The value of $\vec{x}$ that minimises the **quadratic form**

$$f(\vec{x}) = \frac{1}{2}\vec{x}^T \cdot \mathbf{A} \cdot \vec{x} - \vec{b}^T \cdot \vec{x} \ , \tag{7.19}$$

happens to be the solution to

$$\mathbf{A} \cdot \vec{x} = \vec{b}$$

for a **symmetric, positive-definite** $N \times N$ matrix $\mathbf{A}$. This is because the gradient of this quadratic form is

$$\vec{\nabla} f(\vec{x}) = \mathbf{A} \cdot \vec{x} - \vec{b} \ . \tag{7.20}$$

At the minimum $\vec{x} = \vec{x}_\star$,

$$\vec{\nabla} f(\vec{x_\star}) = 0 \qquad \text{hence} \qquad \mathbf{A} \cdot \vec{x}_\star = \vec{b} \ .$$

(Note the similarity of this quadratic form with equation (7.14) seen earlier.) So to solve the matrix equation, one of the multi-dimensional iterative methods seen in section 7.2 can be used with the quadratic form above.

The best iterative methods, such as "conjugate gradient" and "bi-conjugate gradient" will (in the absence of round-off error) get the exact solution in $N_{\rm iter} = N$ iterations or less. This is better than Jacobi where convergence is as slow as $N_{iter} \sim \mathcal{O}(N^2)$ for some problems, and better than G-S & SOR. Convergence can be further accelerated by using a technique called ***pre-conditioning***. This effectively pushes the contours of $f(\vec{x})$ towards being spherical (in N dimensions), rather than very elongated ellipsoids (i.e. imagine fig. 7.2, but more elongated), so that a descent along the local gradient gets close to the global minimum.

To show that $\vec{\nabla} f = \mathbf{A} \cdot \vec{x} - \vec{b}$, consider the k-th component of the gradient of (7.19);

$$\begin{aligned}
(\vec{\nabla} f)_k &= \frac{\partial}{\partial x_k}\left[\frac{1}{2}\sum_i x_i \left(\sum_j A_{ij} x_j\right) - \sum_i b_i x_i\right] \\
&= \frac{1}{2}\sum_i \left[\delta_{ik}\left(\sum_j A_{ij} x_j\right) + x_i \left(\sum_j A_{ij}\delta_{jk}\right)\right] - b_k \\
&= \frac{1}{2}\left[\sum_j A_{kj} x_j + \sum_i x_i A_{ik}\right] - b_k \\
&= \frac{1}{2}\left[\mathbf{A} \cdot \vec{x}\right]_k + \frac{1}{2}\left[\vec{x}^T \cdot \mathbf{A}\right]_k - b_k \\
&= \frac{1}{2}\left[\mathbf{A} \cdot \vec{x}\right]_k + \frac{1}{2}\left[\mathbf{A}^T \cdot \vec{x}\right]_k - b_k \\
&= \left[\mathbf{A} \cdot \vec{x}\right]_k - b_k
\end{aligned}$$

where the last line makes use of the fact that $\mathbf{A}$ is symmetric. We have also used $\partial x_i / \partial x_k = \delta_{ik}$ where δ_{ik} is the Kronecker delta function.