Numerical Algorithms for HPC

Linear Algebra introduction

epcc

1

Overview

- This lecture will cover:
  - why matrices and linear algebra are so important
  - basic terminology
  - Gauss-Jordan elimination
  - LU factorisation
  - error estimation and iterative improvement

epcc

2

2

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

3

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
Partial pivoting
Crout's algorithm
Singular value decomposition

4

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

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

epcc

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

epcc

Standard Notation

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

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

- 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 inverting a matrix when *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:

  2a + 3p = 40 (i)

  3a + 5p = 65 (ii)

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

  - computing 2 x (ii) - 3 x (i) gives p = 130 - 120 = 10
  - substitute in (i) gives a = 1/2 x (40 - 3 x 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 Condition Number

- 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 number
  - 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 a condition number of 8 million!
- Very hard to compute for real problems
  - but methods exist for obtaining good estimates

Relevance of Condition Number

- Gives a measure of the range of the scales of numbers in the problem
  - e.g. 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

Simultaneous Equations revisited

- One way to solve our equations

  2a + 3p = 40 (i)

  3a + 5p = 65 (ii)

Note where the $1.5 = 3/2$ comes from

- would be to replace (ii) with (ii) – 1.5 x (i) giving

  2a + 3p = 40 (i)

  0a + 0.5p = 5 (ii updated)

- Not as nice this time due to non-integer values but can think of this as a new matrix and vector equivalent to initial problem

$$\begin{bmatrix}2 & 3\\ 3 & 5\end{bmatrix}\begin{bmatrix}a\\ p\end{bmatrix} = \begin{bmatrix}40\\ 65\end{bmatrix} \text{ becomes } \begin{bmatrix}2 & 3\\ 0 & 0.5\end{bmatrix}\begin{bmatrix}a\\ p\end{bmatrix} = \begin{bmatrix}40\\ 5\end{bmatrix}$$

Note the 0 here

Easy to calculate $p = \frac{5}{0.5} = 10$

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*)

eliminate

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

sweep

- 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}
    - Within a process rows won't usually move in memory – just note permutations
    - More complicated in parallel as rows or columns may be distributed!
  - 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)
  - *iterative* methods are optimised for a particular *b* - see later
- Possible to reduce *A* to purely diagonal form
  - solving a diagonal system is trivial

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

  - better to reduce to upper triangular - Gaussian Elimination

Gaussian Elimination

- Operate on active sub-matrix of decreasing size

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & a'_{32} & a'_{33} & a'_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{bmatrix} \rightarrow \begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & a'_{43} & a'_{44} \end{bmatrix} \rightarrow \dots$$

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

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

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 $Ux = b$ easy to solve
  - likewise for lower-triangular $Lx = b$ using forward-substitution
- Imagine we could decompose $A = LU$
  - $Ax = (LU)x = L(Ux) = b$
  - 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

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdot \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} & \cdot \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} & \cdot \\ l_{41}u_{11} & l_{41}u_{12} + l_{42}u_{22} & l_{41}u_{13} + l_{42}u_{23} + l_{43}u_{33} & \cdot \end{bmatrix}$$

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, i.e. decomposes row permutation of A
  - computation of l_{ij} requires division by u_{ij}
  - can promote a sub-diagonal element to a diagonal element 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 $i = j+1, j+2, \ldots, N$
  - pivot as appropriate before proceeding to next column
- For details see book: Numerical Recipes section 2.3
- Can parallelise various versions of LU algorithms but will look at more general parallel matrix libraries shortly

Procedure

- To solve $A\,x = b$
  - decompose A into L and U factors, e.g. via Crout's algorithm
  - replaces A in-place, so now $LUx = b$.
  - 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 $A\,x = b$ there will inevitably be errors due to limited precision
  - can quantify this by computing the residual vector $r = b - A\,x$
  - 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
- Residue may not be the best measure
  - really want difference δx from the perfect solution x
  - residual gives the error in the RHS b
  - but of course we don't know the perfect solution!

Error in the Solution

- Let $\hat{x}$ be perfect solution (which we don't know!)
  - i.e. $A\hat{x} = b$, so the residual $\hat{r} = b - A\hat{x} = 0$
- Our numerical solution is x
  - we have a finite residual $r = b - Ax$
  - i.e. $Ax = b - r$
- If we write $x = \hat{x} + \delta x$

  $Ax = A(\hat{x} + \delta x) = A\hat{x} + A\delta x = b + A\delta x$

  but: $Ax = b - r$ *i.e.* $b - r = b + A\delta x$

  so: $A\,\delta x = -r$
- Can solve for error δx in numerical solution x
  - without knowing the perfect solution!

Iterative Improvement

- Exact solution $\hat{x} = x - \delta x$
  - but we know that $A\,\delta x = -r$
  - can easily solve for δx as we already have the LU factors of A
- Procedure
  - compute LU factors of A and solve $Ax = b$
  - calculate residual vector $r = b - Ax$ and compute residue
  - solve $A\delta x = -r$ and improve the solution $x \rightarrow x - \delta x$
  - re-compute the residue and see how much it has reduced
  - repeat as appropriate
- Must compute elements r_i in double precision
  - but only need to store using single precision
  - Result can often then be found to full single precision

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 compute the residue!**
- 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 $Ax = c, Ax = d, \ldots$ at little extra cost
  - the larger the condition number, the harder the problem
  - pivoting is essential in practice for numerical stability

LU Factorisation Exercise

- Solve a dense system of equations
  - you are given a template code in C, Fortran or Java
- The codes should compile and run as they stand

```
$ gfortran -o lufact lufact.f90
$ ./lufact

$ gcc -o lufact lufact.c
$ ./lufact

$ javac lufact.java
$ java lufact
```

- Supplied code doesn't do anything useful
  - generates a random matrix *A* and a right-hand-side b
  - but **you** must supply LU decomposition and forward/backward substitution code to solve for x
  - as supplied, codes simply set L and U to zero and $x = b$

Exercise notes

- For simplicity
  - do not implement partial pivoting
  - the system is constructed to have a solution $x_i = 1$
    - unlike real problems, easy to check if you have the right answer!
- Debugging small problems (*N* <= 6)
  - code writes out L, U, L multiplied by U, x and Ax
  - if your LU decomposition is correct
    - L will be unit lower triangular, U will be upper triangular
    - L times U will be equal to the original matrix A
  - if your forward/backward substitution is correct
    - x will contain numbers all very close to 1.0 (the solution)
    - Ax will be close to b (since we are trying to solve $Ax = b$)
    - you **MUST** quantify this by computing the residual

Aims

- You should try to
  - get the right answer!
  - quantify the error by computing the residual
    - for this test case we can also compare to the error in the solution
  - see how the residual (a measure of the error) varies with N
  - investigate iterative improvement
  - look at the effect of using double precision

Asides (not examinable!)

- Uniqueness and existence of LU decomposition
  - LU decomp always exists if we allow pivoting
  - To see if LU decomp exists need to look at determinants of the submatrices resulting when we remove everything to the left and above each diagonal
  - If LU decomp exists and we force all diagonal entries of L (or U) to be all 1 then LU decomp is unique
- If *A* is Symmetric Positive Definite
  - can decompose $A = U^T U$
  - Crout's algorithm is simpler as $l_{ij} = u_{ji}$
  - called a *Cholesky* decomposition
    - ability to do Cholesky actually proves that *A* is SPD

29

Aside: LU factorisation is Gaussian Elimination!

- Gaussian elimination transforms a square matrix into an upper triangular matrix

$$A \rightarrow U$$

- We can think of the Gaussian Elimination process as an operation being performed on the matrix A
- In turn can construct a matrix which performs this operation: $\tilde{L}A = U$
- This matrix, $\tilde{L}$, happens to be lower triangular, rearranging gives

$$\tilde{L}\,A = U$$
$$A = \tilde{L}^{-1}U$$

- The inverse of a lower triangular matrix is also lower triangular: $L = \tilde{L}^{-1}$

$$\boxed{A = LU}$$

30

Eigenvalues

- The following slides are all for information only
  - you are not expected to understand the maths and none of this is examinable

Eigenvalues

- Can compute the *eigenvalues* of *A*
  - a small eigenvalue indicates a numerical problem
  - an *N* x *N* matrix has, in general, *N* eigenvalues λ_1, λ_2, ..., λ_N.
  - conventionally order them so that $|\lambda_1| < |\lambda_2| < ... < |\lambda_{N-1}| < |\lambda_N|$
  - eigenvalues for first problem on slide 9 are 0.15 and 6.85
  - eigenvalues for second problem are 0.000001 and 8.000009
- A matrix is, for most purposes, "equivalent" to the diagonal matrix formed by the eigenvalues

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \approx \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 \\ 0 & 0 & 0 & \lambda_4 \end{bmatrix}$$

Characterising Matrices

- Individual entries in matrix have little meaning
  - values can be changed by re-scaling equations
  - structure can be changed by re-labelling variables
- Important global characteristics
  - determinant
  - condition number
- Can both be expressed in terms of eigenvalues
  - unfortunately, computing the eigenvalues is much harder than solving linear equations!
  - we will see that we can compute the determinant in other ways

Matrix Properties

- Determinant
  - the product of the eigenvalues λ_i
  - a zero eigenvalue implies a zero determinant
  - comes from a singular matrix, ie a set of equations that are *impossible* to solve (or no unique solution)
- Condition number
  - given by ratio of maximum to minimum eigenvalue: λ_N / λ_1
  - poorly conditioned matrices have a large condition number
    - and a small determinant
  - correspond to equations that are *numerically difficult* to solve

Characterising a Matrix

- This is an *Applied* course so
  - for most practical purposes a matrix can be thought of as being a diagonal matrix made up of the eigenvalues
  - a theorem true for diagonal matrices is probably true in general!
- When does $A^M = A \times A \times A \times \cdots \times A$ converge to zero?
  - when all the eigenvalues have absolute value < 1

$$\begin{bmatrix} \lambda_1 & \cdot & \cdot & \cdot \\ \cdot & \lambda_2 & \cdot & \cdot \\ \cdot & \cdot & \lambda_3 & \cdot \\ \cdot & \cdot & \cdot & \lambda_4 \end{bmatrix}^M = \begin{bmatrix} \lambda_1^M & \cdot & \cdot & \cdot \\ \cdot & \lambda_2^M & \cdot & \cdot \\ \cdot & \cdot & \lambda_3^M & \cdot \\ \cdot & \cdot & \cdot & \lambda_4^M \end{bmatrix}$$

epcc

35

35

Matrices in Computational Science

- Often have very nice properties, eg
  - Symmetric: $a_{ij} = a_{ji}$ which guarantees real eigenvalues
  - Positive Definite: all eigenvalues are greater than zero
  - for an SPD matrix $x^T Ax$ is positive for all vectors x

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix} \begin{bmatrix} \lambda_1 & \cdot & \cdot & \cdot \\ \cdot & \lambda_2 & \cdot & \cdot \\ \cdot & \cdot & \lambda_3 & \cdot \\ \cdot & \cdot & \cdot & \lambda_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \lambda_1 x_1^2 + \lambda_2 x_2^2 + \ldots$$

- Physical significance of eigenvalues / vectors
  - engineering: frequency of vibration / direction of movement
  - quantum mechanics: energy of electron / distribution in space

epcc

36

36

Special Matrices

- If *A* is singular or very ill-conditioned
  - can do a Singular Value Decomposition
  - SVD is, in some sense, the “best” of all possible inverses
  - beyond the scope of this course