# Matrices

Basic Linear Algebra

# Overview

- Lecture will cover
  - why matrices and linear algebra are so important
  - basic terminology
  - Gauss-Jordan elimination
  - LU factorisation
  - error estimation
  - libraries

# Linear algebra

- In mathematics linear algebra is the study of linear transformations and vector spaces…
- …in practice linear algebra is the study of matrices and vectors
- Many physical problems can be formulated in terms of matrices and vectors

# Health Warning

- Don’t let the terminology scare you
  - concepts quite straightforward, algorithms easily understandable
  - implementing the methods is often surprisingly easy
  - but numerous variations (often for special cases or improved numerical stability) lead to an explosion in terminology

Cholesky Decomposition

LU factorisation

Condition number

Back-substitution

Symmetric Positive Definite

Gauss-Jordan

Forward-substitution

Singular value decomposition

Crout’s algorithm

Partial pivoting

# Basic matrices and vectors

- Matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

- Vector

$$v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

- A matrix multiplied by a vector gives another vector

$$Av = w = \begin{pmatrix} a_{11}v_1 + a_{12}v_2 + a_{13}v_3 \\ a_{21}v_1 + a_{22}v_2 + a_{23}v_3 \\ a_{31}v_1 + a_{32}v_2 + a_{33}v_3 \end{pmatrix}$$

# Linear Systems as Matrix Equations

- Many problems expressible as linear equations
  - two apples and three pears cost 40 pence
  - three apples and five pears cost 65 pence
  - how much does one apple or one pear cost?
- Express this as

$$2a + 3p = 40$$

$$3a + 5p = 65$$

- Or in matrix form

$$\begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix} \begin{bmatrix} a \\ p \end{bmatrix} = \begin{bmatrix} 40 \\ 65 \end{bmatrix}$$

  - *matrix* x *vector* = *vector*

# Standard Notation

- For a system of *N* equations in *N* unknowns

$$
\begin{gathered}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1N}x_N = b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2N}x_N = b_2 \\
\vdots \\
a_{N1}x_1 + a_{N2}x_2 + \ldots + a_{NN}x_N = b_N
\end{gathered}
$$

  - coefficients form a matrix *A* with elements $a_{ij}$
  - unknowns form a vector *x* with elements $x_i$
  - solution forms a vector *b* with elements $b_i$

- All linear equations have the form $A\,x = b$

# Matrix Inverse

- $A\,x = b$ implies $A^{-1}A\,x = x = A^{-1}\,b$
  - simple formulae exist for $N$=2

$$A^{-1} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^{-1} = \frac{1}{a_{11}a_{22} - a_{12}a_{21}} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}$$

$$\begin{bmatrix} 5 & -3 \\ -3 & 2 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix} \begin{bmatrix} a \\ p \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ p \end{bmatrix} = \begin{bmatrix} 5 & -3 \\ -3 & 2 \end{bmatrix} \begin{bmatrix} 40 \\ 65 \end{bmatrix} = \begin{bmatrix} 5 \\ 10 \end{bmatrix}$$

- Rarely need (or want) to store the explicit inverse
  - usually only require the solution to a particular set of equations
- Algebraic inversion impractical for large *N*
  - use numerical algorithms such as Gaussian Elimination

# Simultaneous Equations

- Equations are:

$$\begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix} \begin{bmatrix} a \\ p \end{bmatrix} = \begin{bmatrix} 40 \\ 65 \end{bmatrix}$$

$2a + 3p = 40$ (i)

$3a + 5p = 65$ (ii)

  - computing 2 x (ii) - 3 x (i) gives $p = 130 - 120 = 10$
  - substitute in (i) gives $a = 1/2 \times (40 - 3 \times 10) = 5$
- Imagine we actually had

$2.00000\,a + 3.00000\,p = 40.00000$ (i)

$4.00000\,a + 6.00001\,p = 80.00010$ (ii)

(ii) - 2 x (i) gives $(6.00001 - 6.00000)\,p = (80.00010 - 80.00000)$

  - cancellations on both sides may give inaccurate numerical results
  - value of $p$ comes from multiplying a huge number by a tiny one
- How can we tell this will happen in advance?

# Matrix Conditioning

- Characterise a matrix by its *condition number*
  - gives a measure of the range of the floating point numbers that will be required to solve the system of equations
- A *well-conditioned* matrix
  - has a small condition number
  - and is numerically easy to solve
- An *ill-conditioned* matrix
  - has a large condition number
  - and is numerically difficult to solve
- A *singular* matrix
  - has an infinite condition nymber
  - is impossible to solve numerically (or analytically)

# Calculating the Condition Number

- Easy to compute condition no. for small problems

  $2a + 3\,p = 40$
  $3a + 5\,p = 65$

  – has a condition number of 46 (ratio of largest/smallest eigenvalue)

  $2.00000\,a + 3.00000\,p = 40.00000$
  $4.00000\,a + 6.00001\,p = 80.00010$

  – has condition number of 8 million!

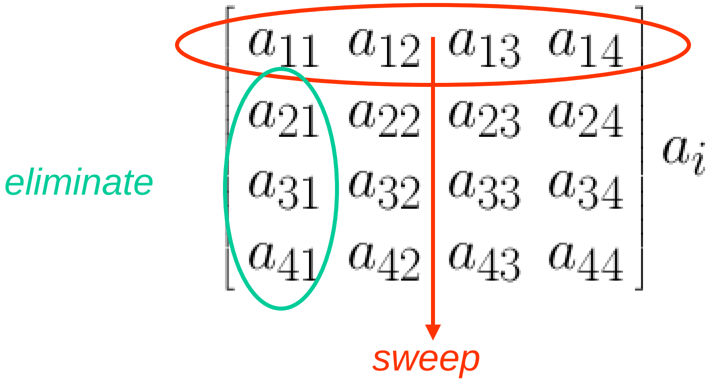
- Very hard to compute for real problems
  – methods exist for obtaining good estimates

# Relevance of Condition Number

- Gives a measure of the range of the scales of numbers in the problem
  - eg if condition number = 46, largest number required in calculation will be roughly 46 times larger than smallest
  - if condition number = $10^7$, this may be a problem for single precision where we can only resolve one part in $10^8$
- May require higher precision to solve ill-conditioned problems
  - in addition to a robust algorithm

# Gauss-Jordan Elimination

- The technique you may have learned at school
  - subtract rows of *A* from other rows to eliminate off-diagonals
  - must perform same operations to RHS (i.e. *b*)



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \; a_{ij} \rightarrow a_{ij} - \frac{a_{i1}}{a_{11}} a_{1j} \; \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & a'_{32} & a'_{33} & a'_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{bmatrix}$$

- Pivoting
  - using row *p* as the *pivot row* (*p*=1 above) implies division by $a_{pp}$
  - very important to do row exchange to maximise $a_{pp}$
  - this is *partial pivoting* (full pivoting includes column exchange)

# Observations

- Gauss-Jordan is a simple *direct* method
  - we know the operation count at the outset, complexity O($N^3$)
- Possible to reduce *A* to purely diagonal form
  - solving a diagonal system is trivial

$$\begin{bmatrix} a'_{11} & 0 & 0 & 0 \\ 0 & a'_{22} & 0 & 0 \\ 0 & 0 & a'_{33} & 0 \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix} \quad \rightarrow \quad \begin{aligned} a'_{11}x_1 &= b'_1 \\ a'_{22}x_2 &= b'_2 \\ a'_{33}x_3 &= b'_3 \\ a'_{44}x_4 &= b'_4 \end{aligned}$$

  - better to reduce to upper triangular - Gaussian Elimination

# Gaussian Elimination

- Operate on active sub-matrix of decreasing size

$$\begin{bmatrix} a'_{11} & \begin{matrix} a'_{12} & a'_{13} & a'_{14} \end{matrix} \\ \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} & \boxed{\begin{matrix} a'_{22} & a'_{23} & a'_{24} \\ a'_{32} & a'_{33} & a'_{34} \\ a'_{42} & a'_{43} & a'_{44} \end{matrix}} \end{bmatrix} \rightarrow \begin{bmatrix} \begin{matrix} a'_{11} & a'_{12} \\ 0 & a'_{22} \\ 0 & 0 \\ 0 & 0 \end{matrix} & \begin{matrix} a'_{13} & a'_{14} \\ a'_{23} & a'_{24} \\ \multicolumn{2}{c}{\boxed{\begin{matrix} a'_{33} & a'_{34} \\ a'_{43} & a'_{44} \end{matrix}}} \end{matrix} \end{bmatrix} \rightarrow \cdots$$

- Solve resulting system with *back-substitution*
  - can compute $x_4$ first, then $x_3$, then $x_2$, etc...

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix} \qquad \begin{aligned} a'_{11}x_1 + a'_{12}x_2 + a'_{13}x_3 + a'_{14}x_4 &= b'_1 \\ a'_{22}x_2 + a'_{23}x_3 + a'_{24}x_4 &= b'_2 \\ a'_{33}x_3 + a'_{34}x_4 &= b'_3 \\ a'_{44}x_4 &= b'_4 \end{aligned}$$

# LU Factorisation

- Gaussian Elimination is a practical method
  - must do partial pivoting and keep track of row permutations
  - restriction: must start a new computation for every different $b$
- Upper-triangular system $U x = b$ easy to solve
  - likewise for lower-triangular $L x = b$ using forward-substitution
- Imagine we could decompose $A = LU$
  - $A x = (LU) x = L (Ux) = b$
  - first solve $Ly = b$ then $Ux = y$
  - each triangular solve has complexity $O(N^2)$
- But how do we compute the $L$ and $U$ factors?

# Computing *L* and *U*

- Clearly only have $N^2$ unknowns
  - assume *L* is *unit* lower triangular and *U* is upper triangular

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ l_{21} & 1 & \cdot & \cdot \\ l_{31} & l_{32} & 1 & \cdot \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ \cdot & u_{22} & u_{23} & u_{24} \\ \cdot & \cdot & u_{33} & u_{34} \\ \cdot & \cdot & \cdot & u_{44} \end{bmatrix}$$

  - writing out in full

$$\left[\begin{array}{r|r|r|c} u_{11} & u_{12} & u_{13} & \cdot \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} & \cdot \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} & \cdot \\ l_{41}u_{11} & l_{41}u_{12} + l_{42}u_{22} & l_{41}u_{13} + l_{42}u_{23} + l_{43}u_{33} & \cdot \end{array}\right]$$

# Implementation

- Can pack LU factors into a single matrix

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \rightarrow \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21} & u_{22} & u_{23} & u_{24} \\ l_{31} & l_{32} & u_{33} & u_{34} \\ l_{41} & l_{42} & l_{43} & u_{44} \end{bmatrix}$$

- RHS computed in columns
  - once $l_{ij}$ or $u_{ij}$ is calculated, $a_{ij}$ is not needed any more
  - can therefore do LU decomposition *in*-place
  - elements of *A* over-written by *L* and *U*
  - complexity is O($N^3$)

# Crout’s Algorithm

- Replaces *A* by its *LU* decomposition
  - implements pivoting, ie decomposes row permutation of *A*
  - computation of $l_{ij}$ requires division by $u_{jj}$
  - can promote a sub-diagonal $l_{ij}$ as appropriate
  - essential for stability with large *N*

- Loop over columns *j*
  - compute $u_{ij}$ for $i = 1, 2 \ldots j$
  - compute $l_{ij}$ for $i = j+1, j+2 \,..\, N$
  - pivot as appropriate before proceeding to next column

- See, e.g., Numerical Recipes section 2.3

# Procedure

- To solve $Ax = b$
  - decompose $A$ into $L$ and $U$ factors via Crout’s algorithm
  - replaces $A$ in-place
  - set $x = b$
  - do in-place solution of $Lx = x$ (forward substitution)
  - do in-place solution of $Ux = x$ (backward substitution)
- Advantages
  - pivoting makes the procedure stable
  - only compute $LU$ factors once for any number of vectors $b$
  - subsequent solutions are $O(N^2)$ after initial $O(N^3)$ factorisation
  - to compute inverse, solve for a set of $N$ unit vectors $b$
  - determinant of $A$ can be computed from the product of $u_{ii}$

# Quantifying the Error

- We hope to have solved *Ax* = *b*
  - there will inevitably be errors due to limited precision
  - can quantify this by computing the residual vector *r* = *b* - *Ax*
  - typically quote the root-mean-square *residue*

$$residue = \frac{||r||_2}{||b||_2}, \qquad ||x||_2 = \sqrt{x^T x} = \sqrt{\sum_{i=1}^{N} x_i^2}$$

  - length defined by $L_2$ *norm* (“two-norm”) - other norms exist

^

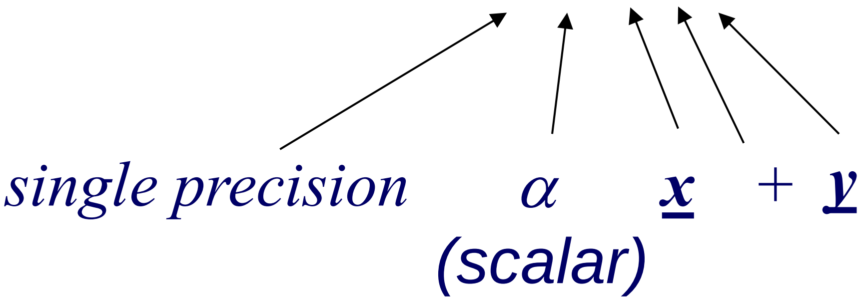
# Linear algebra libraries

- Linear Algebra is a well constrained problem
  - can define a small set of common operations
  - implement them robustly and efficiently in a library
  - mainly designed to be called from Fortran (see later ...)
- Often seen as the most important HPC library
  - eg LINPACK benchmark is standard HPC performance metric
  - solve a linear system with LU factorisation
  - possible to achieve performance close to theoretical peak
- Linear algebra is unusually efficient
  - LU decomposition has O($N^3$) operations for O($N^2$) memory loads

# BLAS

- Basic Linear Algebra Subprograms
  - Level 1: vector-vector operations (e.g. $\underline{\mathbf{x}}\cdot\underline{\mathbf{y}}$)
  - Level 2: matrix-vector operations (e.g. $\mathbf{A}\underline{\mathbf{x}}$)
  - Level 3: matrix-matrix operations (e.g. $\mathbf{AB}$)

    ($\underline{\mathbf{x}}$, $\underline{\mathbf{y}}$ vectors, $\mathbf{A}$, $\mathbf{B}$ matrices)
- Example: SAXPY routine



*single precision* $\alpha$ $\underline{\boldsymbol{x}}$ $+$ $\underline{\boldsymbol{y}}$

(scalar)

$\underline{\boldsymbol{y}}$ *is replaced "in-place" with* $\alpha\,\underline{\boldsymbol{x}} + \underline{\boldsymbol{y}}$

# LAPACK

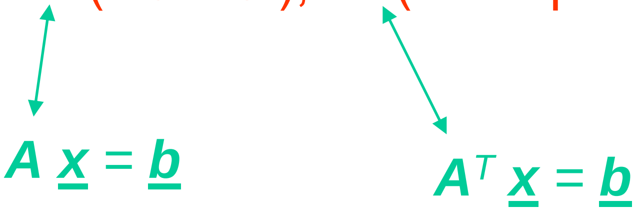
- LAPACK is built on top of BLAS libraries
  - Most of the computation is done with the BLAS libraries
- Original goal of LAPACK was to improve upon previous libraries to run more efficiently on shared memory and multi-layered systems
  - Spend less time moving data around!
- LAPACK uses BLAS 3 instead of BLAS 1
  - matrix-matrix operations more efficient than vector-vector
- **Always use libraries for Linear Algebra**

# LU factorisation

- LU factorisation
  - `call SGETRF(M, N, A, LDA, IPIV, INFO)`
  - does an in-place LU factorisation of *M* by *N* matrix *A*
    - we will always consider the case *M* = *N*
  - *A* can actually be declared as `REAL A(NMAX,MMAX)`
    - routine operates on M x N submatrix
    - must tell the library the Leading Dimension of A, ie set LDA=NMAX
  - `INTEGER IPIV(N)` returns row permutation due to pivoting
  - error information returned in the integer `INFO`

# Solving: Forward/backward substituion

- Forward / backward substitution
  - `call SGETRS(TRANS,N,NRHS,A,LDA,IPIV,B,LDB,INFO)`
  - expects a factored **`A`** and **`IPIV`** from previous call to **`SGETRF`**
  - solves for multiple right-hand-sides, ie **`B`** is **`N`** x **`NRHS`**
  - we will only consider **`NRHS=1`**, ie RHS is the usual vector $\underline{\mathbf{b}}$
  - solution *x* is returned in $\underline{\mathbf{b}}$ (ie original $\underline{\mathbf{b}}$ is destroyed)
- Options exist for precise form of equations
  - specified by character variable **`TRANS`**
  - ‘N’ (*N*ormal), ‘T’ (*T*ranspose)



$A\,\underline{\mathbf{x}} = \underline{\mathbf{b}}$

$A^T\,\underline{\mathbf{x}} = \underline{\mathbf{b}}$

# Summary

- Dense matrices arise from linear equations
  - standard notation is $Ax = b$
- Matrices characterised by their condition number
  - equations difficult to solve numerically have large condition number
    - an ill-conditioned matrix
  - may lead to large errors in our solution so **always quantify the error**
- Have covered direct solution methods for $Ax = b$
  - all are basically variants of Gaussian Elimination
  - rather than storing $A^{-1}$, compute the LU factors of $A$
  - can then solve further equations $Ax = c$, $Ax = d$, ... at little extra cost
  - the larger the condition number, the harder the problem
  - pivoting is essential in practice for numerical stability
- **Always use libraries for Linear Algebra**