Overview of slides 06

1. ODE Boundary value problems
2. Linear algebra
   - Direct elimination (Gaussian elimination)
   - Krylov subspace methods
   - Sparse matrices
3. Summary

Boundary value ODEs

Learned in Comp. Phys. 1 — solving boundary value problems and eigenvalue problems in ODEs. Methods:

Shooting method

1. Guess unknown initial values v_i
2. Solve ODE with these values: $f(x|v_i)$
3. Find solution at final point x_f
4. Solve $f(x_f|v_i) - v_f = 0$ using root finding methods.

Relaxation method

1. Guess entire solution satisfying boundary conditions
2. 'Relax' trial solution to actual solution

Eigenvalue problems may be made into boundary value problem by treating the eigenvalue as an additional variable.

Matrix method

Another common technique:

Discretization: Transform ODE to matrix equation

Example

Consider the boundary value problem

$$y''(x) = f(x)\,, \qquad y(a) = Y_a, \quad y(b) = Y_b\,.$$

Divide $[a, b]$ into N sub-intervals, with $N + 1$ equally spaced points.

$$x_i = a + i\delta\,, \quad y_i = y(x_i) \quad \delta = \frac{b-a}{N}\,, \quad i = 0, \ldots, N\,.$$

Replacing $y''(x)$ with discrete derivative, we get

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{\delta^2} = f(x_i)\,, \quad y_0 = Y_a, \;\; y_N = Y_b\,.$$

A system of linear equations, i.e., a matrix equation.

Matrix equation

Obtained system of $N-1$ linear equations:

$$y_{i-1} - 2y_i + y_{i+1} = \delta^2 f(x_i) \equiv \hat{f}_i\,, \quad i = 1, \ldots, N-1\,. \qquad (*)$$

We can write this in matrix form

$$\begin{pmatrix} -2 & 1 & 0 & \ldots\ldots & & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & -2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-1} \end{pmatrix} = \begin{pmatrix} \hat{f}_1 - Y_a \\ \hat{f}_2 \\ \hat{f}_3 \\ \vdots \\ \hat{f}_{N-1} - Y_b \end{pmatrix}$$

Boundary conditions enter first and last equations:

$$y_0 - 2y_1 + y_2 = \hat{f}_i \qquad \Longrightarrow \qquad -2y_1 + y_2 = \hat{f}_1 - y_0 = \hat{f}_1 - Y_a$$

Matrix equation

Discretized to $N+1$ points, with $N-1$ interior points.
The boundary values of $y(x)$ are known (Dirichlet boundary conditions).

$\Longrightarrow \qquad (N-1) \times (N-1)$ matrix.

Could solve by inverting matrix:

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-1} \end{pmatrix} = \begin{pmatrix} -2 & 1 & 0 & \ldots\ldots & & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & -2 \end{pmatrix}^{-1} \begin{pmatrix} \hat{f}_1 - Y_a \\ \hat{f}_2 \\ \hat{f}_3 \\ \vdots \\ \hat{f}_{N-1} - Y_b \end{pmatrix}$$

Calculating inverse of matrix explicitly $\longrightarrow$

- inefficient
- usually unnecessary
- but works for moderate N.

Neumann boundary conditions

The boundary conditions were: $y(a) = Y_a$, $y(b) = Y_b$

Values of function given at boundary $\longrightarrow$ Dirichlet boundary conditions

If Derivatives given at boundary? $\longrightarrow$ Neumann boundary conditions

E.g., $y'(a) = \xi_a$ given instead of $y(a)$.

We used for first equation:

$$y_0 - 2y_1 + y_2 = \hat{f}_1 \qquad \Longrightarrow \qquad -2y_1 + y_2 = \hat{f}_1 - y_0 = \hat{f}_1 - Y_a$$

No longer works, y_0 not known. Need additional equation for y_0.

Use a finite difference formula for $y'(x)$:

Option 1: forward difference:

$$\xi_a = (y_1 - y_0)/\delta$$

$$\Longrightarrow \quad -y_1 + y_2 = \hat{f}_1 + \delta\xi_a$$

Problem: an $O(\delta)$ approximation.

Destroys $O(\delta^2)$ accuracy of complete procedure

Neumann boundary conditions

$y'(a) = \xi_a$ given. For the first equation, $y_0 - 2y_1 + y_2 = \hat{f}_1$, we need additional equation for y_0.

Use a finite difference formula for $y'(x)$:

Option 2: Use a second-order forward difference: $$\xi_a = \frac{4y_1 - y_2 - 3y_0}{2\delta}$$

Option 3: Use a second-order centred difference: $$\xi_a = \frac{y_1 - y_{-1}}{2\delta}$$

Problem: a fictional external point ($x_{-1} = a - i\delta$) is introduced. Need equation for y_{-1} as well. Can use

$$y_{-1} - 2y_0 + y_1 = \hat{f}_0$$

Linear algebra

Starting from a boundary value problem we ended up with a linear algebra problem!

$$Ay = b \quad \underset{\text{formally}}{\Longleftrightarrow} \quad y = A^{-1}b$$

The problem is 'equivalent' to inverting the matrix A

Matrix problems appear in

- solving sets of linear equations
- static solutions of pdes
- quantum mechanics: single-particle, many-particle, many-spin,...
- nonlinear or correlated curve fitting
- .....

Related problems: calculating eigenvalues and eigenvectors, eg $H\Psi = E\Psi$

Solving linear sets of equations — Methods

- Direct elimination methods
  - Gauss–Jordan, LU decomposition, QR, Cholesky
  - Works with all kinds of matrices but best for small — usually, matrix has to fit in memory
- Iteration
  - Jacobi, Gauss–Seidel, overrelaxed Gauss–Seidel
  - Write $Ax = (E - F)x = b$ where E is easily invertible
  - Iterate $x^{(n+1)} = E^{-1}(Fx^{(n)} + b)$
  - Requires diagonally dominant matrices, can be arbitrarily large
  - In slides 07
- Krylov subspace methods — e.g., Conjugate gradient
  - When system is so big that only sparse matrices can be used
  - Does not require A to be known explicitly, only the vector multiplication $y = Ax$

Gaussian elimination

We want to find the x_i in the equation(s)

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

We can

- interchange any two rows of A, b
- replace any row by a linear combination of itself and another
- interchange columns of A and the corresponding rows of x

The most naïve method uses just the second operation

First and third: pivoting

Gaussian elimination without pivoting

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

$$\longrightarrow \quad \begin{bmatrix} 1 & a_{12}/a_{11} & a_{13}/a_{11} & a_{14}/a_{11} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1/a_{11} \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

$$\longrightarrow \quad \begin{bmatrix} 1 & a_{12}/a_{11} & a_{13}/a_{11} & a_{14}/a_{11} \\ 0 & a_{22} - \frac{a_{21}a_{12}}{a_{11}} & a_{23} - \frac{a_{21}a_{13}}{a_{11}} & a_{24} - \frac{a_{21}a_{14}}{a_{11}} \\ 0 & . & . & . \\ 0 & . & . & . \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1/a_{11} \\ b_2 - \frac{a_{21}b_1}{a_{11}} \\ . \\ . \end{bmatrix}$$

Without pivoting

$$\longrightarrow \quad \begin{bmatrix} 1 & a_{12}/a_{11} & a_{13}/a_{11} & a_{14}/a_{11} \\ 0 & 1 & \cdot & \cdot \\ 0 & 0 & 1 & \cdot \\ 0 & 0 & 0 & \tilde{a}_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1/a_{11} \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \tilde{b}_4 \end{bmatrix}$$

We carried out forward elimination. Matrix now in upper triangular form.

We can obtain the x_i by back-substitution:

$$x_4 = \tilde{b}_4/\tilde{a}_{44}, \qquad x_3 = \tilde{b}_3 - \tilde{a}_{34}x_4 \quad \text{etc}$$

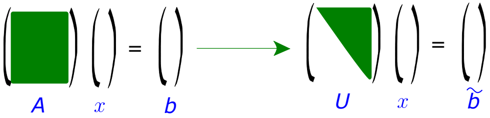
Problems: doesn't work if there's a zero on the diagonal.
Also unstable to rounding error.
Pivoting: swap rows so largest available element appears on diagonal

Gaussian elimination is LU decomposition

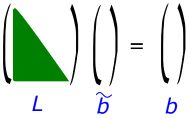
Solving $Ax = b$:
Gaussian elimination + back-substitution can be rewritten as

1. $A = LU$
2. $LUx = b \implies Ux = L^{-1}b = \tilde{b}$
3. $x = U^{-1}\tilde{b}$

First two steps are equivalent to forward elimination



Calculating $\tilde{b} = L^{-1}b$ means solving $L\tilde{b} = b$.
L is triangular; so this is forward substitution.
No explicit matrix inversion



Third step is back-substitution

Matrix inverse U^{-1} is not explicitly formed.

LU decomposition

If $Ax = b$ has to be solved for many different b vectors:

- Pre-compute $A = LU$
  E.g., Crout's algorithm or Doolittle's algorithm
- Calculate $x = U^{-1}(L^{-1}b)$ for each b.
  No explicit matrix inversion, because L, U are triangular.
  Instead, forward substitution or back-substitution

LU decomposition is not unique

Either L of U can be specified to have 1's on the diagonal
$\rightarrow$ unique decomposition

Price of Direct method: Operation count and storage

Operation count

- Forward elimination $\sim N^3$
  Back-substitution $\sim N^2$, negligible in comparison
- Alternative count: LU decomposition $\sim N^3$
  Forward substition (solve $L\tilde{b} = b$) or back-substitution (solve $Ux = \tilde{b}$) $\sim N^2$, negligible in comparison
  This is why pre-computing LU decomposition can make sense

Storage (fast memory or RAM)

A is stored and modified OR L and U are stored

$\Longrightarrow$ Limited by RAM size ($N \approx 10^4$ on typical 2020 desktops)

Seriously inadequate for many problems, even boundary-value ODE's

Krylov subspace methods

When N is too big to hold full matrix A in memory

but not too big for matrix-vector multiplications: $A\mathbf{x}$

Conjugate gradient

The same algorithm as in multidimensional minimisation of quadratic function

Minimise quadratic form $\quad f(x) = \frac{1}{2}(x, Ax) - (x, b)$

$\Longrightarrow$ find x for which $\quad \nabla f = Ax - b$ vanishes

$\Longrightarrow$ solve $\quad Ax = b$

Idea

- start with some guess x_0, search direction p_0
- minimise f along p : $f(x_1) = f(x_0 + \alpha p_0) = \min$
- find new direction p_1 so that $\min f(x_1 + \alpha_1 p_1)$ also minimises $f(x_0 + \lambda_1 p_0 + \lambda_2 p_1)\,;\forall \lambda_1, \lambda_2$: conjugate directions
- find new conjugate direction $p_2, \ldots,$ iterate till $\|\nabla f\|^2 < \varepsilon$

Such directions turn out to be conjugate or A-orthogonal: $\boxed{(p_1, Ap_2) = 0}$

Conjugate gradient

Algorithm

Minimize successively along conjugate directions, $(\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n)$:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}_k$$

$$\mathbf{p}_0 = \mathbf{r}_0$$

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$$

Defined $\mathbf{r}_k = -\nabla f\left(\mathbf{x}^{(k)}\right) = \mathbf{b} - A\mathbf{x}^{(k)}$

$$\beta_k = -\frac{\mathbf{r}_{k+1}^{\mathrm{T}} \cdot A \cdot \mathbf{p}_k}{\mathbf{p}_k^{\mathrm{T}} \cdot A \cdot \mathbf{p}_k}$$

$$\alpha_k = \frac{\mathbf{r}_k^{\mathrm{T}} \cdot \mathbf{r}_k^{\mathrm{T}}}{\mathbf{p}_k^{\mathrm{T}} \cdot A \cdot \mathbf{p}_k}$$

$\mathbf{r}_k$'s are called residuals

- Reach minimum in N steps. In practice, much fewer steps.
- Don't have to modify A, or maybe not even store A
  All we need to do with A is multiply vectors
  $\rightarrow$ suitable for using sparse storage

Krylov subspace

$$
\begin{aligned}
&\mathbf{y}\\
&A\mathbf{y}\\
&A^2\mathbf{y} = A(A\mathbf{y})\\
&A^3\mathbf{y} = A(A^2\mathbf{y})\\
&\vdots\\
&A^{k-1}\mathbf{y}
\end{aligned}
$$

CG builds on this set of vectors

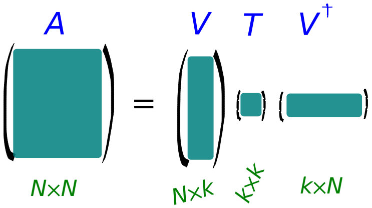
Subspace spanned by these vectors:
Krylov subspace

Orthonormal basis using Gram-Schmidt

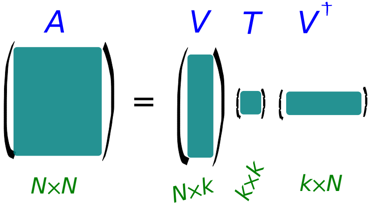
Iterative algorithms with $k \ll N$:

Represent the $N \times N$ problem as a $k \times k$ problem within subspace

Keep growing k until representation gives accurate enough answer.



Conjugate gradient, in Krylov subspace language



Tiny T matrix is the representation of huge A matrix

Solve the problem within (tiny) Krylov subspace: $T\mathbf{y} = V^\dagger \mathbf{b}$

Rotate solution $\mathbf{y}^*$ back to original (huge) space: $\mathbf{x}^* = V\mathbf{y}^*$

$\rightarrow$ entirely different interpretation of CG algorithm!

Krylov subspace

Other algorithms based on Krylov subspace

- Many others:
  gmres, biconjugate gradient, bicgstab, minimal residual, . . .

- Several of these are implemented in scipy: scipy.sparse.linalg.isolve
  (Direct solvers are in scipy.sparse.linalg.dsolve)

- Krylov subspace methods also for eigenvalues/eigenvectors
  - Trivial version: power method
  - More useful: Lanczos and Arnoldi algorithms:

$$\boxed{\text{Diagonalize the } T \text{ matrix}}$$

$\rightarrow$ recovers some eigenvalues of A

Sparse matrices

We had the equation

$$y_{i+1} - 2y_i + y_{i-1} = f_i \Longrightarrow \mathbf{Ay} = \mathbf{f} \quad \text{with} \quad A_{ij} = \delta_{i,j-1} - 2\delta_{ij} + \delta_{i,j+1}$$

This is a tridiagonal matrix
— a very common type in physical and mathematical problems
Other common types of matrices:

- band diagonal (with bandwidth M)
- tridiagonal with fringes (eg the two-dim Laplace operator)
- cyclic tridiagonal or banded (with fringes)
- band triangular
- block diagonal
- block tridiagonal
- block triangular
- singly/doubly bordered block diagonal

Sparse matrices

Sparse storage can save space (RAM)

- avoid wasting storage with 64-bit zeros — only store nonzero elements. Some bookkeeping necessary
- Can store sparse matrices with considerably more than $\sim 10^8$ elements, even on average laptop/desktop
- Various formats in use [NR 2.7] (E.g., save triplets (i, j, A_{ij}))

Sparse storage can save computation time

Number of compute operations could be reduced from N^3 to N^2

Avoid storing matrix?

Often, all one needs is to generate Ax for various vectors x.
Maybe don't need to create matrix A explicitly in memory?

- supply function that takes vector $\mathbf{x}$ and outputs vector $A\mathbf{x}$
- could even be the same function for different matrix sizes

Sparse matrices in Python

Scipy provides data structures and routines for sparse matrices:

```
import scipy.sparse as sparse
```

Various different types for sparse matrix available.

- bsr matrix: Block Sparse Row matrix
- coo matrix: A sparse matrix in COOrdinate format.
- csc matrix: Compressed Sparse Column matrix
- csr matrix: Compressed Sparse Row matrix
- dia matrix: Sparse matrix with DIAgonal storage
- dok matrix: Dictionary Of Keys based sparse matrix.
- lil matrix: Row-based linked list sparse matrix
- spmatrix: This class provides a base class for all sparse matrices.

Sparse matrices in Python

Some sparse types are better for constructing and others better for computation.

- Construction: coo matrix, dok matrix, lil matrix
- Computation: bsr matrix, csc matrix, csr matrix

Example of construction with coo matrix

```
row = np.array([0, 3, 1, 0])
col = np.array([0, 3, 1, 2])
data = np.array([4.0, 5.0, 7.0, 9.0])
A = sparse.coo matrix((data, (row, col)), shape=(4, 4)))
print(A)
```

gives:
(0, 0) 4
(3, 3) 5
(1, 1) 7
(0, 2) 9

Sparse matrices in Python

Other useful operations:

- Convert to dense matrix using todense.
- Convert to csr or csc using tocsr and tocsc for fast arithmetic.
- Look at non zeros using plt.spy from matplotlib.

Given a matrix A and a vector b, we wish to find an x that solves:

$$Ax = b$$

When A is dense we use solve from numpy:

```
import numpy as np
x = np.linalg.solve(A,b)
```

When A is sparse we use solve from scipy:

```
import scipy.sparse as sparse
x = sparse.linalg.spsolve(A,b)
```

Sparse matrices in Python

For constructing banded matrices spdiags is very useful:

Example

```
import scipy.sparse as sparse
import numpy as np
data = -np.ones((3,4))
data[1,:] *= -2
A = sparse.spdiags(data, [-1,0,1], 4, 4)
print(A.todense())
```

Summary

- Many boundary-value problems can be discretised:
  - Turn ODEs into matrix equations
  - Works also for many PDE's (multidimensional BVP's)
- General methods for solving matrix equation $Ax = b$:
  - Direct elimination (Gauss–Jordan etc)
  - Krylov subspace based iteration for sparse matrices
  - Direct Iteration (Jacobi, Gauss–Seidel)
    for diagonally dominant matrices (next slides)
- Sparse matrices appear in many physical problems
  - Huge savings in storage and computation
  - Many numerical methods are tailor-made for sparse matrices