AMSC460 Computational Methods

Michael Li

Contents

1 Taylor Polynomials

We can approximate the value of $f(x)$ with a point x_0 near x

$$p_n(x) = f(x_0) + f'(x_0)(x - x_0) + \ldots + f^{(n)}(x_0)\frac{(x - x_0)^n}{n!}$$

- we have to know the values of $f(x_0), f'(x_0), \ldots$

Remainder term $R_{n+1} = f(x) - p_n(x)$ can be expressed as

$$R_{n+1} = f^{(n+1)}(t)\frac{(x - x_0)^{n+1}}{(n+1)!} \quad \text{for } t \text{ between } x, x_0$$

Example: approximate $\sqrt{9.1}$ such that $|\tilde{y} - y| \leq 10^{-6}$

1. let $x_0 = 9 \implies f(9) = 3, f'(9) = 1/6, f''(9) = 1/108$
2. $\tilde{y} = p_2(x_0) = f(9) + f'(9)(9.1 - 9) + f''(9)\frac{1}{2}(9.1 - 9)^2 \approx 3.01662037037$
3. $R_{n+1} = R_3 = f'''(t)\frac{(9.1-9)^3}{3!} = \frac{3}{8}\frac{1}{6}\frac{1}{10^3}t^{-5/2}$
4. since t is between 9.1 and 9, and $t^{-5/2}$ is decreasing, we have $|t^{-5/2} \leq 9^{-5/2}$. Thus

$$|R_3| \leq \frac{3}{8}\frac{1}{6}\frac{1}{10^3}9^{-5/2} \approx 2.572 \cdot 10^{-7}$$

   And we have that $|\tilde{y} - y| \leq 2.573 \cdot 10^{-7} \leq 10^{-6}$ as desired

2 Origin of Errors

For simple problems,we can find a formula such as

$$I_1 = \int_0^1 \sin(x)\,dx = -\cos(1) + \cos(0) = 1 - \cos(1)$$

We can then use I_1 to approximate an answer ≈ 0.45969769413186

However, for complicated problems such as

$$I_2 = \int_0^1 \sin(\sin(x))\,dx$$

We can't create a simple formula. However, we can approximate (in this case with Reimann Sums with interval $n = 1000$)

$$\hat{I}_2 = \frac{1}{1000}\sum_{j=1}^{1000} \sin(\sin((j - \frac{1}{2})\frac{1}{1000})) \approx 0.430606129785715$$

Things to consider

- how large is the error between $\hat{I}_2$ and I_2
- how fast does error decrease if we increase n

Absolute Error: $|\hat{x} - x|$

Relative Error: $\epsilon_x = \frac{\hat{x}-x}{x} \implies \hat{x} = x(1 + \epsilon_x)$

2.1 Sources of Errors

Take the example problem of dropping a mass from height $h = 5$ft. We want to find t_0, the time for the mass to hit the floor. Using Newton's law, we can approximate

$$t_0 = \sqrt{\frac{2h}{g}} = \sqrt{\frac{5}{16}}$$

However, suppose that our calculator doesn't have a sqrt function. In this case, we need to approximate $\sqrt{\frac{5}{16}}$

```
x = 1
for i = 1 to 5:
  x = (x + a/x)/2
```

Using this approximation, we get that $t_0 \approx 0.5590169944$. However, this still might be an accurate value of t_0

- **Error in given data**: input values might not be accurate. Even if we don't need to approximate calculations, the error in the input data can propagate throughout the entire problem, polluting the final answer
- **Modeling error**: we don't consider all factors in a given problem (e.g. assuming that the only force acting on the mass is $-mg$)
- **Approximation error**: there are some problems we can't solve, so we have to approximate an answer
- **Roundoff error**: computers can only operate on a finite number of digits, which creates roundoff error

3 Error Propagation and Roundoff Error

Most problems have a certain number of inputs $x_1, \ldots, x_n$ and a certain number of output values $y_1, \ldots, y_m$

Consider the simplest case of 1 input and 1 output value: $y = f(x)$

If we only have an approximation $\tilde{x}$ of input value x, the best we can do is compute $\tilde{y} = f(\tilde{x})$.. This gives a relative error of

$$\epsilon_y = \frac{\tilde{y} - y}{y} = \frac{f(\tilde{x}) - f(x)}{f(x)} \approx \frac{(\tilde{x} - x)f'(x)}{f(x)} = \frac{xf'(x)}{f(x)} \cdot \frac{\tilde{x} - x}{x} = c_f \cdot \epsilon_x$$

- $c_f(x) = \frac{xf'(x)}{f(x)}$ is the **condition number** of f at x and determines how sensitive a problem is to small changes to input values
  - **Well-conditioned**: $|c_f|$ is not much larger than 1
  - **Ill-conditioned**: $|c_f| \gg 1$

Example: $f(x) = \frac{1}{x}$

$c_f(x) = -1$ so f is well conditioned for all x. Taking $x = 2$ and $\hat{x} = 1.96$ we have that

$$x = 2 \quad \hat{x} = 1.96 \quad \epsilon_{\hat{x}} = \frac{\hat{x} - x}{x} = -0.02$$

$$y = f(x) = 0.5 \quad \hat{y} = f(\hat{x}) \approx 0.5102 \quad \epsilon_{\hat{y}} = \frac{\hat{y} - y}{y} \approx 0.0204$$

Thus $|\epsilon_{\hat{x}}| \approx |\epsilon_{\hat{y}}|$

Example: $f(x) = \ln x$

$c_f(x) = \frac{1}{\ln x}$ so for $x = 1.01$, we have that $c_f(x) \approx 100$. Using the Taylor approximation of $\ln x \approx 0 + 1(x-1)$ for x and choosing $x = 1.01$ and $\hat{x} = 1.02$, we have that

$$x = 1.01 \quad \hat{x} = 1.02 \quad \epsilon_{\hat{x}} = \frac{\hat{x} - x}{x} \approx 0.0099$$

$$y = f(x) \approx 0.00995 \quad \hat{y} = f(\hat{x}) \approx 0.0198 \quad \epsilon_{\hat{y}} = \frac{\hat{y} - y}{y} \approx 0.99$$

Thus $|\epsilon_{\hat{y}}| \approx 100\epsilon_{\hat{x}}$

3.1 Error Propagation for Arithmetic Operations

Let x, y be exact values and let $\tilde{x}, \tilde{y}$ be approximate values

- **Product**: $z = x \cdot y \implies \tilde{z} = \tilde{x}\tilde{y} = x(1+\epsilon_x)y(1+\epsilon_y) = z(1+\epsilon_x+\epsilon_y+\epsilon_x\epsilon_y) \implies \epsilon_z = \epsilon + \epsilon_y + \epsilon_x\epsilon_y \approx \epsilon_x + \epsilon_y$
- **Division**: $\epsilon_z \approx \epsilon_x + \epsilon_{1/y}$ using $f(x) = 1/x$
- **Sum**: $z = x + y \implies \epsilon_z = \frac{x(1+\epsilon_x)+y(1+\epsilon_y)-(x+y)}{x+y} = \frac{x}{x+y}\epsilon_x + \frac{y}{x+y}\epsilon_y \implies |\epsilon_z| \leq \left|\frac{x}{x+y}\right| |\epsilon_x| + \left|\frac{y}{x+y}\right| |\epsilon_y|$
  - if $x \approx -y$, we have that $\left|\frac{x}{x+y}\right|$ and $\left|\frac{y}{x+y}\right|$ are much greater than 1, and cause a larger magnification of relative errors. This is called **subtractive cancellation**

    Example: $x = 101, y = -100 \implies \frac{x}{x+y} = 101, \frac{y}{x+y} = -100$
  - if x, y are opposite signs but different magnitude, $\left|\frac{x}{x+y}\right|$ and $\left|\frac{y}{x+y}\right|$ are not much larger than than 1. This is fine

    Example: $x = 10, y = -1 \implies \frac{x}{x+y} = \frac{10}{9}, \frac{y}{x+y} = \frac{-1}{9}$
  - if x, y have the same sign, $\left|\frac{x}{x+y}\right|$ and $\left|\frac{y}{x+y}\right|$ are less than 1. Added together, they give 1. Thus $|\epsilon_z| \leq \max(|\epsilon_x|, |\epsilon_y|)$

    Example: $x = 3, y = 7 \implies \frac{x}{x+y} = \frac{3}{10}, \frac{y}{x+y} = \frac{7}{10}$

  UPSHOT: None of these operations cause significant magnification of relative errors, except **subtractive cancellation**

3.2 Forward Error Analysis

We try to find the upper bound for relative error at each stage of the algorithm

- start with the bounds for the errors in the given data
- when a function f is applied, we multiply the error bound by $|c_f|$
- when arithmetic operators are used on the values, we use the above error propagation formulas
- each time we round, we add $|\epsilon_M|$ to the error bound

Example: $y = f(x) = 1 - \cos x$ for $x = 10^{-5}$ using double precision machine numbers

$c_f = \frac{x \sin x}{1 - \cos x} \approx \frac{x \cdot x}{x^2/2}$ so the function is **well-conditioned** and has unavoidable error of $|c_f|\epsilon_M + \epsilon_M \approx 3 \cdot 10^{-16}$

In contrast, consider the algorithm $y_1 = \cos x, \quad y = 1 - y_1$. Using machine arithmetic, we have

$\hat{x} = fl(x), \quad \tilde{y}_1 = \cos\hat{x}, \quad \hat{y}_1 = fl(\tilde{y}), \quad \tilde{y} = 1 - \hat{y}_1, \quad \hat{y} = fl(\tilde{y})$. This gives relative errors of

$\left|\frac{\hat{x}-x}{x}\right| \leq \epsilon_M, \quad \left|\frac{\tilde{y}_1 - y_1}{y_1}\right| \leq c_1\epsilon_M$ with $c_1 = \left|\frac{x(-\sin x)}{\cos x}\right| \approx 10^{-10}, \quad \left|\frac{\hat{y}_1 - y_1}{y_1}\right| \leq c_1\epsilon_M + \epsilon_M$

$\left|\frac{\tilde{y} - y}{y}\right| \leq c_2(c_1\epsilon_M + \epsilon_M)$ with $c_2 = \left|\frac{y_1(-1)}{1-y_1}\right| \approx 2 \cdot 10^{10}$

Thus we have

$$\left|\frac{\hat{y}-y}{y}\right| \leq c_2(c_1\epsilon_M + \epsilon_M) + \epsilon_M \approx 2\epsilon_M + 2\cdot 10^{10}\epsilon_M + \epsilon_M \approx 2\cdot 10^{-6}$$

Which is much larger than the unavoidable error, and thus this algorithm is **numerically unstable**

4 Unavoidable Error, Numerically Stable/Unstable Algorithms

We want to compute $f(x)$ for an input x. Assuming there are no measurement errors, in the **ideal algorithm** we would

- round the input to the closest machine number $\hat{x} = fl(x)$
- use $\hat{x}$ and compute $\tilde{y} = f(\hat{x})$ with some extra precision
- round this result: $\hat{y} = fl(\tilde{y})$

This results in **unavoidable error**:

$$|\epsilon_{\hat{y}}| \leq |c_f(x)|\epsilon_M + \epsilon_M$$

In typical computations, we break up the computation of $f(x)$ into a sequence of operations available on our machine

For example, we can use the algorithm we used from the previous section to estimate $y = 1 - \cos(x)$ at $x = 5\cdot 10^{-4}$

```
format compact; format long g
x = 5e-4;
yalg1 = 1 - cos(x)
  > 1.24999997352937e-07
```

Relative Error

We can find relative error by using symbolic toolbox and vpa to use 30 digits accuracy

```
X = sym(5*10^-4)
Y = vpa(1 - cos(X))
  > Y = 0.000000124999997395833355034722125341O2
relerr_alg1 = double(yalg1-Y)/Y
  > relerr_alg1 = -3.4317095325145e-10
```

Thus relative error is about $3.4\cdot 10^{-10}$

Unavoidable error

```
cf = x*sin(x)/(1-cos(x))
  > cf = 1.99999995901967
epsM = 1e-16;
unav_err = cf*epsM + epsM
  > unav_err = 2.99999995901967e-16
```

Since the error from algorithm 1 is $3.4\cdot 10^{-10}$ which is much larger than the unavoidable error $3\cdot 10^{-16}$, algorithm 1 is **numerically unstable**. This large error comes from a few factors:

Note that $y_1 = \cos(x)$ is rounded to the closest machine number, resulting in an error as large as ϵ_M

When we compute $y = 1 - y_1$ (note that $y_1 \approx 1$), we have **subtractive cancellation** so ϵ_{y_1} will be multiplied by $\left|\frac{y_1}{1-y_1}\right|$

```
y1 = cos(x);
factor = abs(y1/(1-y1))
  > factor = 7999999.16941204
```

```
factor*epsM
  > ans = 7.99999916941204e-10
```

Thus the roundoff error in y_1 may cause an error as large as $8 \cdot 10^{-10}$, corresponding to the order of magnitude in the error from algorithm 1

A better algorithm would be to multiply and divide $1 - \cos x$ by $1 + \cos x$, then replacing $1 - \cos^2 x$ using $\sin^2 x + \cos^2 x = 1$

$$y = 1 - \cos x = \frac{(1 - \cos x)(1 + \cos x)}{1 + \cos x} = \frac{\sin^2 x}{1 + \cos x}$$

This algorithm is free from subtractive cancellation. Thus

```
yalg2 = sin(x)^2/(1+cos(x))
  > yalg2 = 1.24999997395833e-07
relerr_alg2 = double(yalg2-Y)/Y
  > relerr_alg2 = 5.55505114272964e-18
```

This is the same order of the unavoidable error, or less. Thus algorithm 2 is **numerically stable**

5 Machine Numbers and Machine Arithmetic

A Matlab program such as

```
x = 0.1;
y1 = cos(x);
y = y - y1
```

is not evaluated exactly. Machines can only store up to a certain number of digits fro each number and use **machine arithmetic**

5.1 Base 10 Machine Numbers

Numbers are represented in decimal notation. An $(n+1)$-digit base 10 number with digits $d_j \in \{0, \dots, 9\}$ is

$$(d_0.d_1d_2 \dots d_n)_{10} = d_0 + d_1 \cdot 10^{-1} + d_2 \cdot 10^{-2} + \dots + d_n \cdot 10^{-n}$$

We can normalize numbers to the form $x = \pm q \cdot 10^e$ where

- q is the **mantissa**
- e is the **exponent**
- the first digit d_0 is nonzero

For example, we can represent $x = 12345$ as

$$x = 12345 = 1.2345 \cdot 10^4$$

Thus simple base 10 machine numbers can be presented as

$$\hat{x} = \begin{cases} \pm(d_0.d_1d_2 \dots d_n)_{10} \cdot 10^e & d_j \in \{0, \dots, 9\}, \quad d_0 \neq 0, \quad e \in Z, \quad e_{\min} \leq e \leq e_{\max} \\ 0 \end{cases}$$

- largest machine number is $x_{\max} = (9.99 \dots 9)_{10} \cdot 10^{e_{\max}} = (10 - 10^{-n}) \cdot 10^{e_{\max}} \approx 10^{e_{\max}+1}$
- smallest positive machine number is $x_{\min} = (1.00 \dots 0) \cdot 10^{e_{\min}} = 10^{e_{\min}}$

We can round a number $\hat{x} = fl(x)$

- normalize x into the form $\pm q \cdot 10^e$
- if $e_{\min} \le e \le e_{\max}$, find the nearest mantissa $\hat{q} = (d_0.d_1 d_2 \dots d_n)_{10}$ then set $\hat{x} = \pm\hat{q} \cdot 10^e$
- if $e > e_{\max}$ then **overflow** occurs
- if $e < e_{\min}$ then **underflow** occurs so let $\hat{x}$ be the closer of 0 or $x_{\min}$

Example: assume we have a machine with $n = 3, e_{\min} = -99$, and $e_{\max} = 99$. Find $\hat{x} = fl(x)$ for $x = \frac{2}{300}$

We have that $x = +\frac{20}{3} \cdot 10^{-3}$, so $q = \frac{20}{3}$ and $e = -3$, meaning that we don't have overflow or underflow

To approximate the mantissa $q = \frac{20}{3}$, we note that $\frac{20}{3} = 6.6666\dots$

- $\hat{q}_{\text{left}} = (6.666)$
- $\hat{q}_{\text{left}} = (6.667)$ (closer so we round to this)

Thus we have that

$$\hat{x} = fl(x) = +(6.667)_{10} \cdot 10^{-2}$$

To find the upper bound for rounding error, note that

$$\left|\frac{\hat{x} - x}{x}\right| = \frac{|\hat{q} \cdot 10^e - q \cdot 10^e|}{q \cdot 10^e} = \frac{|\hat{q} - q|}{q}$$

Since $q \ge 1$ and $|\hat{q} - q| \le 0.5 \cdot 10^{-n}$ (spacing between 2 successive mantissa values is 10^{-n} and the largest possible value is half of this), rounding error is bounded by

$$\left|\frac{\hat{x} - x}{x}\right| = \frac{|\hat{q} - q|}{q} \le 0.5 \cdot 10^{-n}$$

Thus **machine epsilon** for base 10 system is $\epsilon_M = 0.5 \cdot 10^{-n}$

5.2 Base 2 Machine Numbers

Numbers are represented in binary notation. An $(n+1)$-digit base 2 number with digits $d_j \in \{0, \dots, 1\}$ is

$$(d_0 \cdot d_1 d_2 \dots d_n)_2 = d_0 \cdot 2^0 + d_1 \cdot 2^{-1} + \dots + d_n \cdot 2^{-n}$$

We can normalize numbers to the form $x = \pm q \cdot 2^e$ where

- q is the **mantissa**
- e is the **exponent**
- the first digit d_0 is nonzero

For example, we can represent $x = (1101)_2$ as

$$x = (1101)_2 = (1.101)_2 \cdot 2^3$$

Thus simple base 2 machine numbers can be presented as

$$\hat{x} = \begin{cases} \pm(1.d_1 d_2 \dots d_n)_2 \cdot 2^e & d_j \in \{0, \dots, 1\}, \quad e \in Z, \quad e_{\min} \le e \le e_{\max} \\ 0 \end{cases}$$

- largest machine number is $x_{\max} = (1.1\dots1)_2 \cdot 2^{e_{\max}} = (2-2^{-n}) \cdot 2^{e_{\max}} \approx 2^{e_{\max}+1}$
- smallest positive machine number is $x_{\min} = (1.00\dots0)_2 \cdot 2^{e_{\min}} = 2^{e_{\min}}$

We can round a number $\hat{x} = fl(x)$

- normalize x into the form $\pm q \cdot 2^e$
- if $e_{\min} \le e \le e_{\max}$, find the nearest mantissa $\hat{q} = (d_0.d_1d_2\dots d_n)_2$ then set $\hat{x} = \pm\hat{q} \cdot 2^e$
- if $e > e_{\max}$ then **overflow** occurs
- if $e < e_{\min}$ then **underflow** occurs so let $\hat{x}$ be the closer of 0 or $x_{\min}$

Example: How does Matlab represent $x = .1$ Matlab uses binary machine numbers with $n = 53, e_{\min} = -1021, e_{\max} = 1024$. Find $\hat{x} = fl(x)$ for $x = \frac{1}{10}$

We have that $x = +\frac{16}{10} \cdot 2^{-4}$, so $q = \frac{16}{10}$ and $e = -4$, meaning that we don't have overflow or underflow

To approximate the mantissa $q = \frac{16}{10}$, we note that $\frac{16}{10} = (1.10011001100\dots)_2$

- $\hat{q}_{\text{left}} = (1.10011001100\dots11001)_2$
- $\hat{q}_{\text{right}} = (1.10011001100\dots11010)_2$ (this is closer so we round to this)

Thus we have that

$$\hat{x} = fl(x) = +\hat{q}_{\text{right}} \cdot 2^{-3}$$

To find the upper bound for rounding error, note that

$$\left|\frac{\hat{x}-x}{x}\right| = \frac{|\hat{q} \cdot 2^e - q \cdot 2^e|}{q \cdot 2^e} = \frac{|\hat{q}-q|}{q}$$

Since $q \ge 1/2$ and $|\hat{q}-q| \le 0.5 \cdot 2^{-n}$ (spacing between 2 successive mantissa values is 2^{-n} and the largest possible value is half of this), rounding error is bounded by

$$\left|\frac{\hat{x}-x}{x}\right| = \frac{|\hat{q}-q|}{q} \le 0.5 \cdot 2^{-n}$$

Thus **machine epsilon** for base 2 system is $\epsilon_M = 2^{-n-1}$

5.3 IEEE654 Machine Numbers

The issue with base 2 machine numbers is that there is a huge hole around 0; the distance between 0 and $x_{\min}$ is much larger than the distance between $x_{\min}$ and the next largest number $x_1 = (1+2^{-n})x_{\min}$

- Rounding numbers $|x| > x_{\min}$ causes a relative error of size $\le \epsilon_M$. However, rounding numbers $|x| < x_{\min}$ gives either 0 or $x_{\min}$, causing relative error of size $\le 100\%$
- $y > x$ and $y - x > 0$ have different meanings. For machine numbers $x = x_{\min}$ and $y = x_1$
  - $y > x$ evaluates to true since $x_1 > x_{\min}$
  - $y - x > 0$ evaluates to false since the computer computes $y - x = x_1 - x_{\min} = 2^{-n} \cdot 2^{e_{\min}}$ which is rounded to 0

We fix this by adding **subnormal numbers** with spacing $2^{-n} \cdot 2^{e_{\min}}$

$$\pm(0.d_1\dots d_n)_2 \cdot 2^{e_{\min}}$$

Now rounding a number x with $|x| \le x_{\min}$ is more well-behaved

$$\left|\frac{\hat{x}-x}{x}\right| \leq \begin{cases} 2^{-n-1} & |x| \geq x_{\min} \\ \min(2^{-n-1}\frac{x_{\min}}{x}, 1) & |x| < x_{\min} \end{cases}$$

This results in **gradual underflow** since we generate values slightly smaller than $x_{\min}$

How to handle overflow, division by 0, 0/0, etc

We introduce a few numbers to handle the above issues

- $+Inf$ and $-Inf$ for handling overflow
- $+0$ and -0 to handle underflow, but still keep sign
- NaN for indeterminate expressions like $0/0$ or $Inf - Inf$

UPSHOT: IEEE754 machine numbers have the following form with $d_j \in \{0, 1\}$ and integer e

$$\hat{x} = \begin{cases} \pm(1.d_1 \dots d_n)_2 \cdot 2^e & e_{\min} \leq e \leq e_{\max} \\ \pm(0.d_1 \dots d_n)_2 \cdot 2^{e_{\min}} & \text{subnormal numbers} \\ Inf, -Inf, NaN & \text{special values} \end{cases}$$

5.4 Machine Arithmetic

For machine numbers x, y, $x + y$ is usually not a machine number. Thus the result is rounded to become a machine number

For operations like $y = sqrt(x)$ are implemented as follows: for an input x

- find the exact result $Y = \sqrt{x}$ (using extra digits)
- return the machine number $y = fl(Y)$

This usually results in an error $|\epsilon_y| \leq \epsilon_M$

UPSHOT: each arithmetic operation in a program causes a relative error of size $\leq \epsilon_M$

Example: for Matlab code $x = .1; y = 1 - cos(x)$, the machine performs the following operations to find $\hat{y}$

- $\hat{x} = fl(.1)$ round .1 to the closest machine number
- $Y_1 = \cos(\hat{x})$ find the true value of $\cos(\hat{x})$ with extra accuracy
- $\hat{y}_1 = fl(Y_1)$ round Y_1 to the closest machine number
- $Y = 1 - \hat{y}_1$ find the true result of $1 - \hat{y}_1$ with extra accuracy
- $\hat{y} = fl(Y)$ round Y to the closest machine number

6 Linear Systems

Example find x_1, x_2, x_3 such that

$$\begin{aligned} 2x_1 + 3x_2 + x_3 &= 1 \\ 4x_1 + 3x_2 + x_3 &= -2 \\ -2x_1 + 2x_2 + x_3 &= 6 \end{aligned}$$

We can write this using matrix-vector notation

$$\begin{bmatrix} 2 & 3 & 1 \\ 4 & 3 & 1 \\ -2 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 6 \end{bmatrix}$$

A matrix A is **singular** if there exists a nonzero vector x such that $Ax = 0$. Since Ax is a linear combination of the column vectors: $Ax = x_1 \cdot (\text{col } 1) + x_2 \cdot (\text{col } 2) + \ldots$

- a matrix is singular if the columns are linearly dependent (either no solution or infinitely many solutions)
- a matrix is nonsingular if the columns are linearly independent (has a unique solution)

Solving the example matrix above using **elimination** gives

$$\begin{bmatrix} 2 & 3 & 1 \\ 0 & -3 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -4 \\ 1 \end{bmatrix}$$

Note that the entries on the diagonal of the first matrix are called **pivots**

This translates to the following equations

$$\begin{aligned} 1x_3 &= 1 \\ -3x_2 - x_3 &= -4 \\ 2x_1 + 3x_2 + x_3 &= 1 \end{aligned}$$

Using **back substituion**, we get $x_3 = 1, x_2 = 1, x_3 = -3/2$

Note that the original linear system $Ax = b$ was transformed into a new linear system $Ux = y$ with an upper triangular matrix U. This means that original linear system is nonsingular

We can also use **LU Decomposition** to solve this equation

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & 3 & 1 \\ 4 & 3 & 1 \\ -2 & 2 & 1 \end{bmatrix}$$

Perform Gaussian Elimination on U and adjust L accordingly

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & 3 & 1 \\ 0 & -3 & -1 \\ 0 & 5 & 2 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -\frac{5}{3} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & 3 & 1 \\ 0 & -3 & -1 \\ 0 & 0 & \frac{1}{3} \end{bmatrix}$$

From here we can use forward substitution to solve $Ly = b$

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -\frac{5}{3} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 6 \end{bmatrix}$$

which gives $y_1 = 1, y_2 = -4, y_3 = \frac{1}{3}$.

The last step involves using **substitution** to solve for x in $Ux = y$, yielding $x_3 = 1, x_2 = 1, x_1 = -\frac{3}{2}$

The benefit of LU decomposition is that if we're solving multiple linear systems with the same matrix A ($Ax = b$ and $A\tilde{x} = \tilde{b}$), we only need to do Gaussian elimination once

- Use Gaussian elimination on matrix A to find LU decomposition $A = LU$
- solve $Ly = b$ using forward substitution, then solve $Ux = y$ using back substitution
- solve $L\tilde{y} = \tilde{b}$ using forward substitution, then solve $U\tilde{x} = \tilde{y}$ using back substitution

The pivot points of matrix U during Gaussian elimination must be nonzero. If it so happens that a pivot is 0, we need to perform **pivoting** (swapping the rows), creating a matrix $\tilde{A}$, and then solving $\tilde{A}x = \tilde{b}$ using LU decomposition

If pivoting breaks down at a particular column j (meaning that all pivot candidates in column j are zero), then the matrix A is **singular**

Note: should always select the pivot with the largest absolute value to avoid unnecessary error

6.1 Solving linear Systems in Matlab

To solve $Mx = b$ where M is in upper or lower triangular, we use $x = M\backslash b$ which will use back or forward substitution appropriately

To solve $Ax = b$

```
[L, U, p] = lu(A, 'vector'); % Gaussian elimination with pivoting, returns p as a vector
y = L\b(p);                  % forward substitution
x = U\y;                     % back sbustitution

x = A\b                      % executes the 3 commands above
```

To solve multiple linear equations $Ax = b$ and $A\tilde{x} = \tilde{b}$

```
[L, U, p] = lu(A, 'vector'); % Gaussian elimination with pivoting to find L, U, p
x = U\(L\b(p));              % use L, U, p to solve Ax = b
xh = U\(L\bh(p));            % use L, U, p to solve A xh = bh
```

6.2 Matrix Inverses

Another way to solve $Ax = b$ is to use inverse $x = A^{-1}b$

We can find A^{-1} by using Gaussian elimination to find L, U, p and then solving linear systems for n right-hand side vectors

$$\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \dots, \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

This will generate n solution vectors, who make up the columns of A^{-1}

6.3 Number of Operations

Elimination updates on a matrix U subtract multiples of the pivot role, so updates look like

$$u_{42} = u_{42} - l_{42} \cdot u_{22}$$

On a computer, this involves

- **Memorization Access**: need to access the values of u_{42}, l_{42}, u_{22}
- **Multiplication**
- **Addition/Subtraction**

Consider the following operations:

- **Finding** L, U, p costs $\frac{1}{3}n^3 + O(n^2)$ operations

  Elimination of $n-1$ columns costs $n(n-1) + (n-1)(n-2) + \cdots + 2 \cdot 1 = \frac{1}{3}n^3 + O(n^2)$
- **Solving** $Ax = b$ if we know L, U, p costs n^2 operations

  Solving $Ly = b$ and finding $y_1, y_2 \ldots, y_n$ cost $0 + 1 + \cdots + (n-1) = \frac{n(n-1)}{2}$ operations

  Solving $Ux = y$ and finding $x_n, x_{n-1}, \ldots, x_1$ costs $1 + 2 + \cdots + n = \frac{(n+1)n}{2}$ operations
- **Finding** A^{-1} costs $\frac{2}{3}n^3 + O(n^2)$ operations

  Finding first column costs $\frac{n(n-1}{2}$ for forward substitution and $\frac{(n+1)n}{2}$ for back substitution

  Finding first column costs $\frac{(n-1)(n-2}{2}$ for forward substitution and $\frac{(n+1)n}{2}$ for back substitution

  $\vdots$

  Total: $\frac{1}{6}n^3 + O(n^2)$ for forward substitution and $n\frac{(n+1)n}{2}$ for back substitution
- **Solving** $Ax = b$ if we know A^{-1} costs n^2 operations

  Matrix operations for $A^{-1}b$ take n^2 operations
- **UPSHOT**: finding L, U, p then solving $Ax = b$ is cheaper

6.4 Errors in Linear Systems

When solving $Ax = b$, we normally don't know the exact values of A and b, and have to use approximations $\hat{A}$ and $\hat{b}$ to solve $\hat{A}x = \hat{b}$. This can lead to a large amount of error. Consider

$$\begin{bmatrix} 1.01 & .99 \\ -.99 & 1.01 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

We can see that $x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ is the solution

Now consider $\hat{b} = \begin{bmatrix} 2.02 \\ 1.98 \end{bmatrix}$, then we get a solution vector $\hat{x} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$

To measure the error of vectors, we use **vector norm** $\|\mathbf{x}\|$ which has the following properties

- $\|\mathbf{x}\| = 0 \implies x = 0$
- $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$
- $\|\mathbf{x} + \mathbf{y}\| = \|\mathbf{x}\| + \|\mathbf{y}\|$

We also define 3 types of norms

- $\|\mathbf{x}\|_1 = |x_1| + \cdots + |x_n|$
- $\|\mathbf{x}\|_2 = (|x_1| + \cdots + |x_n|)^{1/2}$
- $\|\mathbf{x}\|_\infty = \max(|x_1|, \cdots, |x_d|)$

If a subscript isn't provided, then the equation is valid for all 3 norms

We can then calculate the **relative error** with respect to a vector norm as $\frac{\|\hat{\mathbf{x}}-x\|}{\|\mathbf{x}\|}$

To measure the error of matrix, we use a **matrix norm** $\|\mathbf{A}\|$ that has the property

- $\|\mathbf{Ax}\| \leq \|\mathbf{A}\|\|\mathbf{x}\|$

Where $\|\mathbf{x}\|$ is one of the vector norms and $\|\mathbf{A}\|$ satisfies

$$\|\mathbf{A}\| = \sup_{x \in R^n, x \neq 0} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} = \max_{x \in R^n, \|\mathbf{x}\|=1} \|\mathbf{Ax}\|$$

Depending on which type of vector is used, we have 3 matrix norms:

- $\|\mathbf{A}\|_\infty = \max_{i=1,\ldots,n} \sum_{j=1,\ldots,n} |a_{ij}|$ (max of row sums of absolute values)
- $\|\mathbf{A}\|_1 = \max_{j=1,\ldots,n} \sum_{i=1,\ldots,n} |a_{ij}|$ (max of column sums of absolute values)