Slide Type

Solving linear systems

Slide Type

Gaussian elimination

Slide Type Slide

- In linear algebra courses, you learn to solve linear systems of the form
$$Ax = v$$
using Gaussian elimination.
- This works pretty well in many cases. Let's do an example based on Newman's `gausselim.py`, for
$$A = \begin{bmatrix} 6 & 5 \\ 4 & 3 \end{bmatrix}, \qquad v = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Slide Type Sub-Slide

Refresher on Gaussian elimination (how `gausselim` works): the equation we need to solve is
$$\begin{bmatrix} 6 & 5 \\ 4 & 3 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$
and therefore
$$\begin{aligned} 6x_1 + 5x_2 &= 2, \\ 4x_1 + 3x_2 &= 1 \end{aligned}$$

1. Divide 1st line by 1st (top-left) coefficient:
$$\begin{aligned} x_1 + \frac{5}{6}x_2 &= \frac{1}{3}, \\ 4x_1 + 3x_2 &= 1 \end{aligned}$$

Slide Type Sub-Slide

2. $4 \times \text{1st eqn} - \text{2nd eqn} = \text{new 2nd eqn.}$:
$$\begin{aligned} x_1 + \frac{5}{6}x_2 &= \frac{1}{3}, \\ 0x_1 + \frac{1}{3}x_2 &= \frac{1}{3}, \end{aligned}$$
and $x_2 = 1$. More eqns $\Rightarrow$ cancel all 1st coefficents of each line similarly.
3. (if more eqns: repeat from 2nd line to eliminate all 2nd coefficients below, and so on...)
4. (or 3.) Back-substitute: $x_2 = 1 \Rightarrow x_1 + 5/6 = 1/3 \Rightarrow x_1 = -1/2$.

In [1]:

Slide Type Sub-Slide

```python
# textbook's Gaussian Elimination code
from numpy import array, empty

def gausselim(A, v):
    N = len(v)
    # Gaussian elimination
    for m in range(N):
        # Divide by the diagonal element
        div = A[m, m]
        A[m, :] /= div
        v[m] /= div
        # Now subtract from the lower rows
        for i in range(m+1, N):
            mult = A[i, m]
            A[i, :] -= mult*A[m, :]
            v[i] -= mult*v[m]
    # Backsubstitution
    x = empty(N, float)
    for m in range(N-1, -1, -1):
        x[m] = v[m]
        for i in range(m+1, N):
            x[m] -= A[m, i]*x[i]
    print(x)
```

In [2]:

Slide Type Sub-Slide

```python
import numpy as np

A1 = np.array([[6, 5], [4, 3]], float)
V1 = np.array([2, 1], float)

gausselim(A1, V1)
```

```
[-0.5  1. ]
```

Slide Type

When Gaussian elimination breaks down

Slide Type Slide

The example below is a valid system but the original code will "break".

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}, \qquad v = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

In theory, $x \approx \begin{pmatrix} -1 \\ 1 \end{pmatrix}$. But according to `gausselim`:

In [3]:

Slide Type Fragment

```python
A2 = np.array([[1e-20, 1], [1, 1]], float)  # I imported np earlier
V2 = np.array([1, 0], float)

gausselim(A2, V2)
```

```
[0. 1.]
```

Slide Type Sub-Slide

Don't divide by (close to) zero!

- Had the top-left number actually been zero, Python would have thrown a `ZeroDivisionError`,
- with 10^{-20} < machine precision, no tripwire from Python, but rounding errors.

In [4]:

Slide Type Sub-Slide

```python
# numpy gives the same wrong result!
np.linalg.solve(A2, V2)
```

Out[4]:

```
array([0., 1.])
```

In [5]:

Slide Type Fragment

```python
# SciPy does not give a better result either ... but at least it gives a
import scipy.linalg as la
la.solve(A2, V2)
```

```
/tmp/ipykernel_173/930078458.py:3: LinAlgWarning: Ill-conditioned matri
x (rcond=1e-40): result may not be accurate.
  la.solve(A2, V2)
```

Out[5]:

```
array([0., 1.])
```

Slide Type

Partial pivoting:

Slide Type Sub-Slide

- Eliminates the issue of dividing by zero if diagonal entries become zero (or very close to zero)

Algorithm outline:

1. At m^{th} row, check to see which of the rows below has the largest m^{th} element (absolute value)
    - Swap this row with the current m^{th} value
    - Proceed with Gaussian elimination

]:

Slide Type Sub-Slide

```python
A3 = np.array([[1, 1], [1e-20, 1]], float)  # swapped rows
V3 = np.array([0, 1], float)  # need to swap this one too

gausselim(A3, V3)
```

```
[-1.  1.]
```

LU Decomposition

Slide Type

Motivation

Slide Type

Suppose you have a system $Ax = f$ where f depends on some parameter of a physical system. When you change the parameter, f changes, but A doesn't. You don't want to re-do the entire Gaussian Elimination procedure each time you changet the parameter.

Slide Type Sub-Slide

The steps in the Gaussian elimination will always be the same: only need to do it once, then store.

Gaussian elimination on a matrix A can be written as a series of matrix multiplications that yields $U = L_n L_{n-1} \cdots L_0 A$, where U is upper triangular (i.e., result of Gaussian elimination):

$$L^{-1} = L_n L_{n-1} \cdots L_0 \Rightarrow \; Ax = LUx = f.$$

(see Newman pp. 222-224 for a 4×4 example; easy but long)

The decomposition

$$\boxed{LU = A}$$

is called the "LU decomposition" of the matrix A.

Slide Type Sub-Slide

How to use LU in practice

- Suppose you know L, U from A.
- Then,
$$Ax = f \Leftrightarrow Ux = L^{-1}f.$$
- Break down into **two triangular-matrix problems,** $Ux = y$ **and** $Ly = f$.
- Triangular $\Rightarrow$ back-substitution (pizza cake!)
- This method is used by `numpy.linalg.solve(A, f)`
- `scipy.linalg.lu_solve(scipy.linalg.lu_factor(A), f)` is equivalent to `numpy.linalg.solve(A, f)`, but intermediate steps give access to the decomposition and allow storage.
- Once you've done the LU decomposition of A, you don't need to do it again $\Rightarrow$ f can change over and over, $Ly = f$ is staightforward, and so is $Ux = y$.

Slide Type Sub-Slide

Issues with LU Decomposition

LU Decomposition fails when A is close to singular, due to rounding error again.

For starters, take a matrix that is actually singular, e.g:

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

Depending on the RHS, we end up with either no solution, or one undetermined coefficient.

$A\binom{x_1}{x_2} = \binom{3}{5}$ $\rightarrow$ can't have $x_1 + 2x_2 = 3$ and $= 5/2$ at the same time

$A\binom{x_1}{x_2} = \binom{3}{6}$ $\rightarrow$ infinite number of solutions

Slide Type Sub-Slide

So, LU won't find a solution when there is none: not really a drawback. But what about

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 4+\delta \end{bmatrix},$$

with δ very small compared to other coefficients? Not singular, but LU won't work if δ is too small.

Slide Type Sub-Slide

```
# delta = 1e-16
# A = np.array([[1, 2], [2, 4+delta]], float)
# print(A)
# v = np.array([3, 5], float)
# np.linalg.solve(A, v)  # returns error if delta/4 < machine precision
```

QR decomposition for eigensystems

Slide Type Slide

Looking for λ's and v's such that $Av = \lambda v$, with v eigenvector, λ eigenvalue

Or for Λ and V such that $AV = V\Lambda$, with V orthonormal matrix of eigenvectors, Λ diagonal matrix of eigenvalues

If A is square and **either symmetric-real or Hermitian** (complex), we can solve this problem with a QR decomposition.

Don't get hung up on the details of the algorithm description below. Recall that it's iterative, and that it can break sometimes.

QR algorithm

Slide Type Sub-Slide

- Gram-Schmidt on columns of A (Exercise 6.8) $\Rightarrow$ matrix of orthonormal basis of column vectors Q
- Denote QR decomposition of A as $A = QR$, where R is upper-triangular
- Q orthonormal $\Rightarrow$ $Q^TQ = I \;\Rightarrow\; R = Q^TA$.

Iterate:

- $A_1 = RQ = Q^TAQ$ --------------> Define A_1
- $A_1 = Q_1R_1$ -------------------------> QR decomposition of A_1
- $A_2 = R_1Q_1 = \underbrace{Q_1^T \underbrace{Q^TA}_{R} QQ_1}_{R_1}$ --> Define A_2
- $A_2 = Q_2R_2$ -------------------------> QR decomposition of A_2
- $A_3 = \ldots$ ... and so on, until obtaining an A_k such that all off-diagonal terms are small enough.

Slide Type Sub-Slide

- Eventually, "it can be proven" that this iteration converges to a (near-)diagonal output

$$A_k = \underbrace{(Q_k^T \cdots Q_1^T Q^T)}_{V^T \text{ (because } Q_i^T Q_i = I)} A \underbrace{(QQ_1 \cdots Q_k)}_{V}$$

$$\Rightarrow A_k = V^TAV \;\Rightarrow\; \boxed{AV = VA_k}.$$

- diagonal entries of A_k (off-diagonal entries are now tiny) are the eigenvalues: $\boxed{A_k = \Lambda}$.
- The eigenvectors are the columns of $\boxed{V = QQ_1 \cdots Q_k}$

`numpy.linalg` implements the QR algorithm in the `numpy.linalg.eigh` function.

Slide Type

Using QR in practice

Slide Type Sub-Slide

```
A = np.array([[2, 1], [1, 2]])  # imported numpy as np earlier

print('A:\n', A)

eig_vs, V = np.linalg.eigh(A)  # calculate eigenvalues & eigenvectors
L = np.diag(eig_vs)  # np.diag constructs a diagonal array

print('\neigenvalues: ', eig_vs)
print('eigenvectors:\n', V)

# we expect that AV = VD
print('\nAV:\n', np.dot(A, V))
print('VL:\n', np.dot(V, L))
```

Slide Type

When QR breaks

Slide Type Sub-Slide

- `eigh` takes only Hermitian or real symmetric matrices as input
- What happens if we try a different (non-symmetric) matrix?

$$A = \begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}$$

Slide Type Fragment

```
A = np.array([[2,3],[1,2]])
eig_vs, V = np.linalg.eigh(A) # calculate eigenvalues & eigenvectors
L = np.diag(eig_vs)  # np.diag constructs a diagonal array

print('AV:\n', np.dot(A, V))
print('VL:\n', np.dot(V, L))
```

Slide Type Sub-Slide

Result above should show that with A is not a symmetric matrix: $AV \neq \Lambda V$. There are less efficient algorithms that will work with non-symmetric A.

Slide Type Fragment

```
A = np.array([[2,3],[1,2]])
eig_vs, V = np.linalg.eig(A) # calculate eigenvalues & eigenvectors
L = np.diag(eig_vs)  # np.diag constructs a diagonal array

print('AV:\n', np.dot(A, V))
print('VL:\n', np.dot(V, L))
```

Finding roots of nonlinear equations

Slide Type Slide

The textbook discusses several methods. We'll summarize a few of them.

Slide Type

Relaxation

Slide Type Sub-Slide

The guinea piggies' favourite method for everything!

- Solving for x in an equation $x = f(x)$
- Guess an initial value x_0 and iterate until the function converges to a fixed point

$$\begin{aligned} x_1 &= f(x_0) \\ x_2 &= f(x_1) \\ &\vdots \end{aligned}$$

- Caveat: Can only find *stable* fixed points

Slide Type

Newton's Method

Slide Type Sub-Slide

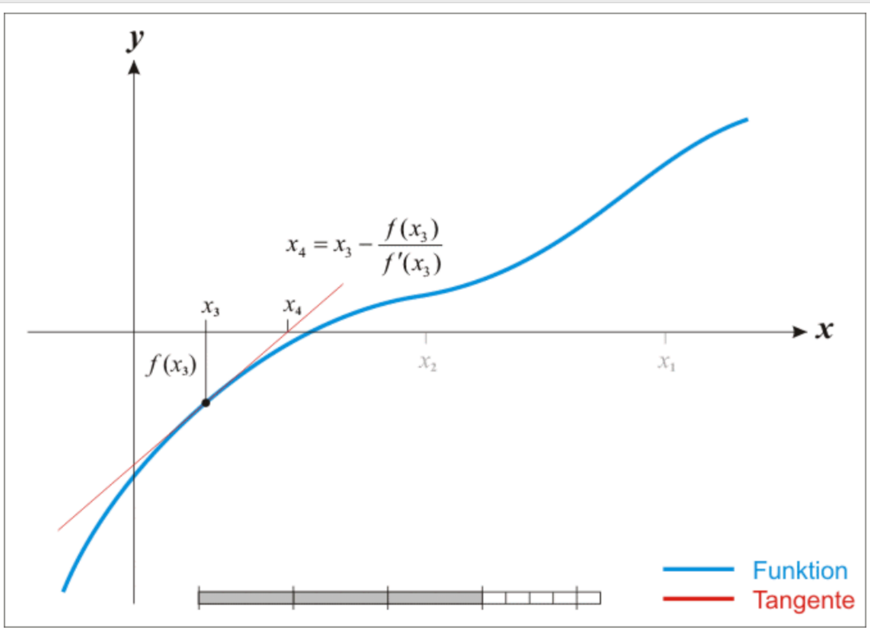
Solving for x in $f(x) = 0$

1. Start with some value x_1, calculate tangent $f'(x_1)$
2. Travel along tangent line to intersection with x-axis at x_2
3. Repeat (calculate tangent $f'(x_2)$, etc.)

Mathematically:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Slide Type Sub-Slide



Using Newton's Method in practice

Slide Type

`Secant method" variation on Newton's Method: If analytic form of f is unknown, calculate $f'(x)$ numerically

- Suggest using forward or backward difference, to avoid re-computing yet another $f(x_k)$

Slide Type Sub-Slide

Pro:

- Much faster than relaxation

Cons:

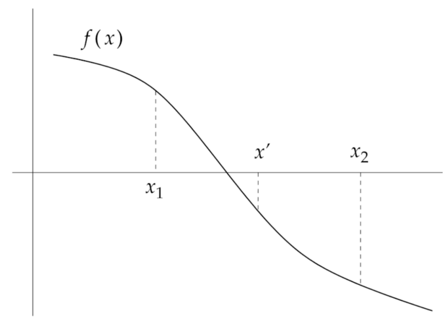
- Need to know f' (although this issue is addressed by the secant method)
- Doesn't always converge
  - need to have good initial guess (like relaxation),
  - small f' gives x_{n+1} much farther away,
  - sometimes, it just does not converge. Period. (e.g., fractals)

Slide Type

Bisection (or Binary Search)

Slide Type Sub-Slide

- Bracket a single root on either side of the zero of the function (x_1, x_2)
- Use midpoint x' as subsequent bracket
- Choose brackets depending on the sign of the value at the midpoint;
  - For this example, $f(x_1) > 0$, $f(x') < 0$, so the next set of brackets is (x_1, x')



Slide Type

Convergence and Usage

Slide Type Sub-Slide

Pro:

- Incredibly easy to remember, therefore to implement
- When there's a root, there's a way (no worries about converging towards at least *a* root)

Cons:

- Only works with a single bracketed root
- Can't find "double roots" where $f(x)$ reaches but does not cross 0 (think $f(x) = x^2$)
- Can't find even one root when there is an even number of roots.
- Large sample intervals can "miss" roots
- Slower than Newton

Slide Type

Method	**Formula**	**Convergence Test**
Relaxation	$\epsilon = \frac{x-x'}{1-1/f'(x)}$	Taylor expansion, assuming proximity to root
Newton	$\epsilon = x - x'$, $O\left(\epsilon_0^{2^N}\right)$	Taylor expansion about solution of f(x)=0
Binary Search	$\epsilon = \frac{\Delta}{2N}$	Error x1/2 each iteration

Slide Type

Finding minima/maxima

Slide Type Slide

Again, there are several methods. We'll focus on the Golden Ratio search.

Slide Type

Golden Ratio search

Slide Type Sub-Slide

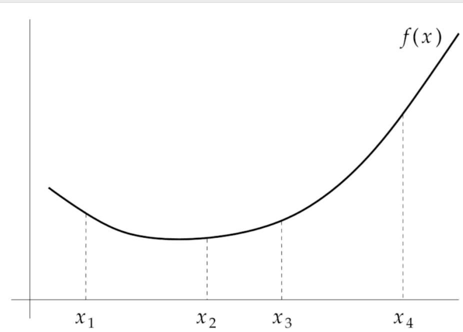
Similar to binary search: find minimum by shrinking intervals

1. Start with 2 points x_1, x_4 bracketing the interval
2. Choose 2 points x_2, x_3 inside the interval

- Check which of $f(x_2)$ and $f(x_3)$ is lower to determine new bracke
  - In this example: $f(x_2) < f(x_3) \Rightarrow$ new interval is $[x_1, x_3]$

Use the golden ratio to determine the most optimal placement of the int

Slide Type Sub-Slide



Why the Golden Ratio?

Slide Type Sub-Slide

- Interior points x_2, x_3 must be symmetric about the midpoint of the interval (why favour one side vs. the other?)
- If you place interior points close (distance ϵ) to the centre of interval:
  - you'll divide your search interval by ≈ 2 (very good), but
  - next step will be difficult: new "interior" point will be bar from new centre, next step will only divide the search interval by $\approx 1 - \epsilon$ (very bad)
- If you place interior points close to edges creates the opposite: first step very bad, next step very good
- **Solution**: find sweet spot(s) to make sure the search interval is divided by same ratio each time.
- See pp. 281-282 of textbook for explanation of why this ratio needs to be
$$z = \frac{1+\sqrt{5}}{2} \approx 1.618$$
the *Golden Ratio*. I.e., $x_4 - x_1 = (x_3 - x_1)z = (x_4 - x_2)z$.

Slide Type

Summary

Slide Type Slide

Finding solutions of linear systems $Ax = v$

- Gaussian elimination if you know the 1st coefficient will always be OK
- Partial pivot to be safe: re-order equations such that biggest first coefficient shows up first
- LU decomposition is strictly the same as Partial pivot, but storing L and U saves times when A stays the same but v changes often

Slide Type Fragment

Finding eigenvalues/eigenvectors

- Matrix is real symmetric or Hermitian: QR algorithm is iterative (can take time to converge) but efficient.
- Otherwise: SciPy will find them for you if patient.

Slide Type Sub-Slide

Finding roots of nonlinear equations

$$f(x) = 0 \quad \text{or} \quad f(x) - x = 0$$

- Relaxation for $f(x) = x$ is easy but works only for stable fixed points
- Newton's method is super fast but you need a good initial guess and confidence that a root exists
- Binary search is easy, converges slowly, has a lot of caveat (double roots, need a good initial bracket...)

Slide Type Fragment

Finding minima/maxima: Golden ratio search, slow and suffers from same limitations as binary search, but works.

LU, then QR

```
import numpy as np
from scipy.linalg import lu, lu_factor, lu_solve

# Example matrix A and right-hand side f
A = np.array([[4, 3, 0],
              [3, 2, -1],
              [0, -1, 1]], dtype=float)

f = np.array([7, 4, -1], dtype=float)

# Step 1: Perform LU Decomposition
lu, piv = lu_factor(A)

# Step 2: Solve the system using the LU decomposition
x = lu_solve((lu, piv), f)

# Print the solution
print("Solution x:", x)

# Verification (optional)
# Multiply A with x to ensure it matches f
print("Verification A @ x:", A @ x)
print("Original f:", f)
```

```
import numpy as np

# Define the matrix
A = np.array([[4, 1, 1],
              [1, 3, 2],
              [1, 2, 2]], dtype=float)

# Use NumPy's eigh for symmetric or Hermitian matrices
eigenvalues, eigenvectors = np.linalg.eigh(A)

print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)

# Verification: A @ v = λ * v
print("Verification for first eigenvector:")
λ = eigenvalues[0]
v = eigenvectors[:, 0]
print("A @ v:", np.dot(A, v))
print("λ * v:", λ * v)
```