5 Linear Systems: Direct Methods

In this chapter we solve linear equations $Ax = b$ when A is a real, square, and nonsingular matrix and b is a vector. Such problems arise frequently in all branches of science, engineering, economics, and finance. There is no single technique that is best in all cases. Most methods can be divided into two classes: *iterative methods* and *direct methods*. In this chapter we study direct methods; in the absence of roundoff error, such methods would yield the exact solution in a finite number of steps. The basic direct method is Gaussian elimination; the bulk of the algorithm involves only the matrix A and amounts to its decomposition into a product $A = LU$.

5 Linear Systems: Direct Methods

- Solve $Ax = b$, when A is square and nonsingular.
- Gaussian Elimination (Chapter 5) is a direct method.
- The Jacobi method (Chapter 7) is an iterative method.

5 Linear Systems: Direct Methods

Solve the system

$$
\begin{aligned}
2x - 3y &= 2 \\
5x - 6y &= 8
\end{aligned}
$$

by

- augmenting and Gaussian Eliminating (as in your linear algebra course) to the equivalent system.
-

$$
\begin{aligned}
2x - 3y &= 2 \\
\frac{3}{2}y &= 3
\end{aligned}
$$

5 Linear Systems: Direct Methods

Solve the system

$$2x - 3y = 2$$
$$5x - 6y = 8$$

by

- factoring A as A=LU and then forward and back substituting.
- $\begin{bmatrix} 2 & -3 \\ 5 & -6 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{5}{2} & 1 \end{bmatrix} \begin{bmatrix} 2 & -3 \\ 0 & \frac{3}{2} \end{bmatrix}.$

5 Linear Systems: Direct Methods

Solve the system

$$\begin{aligned} 2x_1 + x_2 &= 0 \\ x_1 + 2x_2 + x_3 &= -3 \\ x_2 + 2x_3 &= -2 \end{aligned}$$

by finding an LU decomposition and then forward substituion followed by backward substitution.

5.1 Algorithm: Backward Substitution

Solve $Ux = b$ for x when U is $n \times n$ by the backward substitution algorithm.

```
for k in range: $(n-1, -1, -1)$ :
        $x_k = \frac{b_k - \sum_{j=k+1}^{n} U_{kj} x_j}{U_{kk}}.$
```

- Try to understand this algorithm one row k at a time.
- Implement $\sum_{j=k+1}^{n}$ with a loop and assignment update.
- Use data structure Arrays for matrices from section 2.4 Comp. Physics.

5.1 Algorithm: Backward Substitution in Python

```
for i in range(n-1,-1,-1):
    x[i] = b[i]
    for j in range (i+1,n):
        x[i] -= U[i,j]*x[j]
    x[i] = x[i]/U[i,i]
print(x)
```

- Can you see how this program implements the algorithm?
- Wait a day and try to rewrite the code without looking.

5.1 Algorithm: Backward substitution (Example)

```
import numpy as np

U = np.array([[3, 2, 1],
             [0, 2 , -2],
             [0, 0, 5]])
n = 3 # size of A is 3x3

b = np.array([2,6,-10])  # U and b from classwork
x = np.zeros(n)  # empty vector x to hold answer

for i in range(n-1,-1,-1): # loop backwards from end
    x[i] = b[i]
    for j in range (i+1,n): # each column greater i
        x[i] -= U[i,j]*x[j]
    x[i] = x[i]/U[i,i]
print(x)
```

5.1 Cost of backward substitution

- $1+\sum_{k=1}^{n-1} 1+((n-k)+(n-k)+1) = \sum_{k=1}^{n}(2(n-k)+1) = n^2$.
- noting that B.S is $\mathcal{O}(n^2)$ and not $\mathcal{O}(n^3)$ is often helpful. Memorize it.
- There is also forward substitution which shares order, $\mathcal{O}(n^2)$.

5.1 Algorithm: Backward Substitution

Write code to solve $Lc = b$ for c when L is $n \times n$ lower triangular by the forward substitution algorithm.

- Try your code when $L = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}$ and $b = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$.
- Check your code by solving by hand.

5.2 Algorithm: LU decompostion

Set up

```
# ALGORITHM: LU decomposition from p.103

import numpy as np

A = np.array([[1, 2, 1],
              [3, 8 , 1],
              [0, 4, 1]],float)
n = 3 # size of A is 3x3
L = np.identity(n)
for j in range(n-1): # loop each column j from 0 to (n-2)
    for i in range(j+1,n): # loop rows
        L[i,j] = (A[j+1,j]/A[j,j]) # multiplier = L[i,j]
        A[i,:] = A[i,:] - L[i,j]*A[j,:] # subtract multipli

print(A)
print(L)
```

5.2 Cost of LU decomposition

- $2\sum_{k=1}^{n-1} n^2 \approx \frac{2}{3}n^3 = \mathcal{O}(n^3)$.
- Comparing to the cost of back substitution we see that the cost of the elimination phase dominates for all but very small problems.

5.2 LU decomposition

Forming A^{-1} explicitly and multiplying by b is generally not recommended. For one, it can be wasteful in storage. Moreover, it is more computationally expensive than LU decomposition, though by less than an order of magnitude. Also it may give rise to a more pronounced presence of roundoff errors. Finally, it simply has no advantage.

5.2 LU decomposition

It is also not recommended to solve Ax=b by computing determinants, cofactors, or

5.3 Pivoting Strategies

$$\begin{cases} x_1 + x_2 + x_3 = 1 \\ x_1 + x_2 + 2x_3 = 2 \\ x_1 + 2x_2 + 2x_3 = 1 \end{cases}$$

5.3 Pivoting Strategies

$$\begin{cases} x_1 + x_2 + x_3 = 1 \\ x_1 + 1.0001x_2 + 2x_3 = 2 \\ x_1 + 2x_2 + 2x_3 = 1 \end{cases}$$

- Exact solution to 5 digits: $x \approx (1, -1.0001, 1.0001)$.
- G.E solution to 3 digits without row interchanges: $x \approx (0, 0, 1)$.
- G.E solution to 3 digits with row interchanges: $x \approx (1.000, -1.000, 1.000)$.

5.3 Pivoting Strategies: Example

Solve $\begin{bmatrix} 3 & 1 & 2 \\ 6 & 3 & 4 \\ 3 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 3 \end{bmatrix}$ by finding the $PA = LU$ factorization (with partial pivoting) and then carrying out forward and backward substitution.

- $P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 1 & 1 \end{bmatrix}, U = \begin{bmatrix} 6 & 3 & 4 \\ 0 & -.5 & 0 \\ 0 & 0 & 3 \end{bmatrix}.$
- You will only be asked to do PA = LU factorization by hand in this course.

5.3 Pivoting Strategies: Example

Elimination with partial pivoting to factor A into PA = LU is an important algorithm. This is how a computer solves a general linear equation $Ax = b$, more or less. This is a far different approach than the theoretical approach $x = A^{-1}b$ you learned in your linear algebra course.

5.4 Efficient Implementation

We will not cover this section. There will be no quiz or test questions from this section. Computer science majors will find it to be interesting reading.

5.5 Cholesky Factorization

- skip section 5.3.
- symmetric: $A^T = A$.
- positive definite: all eigenvalues $\lambda_i > 0$.

5.5 Cholesky Factorization

- for a positive definite, symmetric matrix A do Cholesky factorization.
- $A = LU = LD\tilde{U} = LDL^T$
- and then $A = GG^T$ when $G = LD^{\frac{1}{2}}$.

5.5 Cholesky Factorization

You will only need to be able to Cholesky factorize a 2×2 matrix on quizzes or exams by hand.

- Example: $A = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}$.
- Solve $\begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix} = GG^T = \begin{bmatrix} g_{11} & 0 \\ g_{21} & g_{22} \end{bmatrix} \begin{bmatrix} g_{11} & g_{21} \\ 0 & g_{22} \end{bmatrix}$ for G.

5.6 Sparse Matrices

You need not read section 5.6. You will not be tested on it. However do think about how you would alter our Gaussian elimination, and back substitution code to solve the 100×100 system Ax=b when

$$A = \begin{bmatrix} 3 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 3 & -1 & \ddots & & & & \vdots \\ 0 & -1 & 3 & -1 & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & -1 & 3 & -1 & 0 \\ \vdots & & & & \ddots & -1 & 3 & -1 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 3 \end{bmatrix}, b = \begin{bmatrix} 2 \\ 1 \\ 1 \\ \vdots \\ \vdots \\ \vdots \\ 1 \\ 2 \end{bmatrix}.$$

5.6 Sparse Matrices

Continuation of previous slide. You can use the numpy commmand

```
from scipy.sparse import diags
A=diags([-1,3,-1],[-1,0,1],shape=(100,100)).toarray()
```

The exact solution is $x = (1, 1, \ldots, 1)$.

5.7 Permutation and ordering strategies

We will not cover this section this semester.

5.8 Errors and the condition number.

Suppose by some algorithm we have computed an approximate solution $\hat{x}$ to $Ax = b$. We are concerned with the following error measurements.

- absolute error: $||x - \hat{x}||$.
- relative error: $\frac{||x-\hat{x}||}{||x||}$.
- residual: $\hat{r} = b - A\hat{x}$.
- The residual is an important measure of error because it is easily computed, whereas the absolute and relative errors are impossible to compute. We will therefore use the residual often in analyzing our algorithms.

5.8 Errors and the condition number.

Solve $Ax = b$ when $A = \begin{bmatrix} 1.2969 & .8648 \\ .2161 & .1441 \end{bmatrix}$ and $b = \begin{bmatrix} .8642 \\ .1440 \end{bmatrix}$.

- using some algorithm $\hat{x} = \begin{bmatrix} .9911 \\ -.4870 \end{bmatrix}$.
- residual $\hat{r} = \begin{bmatrix} -10^{-8} \\ 10^{-8} \end{bmatrix}$. So $||\hat{r}||_\infty = 10^{-8}$ is small.
- However the exact solution is $x = \begin{bmatrix} 2 \\ -2 \end{bmatrix}$, so $||x - \hat{x}||_\infty = 1.513$.

5.8 Condition number and relative error estimate

- $\hat{r} = b - A\hat{x} = Ax - A\hat{x} = A(x - \hat{x})$.
- $x - \hat{x} = A^{-1}\hat{r}$.
- This gives us an important bound on the absolute error:

$$\|e\| = \|x - \hat{x}\| \leq \|A^{-1}\|\|\hat{r}\|.$$

  You should memorize it.

5.8 Condition number and relative error estimate

Since the relative error is more often used than the absolute error, we compute the following bound using the previous slide together with the fact that $Ax = b$ is equivalent to $x = A^{-1}b$.

- Combine $\|x - \hat{x}\| = \|A^{-1}\|\|\hat{r}\|$
- and $\|x\| = \|A^{-1}b\| \leq \frac{\|b\|}{\|A\|}$
- to get the important bound on the relative error:
$$\frac{\|x - \hat{x}\|}{\|x\|} \leq \|A^{-1}\|\|\hat{r}\|\frac{\|A\|}{\|b\|}.$$

5.8 Condition number and relative error estimate

The previous bound is important and comes with some new vocabulary.

- Condition number of square matrix A is $\kappa(A) = ||A|| ||A^{-1}||$.
- relative forward error: $\frac{||x-\hat{x}||}{||x||}$.
- relative backward error: $\frac{||\hat{r}||}{||b||}$.
- KEY BOUND: $\frac{||x-\hat{x}||}{||x||} \leq \kappa(A)\frac{||\hat{r}||}{||b||}$.

5.8 Error and condition number

The equation

$$\frac{||x - \hat{x}||}{||x||} \leq \kappa(A) \frac{||\hat{r}||}{||b||}$$

is extremely important in numerical linear algebra. You should spend some time thinking about it. Essentially the product of the condition number, $\kappa(A)$, and the relative backward error bound the relative forward error. Since we want the relative forward error to be small, we often use this equation to bound the relative forward error by approximating the right side.

5.8 Error and condition number

If the condition number, $\kappa(A)$, is large the equation

$$\frac{||x - \hat{x}||}{||x||} \leq \kappa(A) \frac{||\hat{r}||}{||b||}$$

does not provide a small bound on the relative forward error, even when the relative backward error is tiny. We call such a problem with large condition number *ill-conditioned*. The condition number $\kappa(A)$ measures the sensitivity of a problem: If A and b are slightly changed how does the solution $x = A^{-1}b$ change? Is the change in x great (ill-conditioned) or small (well-conditioned)?
In the course of each problem all involved decide what is considered "large" and what is considered "tiny".

5.8 Errors and the condition number.

Solve $Ax = b$ when $A = \begin{bmatrix} 1.2969 & .8648 \\ .2161 & .1441 \end{bmatrix}$ and $b = \begin{bmatrix} .8642 \\ .1440 \end{bmatrix}$.

- $A^{-1} = 10^8 \begin{bmatrix} .1441 & -.8648 \\ -.2161 & 1.2969 \end{bmatrix}$
- condition number: $\kappa(A) = 2.1617 \cdot 1.513 \cdot 10^8 \approx 3.27 \times 10^8$.

5.8 Error and condition number

- You should memorize $\kappa_2(Q) = 1$ when Q is orthogonal.
- You should memorize $\kappa_2(A) = \frac{\lambda_1}{\lambda_n}$ when A is symmetric positive definite.
- However in general, the norm and condition number are not computed in practice, only estimated. There is not enough time to solve the eigenvalue problem and get $\kappa_2(A) = \sqrt{\frac{\lambda_1(A^T A)}{\lambda_n(A^T A)}}$.

5.8 Error and condition number Python

Numpy provides an algorithm, *cond*, to approximate the condition number of a matrix. We will not program this algorithm in this class ourselves. Here is how to use it.

- from numpy import linalg as LA
- LA.cond(a) $= \kappa_2(a)$ and LA.cond(a, np.inf) $= \kappa_\infty(a)$.
- Python can also compute norms of matrices $\|A\|_2 =$ LA.norm(a) and $\|A\|_\infty =$ LA.norm(a, np.inf)