Chapter 3

Matrix Algebra

Outline of Section

- Linear Algebraic Equations
- Direct Methods: Gaussian and Gauss-Jordan elimination
- Direct Methods: LU Decomposition
- Iterative Methods
- Eigensystems

3.1 Introduction

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

$$\begin{aligned} a_{11}\,x_1 + a_{12}\,x_2 + a_{13}\,x_3 + ... + a_{1N}\,x_N &= b_1, \\ a_{21}\,x_1 + a_{22}\,x_2 + a_{23}\,x_3 + ... + a_{2N}\,x_N &= b_2\,, \\ ... &= ..., \\ a_{M1}\,x_1 + a_{M2}\,x_2 + a_{M3}\,x_3 + ... + a_{MN}\,x_N &= b_M, \end{aligned} \tag{3.1}$$

which can be written as the matrix operation

$$\mathbf{A} \cdot \vec{x} = \vec{b}, \tag{3.2}$$

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. Any coefficient a_{ij} in the above simultaneous equations (which can be zero) 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 (3.2) arises frequently in many numerical methods. In section 9.5 we will see them in implicit finite difference methods for solving sets of coupled linear ODEs. In section 12 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 11) 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. We will encounter one such case in the stability analysis of finite difference methods for coupled ODEs in section 8.6.

3.2 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 fewer equations than unknowns (i.e., $M < N$) 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), which is 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.

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{B}, \tag{3.3}$$

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

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}, \tag{3.4}$$

so setting $\mathbf{B}$ to the identity matrix so the matrix equations become $\mathbf{A} \cdot \mathbf{X} = \mathbf{I}$ 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 $\det \mathbf{A} = 0$.

3.3 Gaussian and Gauss-Jordan elimination

Direct methods such as **Gaussian elimination** and **Gauss-Jordan elimination** involve manipulating the matrix to eliminate elements so that the system can be easily solved. These are basically what you would do if you needed to solve the matrix equation using pen and paper. The manipulation involves swapping rows and adding multiples of different rows to each other (or subtracting them from each other). The 'row-swapping' operation is known as 'pivoting', and is in general the only part of direct methods that can be tricky to code up.

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, the entries in $\mathbf{A}$ inform your / your code's choices about what manipulations to perform, and $\vec{b}$ (in Gaussian elimination) or $\mathbf{I}$ (in Gauss-Jordan elimination) 'shadow' those moves, and thereby become transformed to the answer.

3.4 LU Decomposition

This is another 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

$$\mathbf{A} \equiv \mathbf{L} \cdot \mathbf{U} \tag{3.5}$$

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

$$\begin{pmatrix} \alpha_{11} & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} \\ 0 & \beta_{22} & \beta_{23} \\ 0 & 0 & \beta_{33} \end{pmatrix}, \tag{3.6}$$

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

If the above factorisation holds, we can write the system

$$\boxed{\mathbf{A}\cdot\vec{x}=\mathbf{L}\cdot\mathbf{U}\cdot\vec{x}=\mathbf{L}\cdot\vec{y}=\vec{b},} \tag{3.7}$$

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 $\vec{y}$.

$$y_1 = \frac{b_1}{\alpha_{11}} \tag{3.8}$$

$$y_i = \frac{1}{\alpha_{ii}}\left(b_i-\sum_{j=1}^{i-1}\alpha_{ij}y_j\right) \quad i=2,3,...,N(\text{in this order}). \tag{3.9}$$

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

$$x_N = \frac{y_N}{\beta_{NN}} \tag{3.10}$$

$$x_i = \frac{1}{\beta_{ii}}\left(y_i-\sum_{j=i+1}^{N}\beta_{ij}x_j\right) \quad i=N-1,N-2,...,1, \tag{3.11}$$

where the calculation in (3.11) must be carried out in the order shown.

The following representation of the coupled equations may help in interpreting the forward and backward substitution expressions above:

$$\begin{aligned}
\alpha_{11}\,y_1 &= b_1,\\
\alpha_{21}\,y_1+\alpha_{22}\,y_2 &= b_2,\\
\alpha_{31}\,y_1+\alpha_{32}\,y_2+\alpha_{33}\,y_3 &= b_3,\\
\alpha_{41}\,y_1+\alpha_{42}\,y_2+\alpha_{43}\,y_3+... &= b_4,\\
\alpha_{51}\,y_1+\alpha_{52}\,y_2+\alpha_{53}\,y_3+... &= b_5,\\
... &= ...,\\
\alpha_{N1}\,y_1+\alpha_{N2}\,y_2+\alpha_{N3}\,y_3+...+\alpha_{NN}\,y_N &= b_N
\end{aligned}$$

$$\begin{aligned}
\beta_{11}\,x_1+\beta_{12}\,x_2+\beta_{13}\,x_3+...+\beta_{1N}\,x_N &= y_1,\\
\beta_{22}\,x_2+\beta_{23}\,x_3+...+\beta_{2N}\,x_N &= y_2,\\
\beta_{33}\,x_3+...+\beta_{3N}\,x_N &= y_3,\\
...+\beta_{4N}\,x_N &= y_4,\\
...+\beta_{5N}\,x_N &= y_5,\\
... &= ...,\\
\beta_{NN}\,x_N &= y_N
\end{aligned}$$

Here we briefly outline how such a decomposition can be performed. Details can be found in the recommended reading material. We follow the treatment of *Numerical Recipes*, but some may prefer the highly pedagogical description in Gerald and Wheatley.

The matrix equation 3.5 represesents $N\times N$ individual equations, with N^2+N unknown variables (the α_{ij} and β_{ij} terms, where the diagonals are counted twice). This means that N of these variables can be specified arbitrarily—and Crout's algorithm for decomposition chooses to

fix the diagonal α_{ii} values to 1. Following this, for each $j = 1, 2, 3, \ldots, N$, it is possible to solve for β_{ij}, $(i = 1, 2, \ldots, j)$:

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj} \tag{3.12}$$

Then, one can solve for α_{ij} for $(i = j + 1, j + 2, \ldots, N)$:

$$\alpha ij = \frac{1}{\beta_{jj}} \left(\alpha_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right). \tag{3.13}$$

The above procedure determines all the α and β values by the time that they are needed in the algorithm. From a computational point of view, the fact that each element of a is only used once lends itself to the storage of the newly-found α and β values in-situ in the same matrix that A was held in.

As with many of these algorithms, stability is a critical requirement, and the division by β_{jj} is an operation that can introduce instability if the value of βjj is small. Crout's altgorithm *with partial pivoting* is an enhancement of this above procedure that helps make use of the freedom to choose which element becomes the diagonal element βjj to improve the algorithm's stability.

In `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.

A significant benefit of the LU method is that it only depends on $\mathbf{A}$; if $\mathbf{A} \cdot \vec{x} = \vec{b}$ is to be solved many times for different choices of $\vec{b}$, it is advantageous to spend the time decomposing $\mathbf{A}$ once first.

If we want to find the inverse of $\mathbf{A}$ then we can decompose the matrix ($\mathcal{O}(N^3)$) once and solve equation (3.4) 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

$$\det \mathbf{A} = \prod_{j=1}^{N} \beta_{jj} \tag{3.14}$$

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

$$\ln(\det \mathbf{A}) = \sum_{j=1}^{N} \ln \beta_{jj} \equiv tr(\mathbf{A}). \tag{3.15}$$

The rest of this chapter concentrates mainly on **iterative methods** for solving $\mathbf{A} \cdot \vec{x} = \vec{b}$ and finding eigenvalues. For large matrices, these turn out to be more suitable (= faster or with smaller memory footprints) than direct methods.

3.5 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.,

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, \tag{3.16}$$

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. [a]

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

$$\begin{pmatrix} \cdot & \cdot & & & & & \\ \cdot & \cdot & \cdot & & & & \\ & \cdot & \cdot & \cdot & & & 0 \\ & & \cdot & \cdot & \cdot & & \\ & 0 & & \cdot & \cdot & \cdot & \\ & & & & \cdot & \cdot & \cdot \\ & & & & & \cdot & \cdot \end{pmatrix} . \tag{3.17}$$

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

$$\boxed{\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}.} \tag{3.18}$$

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

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

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

$$\boxed{\vec{x}_{n+1} = -\mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U}) \cdot \vec{x}_n + \mathbf{D}^{-1} \cdot \vec{b}} \tag{3.19}$$

which we can use to find $\vec{x}$ starting from a guess $\vec{x}_0$. Equation (3.19) 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 $|\lambda^i| < 1$. [b]

In the above, the lower and upper triangular matrices only appear together as a sum. We could have used a new matrix to represent the sum and only used that, but we shall in later in the chapter that keeping them separate can be useful.

We now need to introduce some new terminology: the **spectral radius** of $\mathbf{T}$. This 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. [c] 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

$$\mathbf{D}^{-1} = \mathrm{diag}\left(\frac{1}{d_1}, \frac{1}{d_2}, \ldots, \frac{1}{d_N}\right). \tag{3.20}$$

Proof of Jacobi convergence condition – The condition $|\lambda^i| < 1$ is simple to prove, using similar principles to those used previously in section 8.6 for the stability analysis of finite difference solution of a set of coupled linear ODEs. As before, we consider the errors at each iteration

$$\vec{\epsilon}_n = \vec{x}_n - \vec{x} \qquad \text{and} \qquad \vec{\epsilon}_{n+1} = \vec{x}_{n+1} - \vec{x}, \tag{3.21}$$

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

$$\begin{aligned} \vec{\epsilon}_{n+1} = \vec{x}_{n+1} - \vec{x} &= -\mathbf{T} \cdot \vec{x}_n + \mathbf{D}^{-1} \cdot \vec{b} - \left(-\mathbf{T} \cdot \vec{x} + \mathbf{D}^{-1} \cdot \vec{b} \right) \\ &= -\mathbf{T} \cdot \vec{x}_n + \mathbf{T} \cdot \vec{x} \\ &= -\mathbf{T} \cdot (\vec{x}_n - \vec{x}), \\ \therefore \qquad \vec{\epsilon}_{n+1} &= -\mathbf{T} \cdot \vec{\epsilon}_n. \end{aligned} \tag{3.22}$$

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

$$\mathbf{T} = \mathbf{R} \cdot \mathbf{\Lambda} \cdot \mathbf{R}^{-1} \quad \text{with} \quad \mathbf{\Lambda} \equiv \text{diag}(\lambda^1, \lambda^2, ..., \lambda^N), \tag{3.23}$$

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

$$\vec{\zeta}_{n+1} = \mathbf{\Lambda} \cdot \vec{\zeta}_n, \tag{3.24}$$

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

$$\left| \frac{\zeta^i_{n+1}}{\zeta^i_n} \right| \equiv \left| \lambda^i \right| < 1. \tag{3.25}$$

(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, iterative methods are usually 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 a full matrix. (It should be noted that 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

$$\varepsilon = \left| \frac{\|\vec{x}_{n+1}\| - \|\vec{x}_n\|}{\|\vec{x}_n\|} \right| \tag{3.26}$$

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

$$\|\vec{x}\|_\ell \equiv \left(\sum_{i=1}^{n} |x_i|^\ell \right)^{\frac{1}{\ell}}. \tag{3.27}$$

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.

3.6 Iterative Gauss-Seidel Method

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

$$\boxed{\vec{x}_{n+1} = -(\mathbf{L} + \mathbf{D})^{-1} \cdot \mathbf{U} \cdot \vec{x}_n + (\mathbf{L} + \mathbf{D})^{-1} \cdot \vec{b},} \tag{3.28}$$

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 (3.28) 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 (3.9).) In this form G-S looks like

$$x_k^{(n+1)} = \frac{1}{D_{kk}} \left(-\sum_{j=1}^{k-1} L_{kj} x_j^{(n+1)} - \sum_{j=k+1}^{N} U_{kj} x_j^{(n)} + b_k \right), \quad k = 1, 2, \ldots, N. \tag{3.29}$$

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

Note that the result for $x_{k-1}^{(n+1)}$ is used when calculating $x_k^{(n+1)}$; allowing for more up-to-date information to be used for each calculation, rather than only using the nth-step result for step $n + 1$.

3.7 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. The Gauss-Seidel relationship (equation 3.29) can be rewritten as follows:

$$x_k^{(n+1)} = x_k^n + \frac{\omega}{D_{kk}} \left(-\sum_{j=1}^{k-1} L_{kj} x_j^{(n+1)} - \sum_{j=k}^{N} U_{kj} x_j^{(n)} + b_k \right), \quad k = 1, 2, \ldots, N, \tag{3.30}$$

if the newly-introduced parameter ω is set to 1.

The above expression is equivalent to

$$\boxed{\vec{x}_{n+1} = (\omega \mathbf{L} + \mathbf{D})^{-1} \cdot \left(-[\omega \mathbf{U} + (\omega - 1)\mathbf{D}] \cdot \vec{x}_n + \omega \vec{b} \right).} \tag{3.31}$$

This new parameter ω is called the **relaxation parameter**, and a high speed of convergence can be achieved by tuning it, which affects the spectral radius of SOR's update matrix. 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—which we describe in the next section.

3.8 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

$$\mathbf{A} \cdot \vec{x} = \lambda \vec{x}, \tag{3.32}$$

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 [d] with associated linearly independent eigenvectors $(\vec{e}_i)$

$$\mathbf{A} \cdot \vec{e}_i = \lambda_i \vec{e}_i \,. \tag{3.33}$$

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

$$\vec{v} = \sum_{i=1}^{N} c_i \vec{e}_i \,. \tag{3.34}$$

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

$$\begin{aligned} \mathbf{A}\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\mathbf{A}\cdot\vec{e}_i = \sum_{i=1}^{N} c_i\,\lambda_i\,\vec{e}_i, \\ \mathbf{A}\cdot\mathbf{A}\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\lambda_i^2\,\vec{e}_i, \\ \mathbf{A}^n\cdot\vec{v} &= \sum_{i=1}^{N} c_i\,\lambda_i^n\,\vec{e}_i, \\ \therefore \quad \lim_{n\to\infty} \mathbf{A}^n\cdot\vec{v} &\to c_j\,\lambda_j^n\,\vec{e}_j, \end{aligned} \tag{3.35}$$

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

$$\boxed{\vec{\omega}_j = \frac{\mathbf{A}^n\cdot\vec{v}}{|\mathbf{A}^n\cdot\vec{v}|} \equiv \frac{\vec{e}_j}{|\vec{e}_j|},} \tag{3.36}$$

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

Then it is simple to calculate the eigenvalue itself since

$$\boxed{{\vec{\omega}_j}^T\cdot\mathbf{A}\cdot\vec{\omega}_j = {\vec{\omega}_j}^T\cdot(\lambda_j\vec{\omega}_j) = \lambda_j.} \tag{3.37}$$

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 (3.36) and (3.37). This works because, the inverse of a matrix has the same eigenvectors but with the inverse eigenvalues. This can be shown since

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

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 $\mathbf{A}$.

3.9 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

$$\mathbf{A}' \equiv \mathbf{A} - \alpha\,\mathbf{I}, \tag{3.38}$$

such that

$$\mathbf{A}'\cdot\vec{e}_i = \mathbf{A}\cdot\vec{e}_i - \alpha\,\mathbf{I}\cdot\vec{e}_i = \lambda_i\,\vec{e}_i - \alpha\,\vec{e}_i = (\lambda_i-\alpha)\,\vec{e}_i\,. \tag{3.39}$$

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 can we obtain $(\mathbf{A}')^{-1}$? 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

$$|\lambda_i - a_{ii}| \leq \sum_{j \neq i} |a_{ij}| \,, \tag{3.40}$$

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}. [e] 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 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[f]

$$\mathbf{A} \equiv \begin{pmatrix} 1 & 0.1 & 0 \\ 0.1 & 5 & 0.2 \\ 0.1 & 0.3 & 10 \end{pmatrix} . \tag{3.41}$$

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 3.5, 3.6, 3.7), and thus determining the suitability of a matrix $\mathbf{A}$ for solution by, e.g., Jacobi iteration.

[a] 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.

[b] Note that a real-valued matrix can have complex eigenvalues.

[c] 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 $\mathbf{A}$.

[d] 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.

[e] The role of 'i' in equation (3.40) 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).

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