CSC336: Numerical Methods

Allison Lau (Instructor: Christina Christara)

Fall 2023

Contents

1 Introduction **2**

2 Computer Arithmetic **2**
2.1 (Human) Representation of Nonnegative integers . . . 2
2.2 (Human) Representation of Reals . . . 3
2.3 Computer Representation of Numbers . . . 4
2.4 Data and Computational Error . . . 6

3 Solving Linear Systems **13**
3.1 Gaussian Elimination . . . 13
3.1.1 Forward substitution algorithm for $n \times n$ lower-triangular matrices . . . 13
3.1.2 Backward substitution algorithm for $n \times n$ upper-triangular matrices . . . 14
3.1.3 Gauss Elimination . . . 15
3.2 LU Decomposition . . . 16
3.2.1 Symmetric Matrices . . . 18
3.2.2 Symmetric Positive Definite Matrices . . . 18
3.2.3 Banded Matrices . . . 19
3.3 LU Factorization for Computing the Inverse of a Matrix . . . 19
3.4 Summary . . . 20
3.5 Gauss Elimination with Pivoting . . . 21
3.5.1 Scaling and GE/LU with Partial Pivoting . . . 23
3.5.2 Complete Pivoting . . . 24
3.5.3 Effect of Pivoting to Special Matrices . . . 24

A Gauss Elimination algorithms **25**

B Properties of Triangular Matrices **25**

C Gauss Elimination (GE) and LU factorization with Pivoting Example **26**

D Computer Arithmetic, Gauss Elimination (GE) and LU factorization Abridged **28**
D.1 Computer Arithmetic . . . 28
D.2 GE/LU and Operation Counts . . . 29

1 Introduction

Scientific computing involves observing phenomena and situations in the world, modelling with measurements which may involve physical laws, restrictions, assumptions and simplifications, building mathematical models with equations (where there might often not be an analytic solution), then simultion or approximate the solution with numerical methods or algorithms. This gives results in values or functions, which then goes on to post processing which might be data visualization. An important parameter is the size of the model. After this step, the data might be interpreted in real world scenaios and evaluated on their correctness, which might then return to the first step of the whole process. In this set of notes, we will discuss computer arithmetic, data and computational erros, solving linear and nonlinear equations and interpolation.

2 Computer Arithmetic

2.1 (Human) Representation of Nonnegative integers

In general, an integer x can be represented in a base b system, where $b \in \mathbb{Z}^+$ as follows:

Base b Integers

This is also the algorithm for converting base b integers to decimal.

$$\begin{aligned} x &= (d_n d_{n-1} \cdots d_0)_b \\ &= d_n \times b^n + d_{n-1} \times b^{n-1} + \cdots + d_0 \times b^0 \end{aligned}$$

where $0 \leq d_i < b$, $i = 0, \cdots, n$, $x \in \mathbb{Z}^+$. The digits used are $0, \cdots, n-1$

An example is the decimal system with base 10. For example,

$$350 = (350)_{10} = 3 \times 10^2 + 5 \times 10^1 + 0 \times 10^0$$

The digits used are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Today's computers are digital and work with pulses set by electric components. Thus all computers operate internally in the binary system ($d_k = 0$ or 1). User-wise they operate in decimal just for our convenience. Note that if several binary digits are grouped together, then a base b system, where $b = 2^k$ where $k \in \mathbb{Z}^+$ is simulated. An example is as follows

$$(1 \underbrace{0101}_{\text{5 in base 16}} \underbrace{1110}_{\text{E in base 16}})_2 = (15E)_{16}$$

To represent nonnegative and negative integers, we use the sign in front of the number digits, then lay out the digits in sequence. On the computer, the sign can be represented as an extra digit, 0 or 1.

Algorithm for converting decimal integers to base b

Initialize a list of remainders r. Divide decimal integer d by b to obtain quotient q and remainder r. Repeat the following steps until quotient equals 0:

1. Set $d = q$.
2. Divide q by b to obtain new quotient q and r and store r.

The list of remainders from bottom to top to obatin the number in base b.

An example of converting $(350)_{10}$ to $(15E)_{16}$ is as follows:

Numerator	Denominator	Quotient	Remainder
350	16	21	14 (= E)
21	16	1	5
1	16	0	1

2.2 (Human) Representation of Reals

Let x be a real number. Then $x = \pm(x_I.x_F)_b = \pm(d_n d_{n-1} \cdots d_0.d_{-1}d_{-2}\cdots)_b$, where x_I is the **integral** part and x_F is the **fraction**. (On the computer, the **sign** is represented by one digit, which is 0 or 1.) The integral part of a real number is represented as a nonnegative integer. The fraction, which may have infinite number of nonzero digits, is represented in a similar way by laying out its digits after the decimal point.

Base b Reals

This is also the algorithm for converting base b fractions to decimal. For base $b \in \mathbb{Z}^+$,

$$\begin{aligned} x_F &= (.d_{-1}d_{-2}d_{-3}\cdots)_b \\ &= d_{-1} \times b^{-1} + d_{-2} \times b^{-2} + d_{-3} \times b^{-3} + \cdots \\ &= \sum_{k=1}^{\infty} d_{-k} \times b^{-k} \end{aligned}$$

As an example, $(0.101)_{10}$ can be converted to base 2 in the following way.

$$(0.101)_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.5 + 0.125 = 0.625$$

An important note is that if x_F is a terminating binary fraction with n digits, the corresponding decimal fraction is also terminating in n digits. The oppositie is not necessarily true.

Algorithm for converting decimal fractions to base b

Initialize a list of integrals i. Multiply decimal fraction m by base b to obtain product p, and separate p into the integral part i and the fraction part f. Repeat the following steps until fraction equals 0:

1. Set $m = f$.
2. Multiply m by b to obtain p, i and f

The list of integrals from top to bottom is the base b fraction.

An example of converting $(0.1)_{10}$ to $(0.00011\cdots)_2 = (0.0\overline{0011})_2$ is as follows:

Multiplier	Base	Product	Integral	Fraction
0.1	2	0.2	0	.2
0.2	2	0.4	0	.4
0.4	2	0.8	0	.8
0.8	2	1.6	1	.6
0.6	2	1.2	1	.2
.	.	.	.	.

Real numbers are usually represented in the computer as **floating point** numbers often in binary system. There are several versions of floating-point numbers. We will give a simplified form as well as the IEEE (Institute of Electrical and Electronics Engineers) Standard.

2.3 Computer Representation of Numbers

A simplified form for representing a **floating-point number** x in **base** b is the following

Floating-point Number with t base-b digits precision

Let $f = \pm(.d_1 d_2 \cdots d_t)_b$ be the **mantissa** (or **significand**). Let $e = \pm(c_{s-1}c_{s-2}\cdots c_0)_b$ be the **exponent** (or **characteristic**). Then

$$x = (f)_b \times b^{(e)_b}$$

The floating-point number is **normalized** if $d_1 \neq 0$, or else $d_1 = d_2 = \cdots = d_n = 0$. **Significant** digits of a nonzero floating-point number are the digits following and including the first nonzero digits of the mantissa. All the digits of the mantissa of a normalized floating-point number are significant. As an example,

(not normalized) $0.01204 \times 10^1 = 0.12040 \times 10^0$ (normalized)

significant digits 1, 2, 0, 4, versus 1, 2, 0, 4, 0

The absolute value of the mantissa is always ≥ 0 and < 1. The exponent is also limited $E_{\min} \leq e \leq E_{\max}$. For a set of floating-point numbers in a simplified form, $-E_{\min} = E_{\max} = (aa\cdots a)_b$ where $a = b - 1$.

According to the above simplified form, the **overflow level (OFL)** is the largest floating-point number

$$N_{\max} = (.aa\cdots a)_b \times b^{(aa\cdots a)_b}$$

and the **underflow level (UFL)** is the smallest in absolute value nonzero floating-point number, given by

$$N_{\min} = (.100\cdots 0)_b \times b^{-(aa\cdots a)_b} \quad \text{if normalized, or}$$
$$N_{\min} = (.00\cdots 01)_b \times b^{-(aa\cdots a)_b} \quad \text{else}$$

Hence, if a number becomes smaller than UFL during a computation, it may be approximated as zero. UFL is a positive value that is very close to zero. If a number becomes larger than OFL during a computation, it may be approximated as infinity. OFL is a positive value that is very large.

$\mathbb{R}_b(t, s)$ denotes the set of all base b floating-point numbers that can be represented by t b-digits mantissa including the sign, and s b-digits exponent including the sign. Note that

$\mathbb{R}_b(t,s) \subset [-OFL,-UFL] \cup \{0\} \cup [UFL,OFL]$. **Overflow** occurs whenever a floating-point number greater than $N_{\max}$ or smaller than $-N_{\max}$ has to be stored in the computer. **Underflow** occurs when a nonzero floating-point number in the range $(-N_{\min}, N_{\min})$ has to be stored in the computer.

Note that $\mathbb{R}_b(t,s)$ is finite while $\mathbb{R}$ is finite. Also $\mathbb{R}$ is dense while $\mathbb{R}_b(t,s)$ is not, and the representable numbers are concentrated towards 0.

A real number $x = \pm(x_I \cdot x_F)_b = \pm(d_k d_{k-1} \cdots d_0 . d_{-1} \cdots)_b$ can be easily represented in a floating point form. First normalize the mantissa

$$x = (d_k \cdots d_0 . d_{-1} \cdots)_b = (.D_1 D_2 \cdots)_b \times b^{k+1}$$

Then there are three common ways to convert $x \in \mathbb{R}$ to a floating-point number $fl(x) \in \mathbb{R}_b(t,s)$.

(i) **Chopping**: Chop after digit t of the mantissa.

(ii) **Rounding (traditional)**: Chop after digit t, then round D_t up or down, depending on whether $D_{t+1} \geq b/2$ or $D_{t+1} < b/2$.

(iii) **Rounding (proper or perfect)**: Same as traditional, except when $D_{t+1} = b/2$ and $D_{t+2} = D_{t+3} = \cdots = 0$, then round D_t up or down to the nearest even.

An example is shown below:

x	chop	trad.round	proper round
.666...	.66	.67	.67
$-.305 \times 10^1$	$-.30 \times 10^1$	$-.31 \times 10^1$	$-.30 \times 10^1$
$-.315 \times 10^1$	$-.31 \times 10^1$	$-.32 \times 10^1$	$-.32 \times 10^1$
$-.3155 \times 10^1$	$-.31 \times 10^1$	$-.32 \times 10^1$	$-.32 \times 10^1$
$-.3055 \times 10^1$	$-.30 \times 10^1$	$-.31 \times 10^1$	$-.31 \times 10^1$

According to the IEEE standard, the form for representing floating-point numbers is slightly more sophisticated than the above simplified form, and is represented by the following,

IEEE Standard Floating-Point Numbers

Let $q \in \{0,1\}$, $d_i \in \{0,1\}$ for $i = 0, \cdots, t-1$. Let $p \in \{0,1\}$ and $e = (-1)^p \times (c_{s-2} \cdots c_1 c_0)_b$ with $c_i \in \{0,1\}$ for $i = 0, \cdots, s-2$.

$$(-1)^q \times (d_0 . d_1 d_2 \cdots d_{t-1})_b \times 2^e$$

Also $E_{\min} \leq e \leq E_{\max}$ with $E_{\min} = -E_{\max} + 1$.

There are numbers of **single precision** (binary32), **double precision** (binary64) and **quadruple precision** (binary128) and their characteristics are given by Table (1).

The mantissa is normalized: $d_0 = 1$ (or else the number is 0). Thus, d_0 need not be stored, and a t-bit mantissa needs $t-1$ storage bits. Note that the IEEE Standard uses proper rounding, and also includes some "special numbers", for example $+\infty$, $-\infty$, NaN (Not-a-Number) to handle indeterminate values that may arise in some computations.

Type of number	No. of bits	t	$E_{\min}$	$E_{\max}$	$\varepsilon_{\text{mach}}$
single precision, binary32	32	$24(=23+1)$	-126	$+127$	1.2×10^{-7}
double precision, binary64	64	$53(=52+1)$	-1022	$+1023$	2.2×10^{-16}
quadruple precision, binary128	128	$113(=112+1)$	-16382	$+16383$	1.9×1.9^{-34}

Table 1: Characteristics of numbers of different precisions

Type of number	t	$E_{\min}$	$E_{\max}$
single precision, binary32	7	-38	$+38$
double precision, binary64	16	-308	$+308$
quadruple precision, binary128	34	-4928	$+4928$

Table 2: Characteristics of numbers of different precisions

2.4 Data and Computational Error

$fl(x) - x$ is the **round-off error** (**representation error**). It is roughly proportional to x, so we can write $fl(x) - x = x\delta$, or $fl(x) = x(1+\delta)$, where δ is the **relative round-off error**. δ may depend on x but the bound, called the **unit round-off**, is independent of x and is often denoted by μ or u.

$|\delta| \leq b^{1-t}$ if normalized numbers and chopping are assumed.

$|\delta| \leq \frac{1}{2}b^{1-t}$ if normalized numbers and rounding are assumed.

$|\delta| = |\frac{x-\hat{x}}{x}|$ with x and its approximation $\hat{x}$ is said to be **correct in r significant b-digits**, if $|\delta| \leq \frac{1}{2}b^{1-r}$ $(= 5b^{-r}$ if $b = 10)$.

The (**absolute**) error in the approximation is $x - \hat{x}$, and the **relative error** is $\frac{x-\hat{x}}{x}$. Often we are interested only in the magnitude of the absolute and relative errors, in which case we take the absolute value.

Let $x, y \in \mathbb{R}_b(t, s)$, i.e. assume x, y are exactly representable on the computer system, and $\circ \in \{+, -, \times, /\}$. Then for the computer's floating point operation $\overline{\circ}$ corresponding to $\circ$. Then in general,

$$x \circ y \neq fl(x \,\overline{\circ}\, y)$$

Computer Arithmetic

The operation $\overline{\circ}$ may vary from machine to machine, but in general,

$$x \,\overline{\circ}\, y = fl(x \circ y)$$

which is achieved by the use of one or two extra temporary digits (bits). Thus
The error of a computation is the round-off error of the correct result.

In other words, computer operations return the correctly rounded result (i.e. the number closest to the correct answer in the representation being used.) As an example,

x	y	$x + y$	$x \,\overline{+}\, y = fl(x+y)$
$.2 \times 10^1$	$.51 \times 10^{-4}$	$.2000051 \times 10^1$	$.20 \times 10^1$
$.2 \times 10^1$	$.51 \times 10^{-1}$	$.2051 \times 10^1$	$.21 \times 10^1$

Note that $\mathbb{R}_b(t,s)$ is not closed with respect to any mathematical operations. Also, the phenomenon in which a nonzero number is added to another and the latter is left unchanged is often referred to as **saturation**.

Computation of Functions

The computer application $\overline{f}(x)$ of a built-in function f to a number $x \in \mathbb{R}_b(t,s)$ is constructed so that

$$\overline{f}(x) = fl(f(x))$$

That is,

The error of computing a function value is the round-off error of the correct result.

As an example, $\sin(1) = 0.84147098...$, $\overline{\sin}(1) = fl(\sin(1)) = fl(0.84147098...) = 0.84$.

The **machine epsilon** (**mach-eps**) $\varepsilon_{\text{mach}}$ is the smallest (non-normalized) floating-point number of the form b^{-i} for some $i \in \mathbb{Z}^+$ and the property $fl(1+\varepsilon_{\text{mach}}) > 1$.

$\varepsilon_{\text{mach}} = b^{1-t}$ if chopping is assumed.

$\varepsilon_{\text{mach}} = \frac{1}{2}b^{1-t}$ if traditional rounding is assumed.

$\frac{1}{2}b^{1-t} < \varepsilon_{\text{mach}} \leq b^{1-t}$ ($\varepsilon_{\text{mach}} = \frac{1}{2}b^{1-t} + b^{-t}$) if proper rounding is assumed.

Thus, $-\varepsilon_{\text{mach}} \leq \delta \leq \varepsilon_{\text{mach}}$. For IEEE Standard numbers, $\varepsilon_{\text{mach}} = 2^{1-t}$, which is that for single precision, $\varepsilon_{\text{mach}} = 2^{-23} \approx 1.19209 \times 10^{-7}$, for double precision, $\varepsilon_{\text{mach}} = 2^{-52} \approx 2.22045 \times 10^{-16}$, for quadruple precision, $\varepsilon_{\text{mach}} = 2^{-112} \approx 1.9259 \times 10^{-34}$. It is important to note that $\varepsilon_{\text{mach}}$ is different from UFL. We have the relation that $0 < \text{UFL} < \varepsilon_{\text{mach}} < \text{OFL}$.

As soon as an error, whether it is from round-off error or a floating-point operation, it may be amplified or reduced in subsequent operations. Let $x, y \in \mathbb{R}$. Then $fl(x) = x(1+\delta_1)$, $fl(y) = y(1+\delta_2)$. For multiplication, the computer computes

$$\begin{aligned} fl(x)\overline{\times}fl(y) &= fl(fl(x) \times fl(y)) \\ &= (x(1+\delta_1)y(1+\delta_2))(1+\delta_3) \\ &= xy(1+\delta_1+\delta_2+\delta_1\delta_2)(1+\delta_3) \\ &= xy(1+\delta_1+\delta_2+\delta_1\delta_2+\delta_3+\delta_1\delta_3+\delta_2\delta_3+\delta_1\delta_2\delta_3) \\ &\approx xy(1+\delta_1+\delta_2+\delta_3) \\ &= xy(1+\delta_\times) \end{aligned}$$

where $|\delta_\times| \leq 3\varepsilon_{\text{mach}}$

For addition, the computer computes

$$\begin{aligned} fl(x)\overline{+}fl(y) &= fl(fl(x)+fl(y)) \\ &= (x(1+\delta_1)+y(1+\delta_2))(1+\delta_3) \\ &= x(1+\delta_1)(1+\delta_3)+y(1+\delta_2)(1+\delta_3) \\ &\approx x(1+\delta_1+\delta_3)+y(1+\delta_2+\delta_3) \\ &= (x+y)(1+\frac{x}{x+y}(\delta_1+\delta_3)+\frac{y}{x+y}(\delta_2+\delta_3)) \\ &= (x+y)(1+\delta_+) \end{aligned}$$

where

$$\begin{aligned} |\delta_+| &\leq \left|\frac{x}{x+y}\right|\cdot 2\varepsilon_{\text{mach}} + \left|\frac{y}{x+y}\right|\cdot 2\varepsilon_{\text{mach}} = \frac{|x|+|y|}{|x+y|}\cdot 2\varepsilon_{\text{mach}} \\ &= 2\varepsilon_{\text{mach}} \qquad \text{when } xy>0 \\ &= 2\varepsilon_{\text{mach}}\frac{|x-y|}{|x+y|} \qquad \text{when } xy<0 \end{aligned}$$

Consider however, the case of $x \approx -y$, then the relative error blows up, known as the phenomenon of **catastrophic cancellation**. Adding nearly opposite (or subtracting nearly equal) numbers may result in having no correct digits at all.

As an example, consider the real numbers $x = 0.123456790 \times 10^0$ and $y = 0.123456789 \times 10^0$. The floating point approximation for x and y are in 8 decimal points with chopping. Then the difference $x - y$ is computed as

$$fl(fl(x) - fl(y)) = 0.00000001 \times 10^0 = 0.10000000 \times 10^{-7}$$

while

$$x - y = 0.000000001 \times 10^0 = 0.10000000 \times 10^{-8}$$

Although the relative error in the floating-point representation of y is at the level of 10^{-8}, the relative error in $fl(fl(x) - fl(y))$ is too high (at the level of 10^0).

$$\frac{0.1 \times 10^8 - 0.1 \times 10^{-7}}{0.1 \times 10^{-8}} = \frac{-0.9 \times 10^{-8}}{0.1 \times 10^{-8}} = -9 \times 10^0$$

Now consider some computation with input $x \in \mathbb{R}$ and output $f(x) \in \mathbb{R}$. Let $fl(x) = x(1+\delta_x)$, and assume $f(x)$ is twice differentiable in a neighbourhood of x. Then, if $f(x) \neq 0$, using Taylor's series and ignoring terms of second or higher order ($\mathcal{O}(\delta_x^2)$), we have

$$\begin{aligned} f(fl(x)) &\approx f(x) + f'(x)(fl(x) - f(x)) \\ &= f(x) + xf'(x)\left(\frac{fl(x)-f(x)}{x}\right) \\ &= f(x) + xf'(x)\delta_x \\ &= f(x)\left(1 + \frac{xf'(x)}{f(x)}\delta_x\right) \end{aligned}$$

so we have

$$f(fl(x)) \approx f(x)(1+\delta_{f(x)}) \qquad \text{with } \delta_{f(x)} = \frac{xf'(x)}{f(x)}\delta_x$$

If $|\delta_x| \leq \varepsilon_{\text{mach}}$, we have

$$|\delta_{f(x)}| \leq \left|\frac{xf'(x)}{f(x)}\right| \varepsilon_{\text{mach}}$$

The factor $\kappa_f = \left|\frac{xf'(x)}{f(x)}\right|$ is the **(relative) condition number** of $f(x)$ and is a measure of the relative sensitivity of the computation of $f(x)$ on relatively small changes in the input x, or in other words, a measure of how the relative error in x propagates in $f(x)$. A computation is **well-conditiond** if relatively small changes in the input, propduce relatively small changes in the output, otherwise it is **ill-conditioned**. Note that once $f(fl(x))$ is computed, it is stored and represented as $fl(f(fl(x))) = fl(f(x)(1+\delta_{f(x)})) = f(x)(1+\delta_{f(x)})(1+\delta) \approx f(x)(1+\delta_{f(x)}+\delta)$, where $|\delta| \leq \varepsilon_{\text{mach}}$.

Example 2.4.1 Let $f(x) = \sqrt{x}$. Then, the condition number of $f(x)$ is $\left|\frac{xf'(x)}{f(x)}\right| = \frac{1}{2}$, which is small and independent of x. Thus the computation of the square root of a number is a well-conditioned computation.

Example 2.4.2 Let $f(x) = e^x$. Then the condition number of $f(x)$ is $\left|\frac{xf'(x)}{f(x)}\right| = |x|$. The condition number depends linearly on x. For large $|x|$, we will have overflow ($e^{700} = 10^{304}$), thus we can say that the computation of the exponential well-conditioned for all acceptable values of x.

Note that if a function has $\kappa_f > \varepsilon_{\text{mach}}^{-1}$, we risk having no correct digits at all in the computed result $\overline{f(x)}$. The condition number of a function is a property inherent to the function itself, and not to the way the function is computed.

Stability is similar to conditioning, but it refers to a numerical algorithm, i.e. the particular way a certain computation is carried out. A numerical algorithm is **stable** if small changes in the algorithm input parameters have a small effect on the algorithm output, otherwise it is called **unstable**. Mathematically equivalent expressions are not necessarily computationally equivalent. Consider the following example.

Example 2.4.3 Consider the equation $(a-b)^2 = a^2 - 2ab + b^2$. Let $a = 15.6$, $b = 15.7$. The floating point representation is in 3 decimal digits with rounding.

$$(a-b)^2 = (-0.1)^2 = 0.01 = 0.1 \times 10^{-1}$$

$$a^2 - 2ab + b^2 = 243.36 - 2 \times 244.92 + 246.49 = 243 - 490 + 246 = 489 - 490 = -0.1 \times 10^1$$

In this case, computing the right hand side does not even get the sign correct. Then in this case, computing the left hand side is a more stable algorithm.

There is no general rule to pick the most stable algorithm for a certain computation. However, we should

- avoid adding nearly opposite (or subtracting nearly equal) numbers
- minimize the number of operations
- add numbers starting from the smallest and proceed to the largest in adding operations

- be alert when adding numbers of very different scales.

Let x be a number and let $y = f(x)$ be the desired result of some computation with input x. Assume the inverse function f^{-1} exists. Assume that instead of y, we computed $\hat{y}$ due to various errors such as the round-off error in x and propagation of error. The **forward error** is $y - \hat{y}$, and may include initial data error in x as well as propagation of error in computations. On the other hand, one can view $\hat{y}$ as being defined as $\hat{y} = f(\hat{x})$, then $\hat{x} = f^{-1}(\hat{y})$x, then $x - \hat{x}$ is the **backward error**. Essentially, $\hat{x}$ denotes an input value for which the exact function f would give the computed output $\hat{y}$. Alternatively, one can view $\hat{y}$ as being the result of inexact computations $\hat{f}$ on exact input x: $\hat{y} = \hat{f}(x)$. Then $\hat{f}(x) = f(\hat{x})$, and $\hat{x} = f^{-1}(\hat{f}(x))$.

Thus, the approximate solution $\hat{y}$ to the original problem is the exact solution to a modified problem ($\hat{x}$) or to the original problem with modified input ($f(\hat{x})$). For both forward and backward errors, the respective relative versions are

$$\text{relative forward error} = \frac{y - \hat{y}}{y} \qquad \text{relative backward error} = \frac{x - \hat{x}}{x}$$

The following relation hold:

$$|\text{relative forward error}| \;\approx\; \text{condition number} \;\times |\text{relative backward error}|$$

It is often easier to estimate the backward error rather than the forward one.

As an example, let $x = 2$ and $f(x) = \sqrt{x}$. Let $\hat{y} = 1.4$. Notice that $\sqrt{1.96} = 1.4$, thus $\hat{x} = 1.96$. Then
Forward error $= 1.4142... - 1.4 = 0.0142$
Relative forward error $= (1.4142... - 1.4)/1.4142... = 0.0142.../1.4142... \approx 0.01$
Backward error $= 2 - 1.96 = 0.04$
Relative backward error $= 2 - 1.96 = 0.04$
Condition number $= 1/2$

We have seen that computer representation of numbers may involve round-off error, which may be propagated through simple arithmetic operations. Also, certain mathematical expressions are approximated by other expressions that are possibly more convenient for calculation. The **truncation** or **discretization** error is the error in approximation of mathematical expressions with exact arithmetic. In addition, the evaluation of the approximate expressions is not performed in finite arithmetic instead of exact arithmetic. This gives rise to an additional error of **rounding error**. The **computational error** is the sum of truncation and rounding errors.

Total Error

The total error is given by

$$\text{Total error} \;=\; \text{propagated data error} \;+ [(\text{truncation error} \;+\; \text{rounding error})]$$

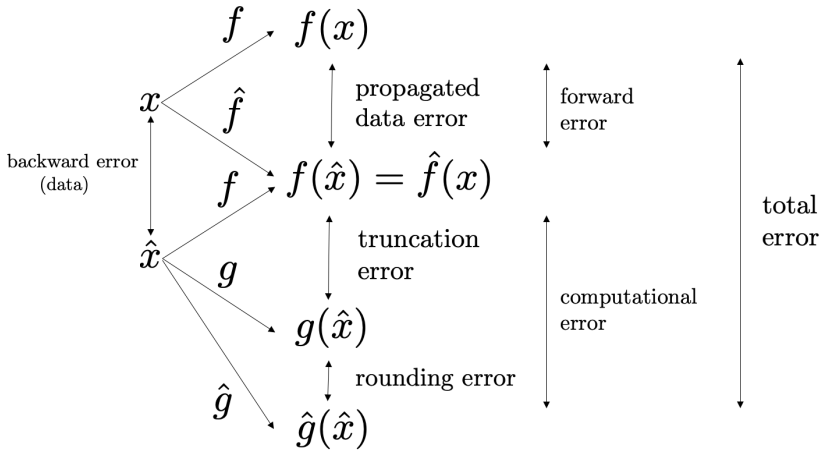
Often, one of these errors is dominant, but there is no general rule that tells us which.

Let x be the input data and $\hat{x}$ be its computer representation, and $f(x)$ be the target computation. Let $g(x)$ be the approximation of $f(x)$ and the computer evaluates $\hat{g}(x)$. Then the

total error is

$$\begin{aligned} f(x) - \hat{g}(x) &= (f(x) - f(\hat{x})) + [(f(\hat{x}) - g(\hat{x})) + (g(\hat{x}) - \hat{g}(\hat{x}))] \\ &= (\text{propagated data error}) + [\text{computational error}] \\ &= (\text{propagated data error}) + [(\text{truncation error}) + (\text{rounding error})] \end{aligned}$$

The propagated data error is dependent on the function itself (the condition number of f and the error in the data) but not the computational method. The computational error is dependent on the computational method.



Example 2.4.4 Assume we want to compute $\sin(\pi/8)$. Note that $\sin(\pi/8) = 0.38268343....$ Let $x = \pi$, $\hat{x} = 3$, $f(x) = \sin(x)$, $g(x) = x - x^3/3!$, and the computation be done in 3-decimal digit digit floating-point arithmetic, denoted by $\hat{g}(\hat{x})$. Then the result will be

$$\hat{g}(\hat{x}) = fl\left(fl\left(\frac{3}{8}\right) - fl\left[fl\left(\frac{1}{6}\right) fl\left(fl\left(\frac{3}{8}\right)^3\right)\right]\right) \approx 0.366$$

The initial data error is $x - \hat{x} = \pi - 3 \approx 0.1415926536$

The propagated data error is $f(x) - f(\hat{x}) = \sin(\pi/8) - \sin(3/8) \approx 0.0164109$

The truncation error is $f(\hat{x}) - g(\hat{x}) = \sin(3/8) - 3/8 + (3/8)^3/6 \approx 0.0000615915865$

The rounding error is approximately $g(\hat{x}) - \hat{g}(\hat{x}) = 3/8 - (3/8)^3/6 - \hat{g}(\hat{x}) = 0.3662109375 - 0.366 = 0.0002109375$

Total error is $f(x) - \hat{g}(\hat{x}) = 0.38268343... - 0.366 = 0.01668343...$

Recall Taylor's Theorem. Let $k \geq 1$ be an integer, and $a \in \mathbb{R}$. Assume $f : \mathbb{R} \to \mathbb{R}$ be $k+1$ times differentiable on the open interval and continuous on the closed interval between a and $x \in \mathbb{R}$. Then there exists ξ between x and a, such that

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k + \frac{f^{(k+1)}(\xi)}{(k+1)!}(x-a)^{(k+1)}$$

with the Taylor polynomial $t_k(x)$ of degree k

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k$$

and the remainder $R_{k+1}(x)$

$$R_{k+1}(x) = \frac{f^{(k+1)}(k+1)!}{(x-a)^{(k+1)}}$$

When x is close to a so that $|x-a|$ is small, the remainder is small, and the Taylor's polynomial is a good approximation to $f(x)$, thus

$$f(x) = t_k(x) + R_{k+1}(x) \approx t_k(x)$$

If f is infinitely differentiable. The infinite Taylor series of $f(x)$ is given by

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)(a)}}{k!}(x-a)^k$$

$f(x)$ is analytic if the infinite series converges to $f(x)$ for x close enough to a.

e^x	$1 + x + \frac{x^2}{2!} + \cdots$	$\sum_{k=0}^{\infty} \frac{x^k}{k!}$	any x
$\sin(x)$	$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots$	$\sum_{k=0}^{\infty}(-1)^k \frac{x^{2k+1}}{(2k+1)!}$	any x
$\cos(x)$	$1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots$	$\sum_{k=0}^{\infty}(-1)^k \frac{x^{2k}}{(2k)!}$	any x
$\log(x)$	$(x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \cdots$	$\sum_{k=1}^{\infty}(-1)^{k+1} \frac{(x-1)^k}{k}$	$0 < x \leq 2$
$\log(1+x)$	$x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots$	$\sum_{k=1}^{\infty}(-1)^{k+1} \frac{x^k}{k}$	$-1 \leq x \leq 1$

Note that the Taylor expansion of $\log(x)$ is around 1 ($a = 1$), while the others are around 0.

Recall the approximation of a function $f(x)$ by its Taylor polynomial $t_k(x)$

$$f(x) = t_k(x) + R_{k+1}(x) \approx t_k(x)$$

In this approximation, the truncation error is the remainder $R_{k+1}(x)$. When approximating a function $f(x)$ by its Taylor polynomial $t_k(x)$. To decide how many terms in the Taylor expansion we should use to keep the truncation error below a certain tolerance, we study mathematically for which k the remainder term in absolute value becomes bounded by the tolerance for all x and ξ of interest. To decide how many terms in the Taylor expansion we should use to keep the computational error below a certain tolerance, we keep adding more terms until the difference in the resulting approximation to the function is small enough (or until it makes no difference).

Lastly in this section, we will introduce the $\mathcal{O}(h^\alpha)$ and $\mathcal{O}(n^\beta)$ notation.

$\mathcal{O}(n^\beta)$

The $\mathcal{O}(n^\beta)$ is used to denote asymptotic complexity of algorithm in terms of problem size n. In this notation, n is assumed to be large and becoming larger and larger, tending to infinity.

$$\mathcal{O}(n^\beta) = c_0 + c_1 n + c_2 n^2 + \cdots + c_\beta n^\beta$$

where c_i are constants independent of n.

$\mathcal{O}(h^\alpha)$

The $\mathcal{O}(h^\alpha)$ is used to denote asymptotic behaviour of error of some discretization (computational) methods, in terms of the distance between neighbour discretization points, i.e. the "refinement" of the discretization. In this notation, h is assumed to be small and becoming smaller and smaller, tendning to 0.

$$\mathcal{O}(h^\alpha) = c_\alpha h^\alpha + c_{\alpha+1} h^{\alpha+1} + \cdots$$

where c_i are constants independent of h.

Note that $h = (b-a)/n$, that is, h is proportional to $1/n$.

$$\mathcal{O}(h^\alpha) = \mathcal{O}(n^{-\alpha}) = c_\alpha \frac{1}{n^\alpha} + c_{\alpha+1} \frac{1}{n^{\alpha+1}} + \cdots = \cdots + c_{\alpha+1} \frac{1}{n^{\alpha+1}} + c_\alpha \frac{1}{n^\alpha}$$

3 Solving Linear Systems

Solving linear systems lies in the heart of any scientific problem. Let A be a $n \times n$ matrix, and $b \in \mathbb{R}^n$. Solving $Ax = b$ means computing $x \in \mathbb{R}^n$ that satisfies $Ax = b$. We are interested in methods for computing x given A and b, and more particular in methods that compute x with high accuracy (small error) and high efficiency (low number of operations).

3.1 Gaussian Elimination

The most fundamental method for solving linear systems is **Gaussian elimination** (GE), which is based on the general technique of transforming a given linear system to another that is mathematically equivalent to the first (in the sense that the solution is the same), but easier to solve. Before this, we will look at the **Forward substitution algorithm for $n \times n$ lower-traingular matrices**.

3.1.1 Forward substitution algorithm for $n \times n$ lower-triangular matrices

Let A be an $n \times n$ lower triangular matrix with all diagonal elements nonzero. Let b be $n \times 1$ and we will solve $Ax = b$. Then the equations are

$$\begin{aligned}
a_{11}x_1 &= b_1 \\
a_{i1}x_1 + a_{i2}x_2 &= b_2 \\
a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{ii}x_i &= b_i \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{ni}x_i + \cdots + a_{nn}x_n &= b_n
\end{aligned}$$

Then starting with the first equation, we compute the first unknown, substitute in the second equation, compute the second unknown, substitute in the third equation, etc., proceeding forward

to the last equation. Thus we have

$$\begin{aligned} x_1 &= b_1/a_{11} \\ x_2 &= (b_2 - a_{21}x_1)/a_{22} \\ x_i &= (b_i - \sum_{j=1}^{i-1} a_{ij}x_j)/a_{ii} \\ x_n &= (b_n - \sum_{j=1}^{n-1} a_{nj}x_j)/a_{nn} \end{aligned}$$

Then we have Algorithm (3.1.1).

Algorithm 1 Forward Substitution Algorithm for $n \times n$ lower-triangular matrices

```
Initialize x as an array of size n
for i = 1 to n do
    x_i = b_i
    for j = 1 to i − 1 do
        x_i = x_i − a_ij x_j
    if a_ii ≠ 0 then
        x_i = x_i / a_ii
    else
        quit
```

The number of operations needed are $1 + 2 + \cdots + (n-1) = n(n-1)/2 \approx n^2/2$ pairs of additions and multiplications (flops) and n divisions. Since number of divisions are much smaller than that of flops, we often ignore them in the operation counts.

3.1.2 Backward substitution algorithm for $n \times n$ upper-triangular matrices

Now let us consider an $n \times n$ upper triangular matrix A, with all diagonal elements nonzero. Let b be $n \times 1$ and we will solve $Ax = b$. Then the equations are

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1i}x_i + \cdots + a_{1n}x_n &= b_1 \\ a_{22}x_2 + \cdots + a_{2i}x_i + \cdots + a_{2n}x_n &= b_2 \\ a_{ii}x_i + \cdots + a_{in}x_n &= b_i \\ a_{nn}x_n &= b_n \end{aligned}$$

Similar to the case of lower-triangular matrices, in this problem we start with the nth equation, then compute the last unknown, substitute in the equation before the last ($n-1$st), then proceed backwards to the first equation. The algorithm is the same as before, with the only difference being the order of traversing. Same as before, there are $n^2/2$ pairs of additions and multiplications (flops) and n divisions.

Algorithm 2 Backward Substitution Algorithm for $n \times n$ upper-triangular matrices

```
Initialize x as an array of size n
for i = n down to 1 do
    x_i = b_i
    for j = i + 1 to n do
        x_i = x_i − a_ij x_j
    if a_ii ≠ 0 then
        x_i = x_i / a_ii
    else
        quit
```

3.1.3 Gauss Elimination

Two linear systems are called **equivalent** when they admit exactly the same solutions or sets of solutions. That is $Ax = b \Leftrightarrow By = c$ when $x = y$. Given a linear system, there are infinitely many equivalent to it. One way to obtain systems equivalent to a given one is to apply the so-called row operations to the initial given system. There are three main row operations:

(i) Scalar multiplication of a row ρ_i with a nonzero scalar $\rho_i \leftarrow \kappa\rho_i$, $\kappa \neq 0$

(ii) Addition of rows $\rho_i \leftarrow \kappa\rho_i + \lambda\rho_j$, $\kappa \neq 0$, $\lambda \neq 0$

(iii) Permutation of rows $\rho_i \longleftrightarrow \rho_j$

Concatenating the column matrix of b to A, they form an augmented matrix $[A : b]$.

Example 3.1.3.1 As an example, let us consider the augmented matrix

$$[A : b] = \left[\begin{array}{cccc:c} 1 & -2 & 1 & 1 & 5 \\ 2 & -1 & 5 & -4 & -5 \\ -1 & 3 & -1 & 1 & 1 \\ -3 & 7 & -5 & 1 & -10 \end{array}\right] \tag{1}$$

To eliminate x_1 from rows 2 to 4, consider the row operations $\rho_2^{(1)} \leftarrow \rho_2 - 2\rho_1$, $\rho_3^{(1)} \leftarrow \rho_3 - (-\rho_1)$, $\rho_4^{(1)} \leftarrow \rho_4 - (-3\rho_1)$. These factors were chosen from that $2 = 2/1 = a_{21}/a_{11}$, $-1 = -1/1 = a_{31}/a_{11}$, $-3 = -3/1 = a_{41}/a_{11}$, so that the coefficients in positions a_{21}, a_{31}, a_{41} are annihilated. Element a_{11} which we divide with is called the **pivot** for step 1. Then we end up with

$$[A^{(1)} : b^{(1)}] = \left[\begin{array}{cccc:c} 1 & -2 & 1 & 1 & 5 \\ 0 & 3 & 3 & -6 & -15 \\ 0 & 1 & 0 & 2 & 6 \\ 0 & 1 & -2 & 4 & 5 \end{array}\right] \tag{2}$$

If we continue in a similar way for the rest of the rows, then we will end up with an upper traingular matrix. In high-level pseudocode, we have For the evolution of this algorithm, refer to Appendix A. Algorithm (3.1.3) overwrites the strictly lower triangular part of A by the strictly lower triangular part of L (the multipliers) and the upper triangular part of A with new numbers that can be considered to form an upper triangular matrix U.

In total there are about $\sum_{k=1}^{n-1}(n-k)^2 \approx \sum_{k=1}^{n-1} k^2 = n(n-1)(n-2)/6 \approx n^3/3$ flops, and $n^2/2$ divisions. If we consider also the right-hand side vector, then our algorithm becomes Algorithm (3.1.3).

Algorithm 3 Gauss Elimination (High Level)

```
for j = 1 to n − 1 do                                   ▷ For each row
    for i = j + 1 to n do                               ▷ For each column
        Eliminate x_j from row i
```

Algorithm 4 Gauss Elimination (without pivoting) algorithm for general $n \times n$ matrices

```
Given matrix A
for k = 1 to n − 1 do                                   ▷ For each A^(k)
    for i = k + 1 to n do                               ▷ For each row i below the pivot
        if a_kk ≠ 0 then ▷ Computer the multiplier if pivot is not 0 and store it in the corresponding element
            a_ik = a_ik/a_kk
        else
            quit
        for j = k + 1 to n do
            a_ij = a_ij − a_ik a_kj
```

Algorithm 5 Gauss Elimination (without pivoting) algorithm for general $n \times n$ matrices with simultaneous processing of the right-hand size vector

```
Given matrix A
for k = 1 to n − 1 do                                   ▷ For each A^(k)
    for i = k + 1 to n do                               ▷ For each row i below the pivot
        if a_kk ≠ 0 then ▷ Compute the multiplier if pivot is not 0 and store it in the corresponding element
            a_ik = a_ik/a_kk
        else
            quit
        for j = k + 1 to n do
            a_ij = a_ij − a_ik a_kj
        b_i = b_i − a_ik b_k
```

The additional work for processing the right hand side vector requires an extra $\sum_{k=1}^{n-1} \approx \sum_{k=1}^{n-1} k = n(n-1)/2 \approx n^2/2$ flops.

Hence the cost of solving a general linear system by Gauss elimination is the following

Gauss elimination: $n^3/3$ flops and $n^2/2$ divisions

Simultaneous processing of the right-hand size vector: $n^2/2$ flops

Back substitution: $n^2/2$ flops and n divisions

Total: $n^3/3 + 2 \times n^2/2$ flops and $n^2/2 + n$ divisions

3.2 LU Decomposition

During the Gauss elimination process, a new matrix, the upper triangular matrix U is stored in the upper triangular part of A. The multipliers $l_{ik}, i = k+1, \cdots, n, k = 1, \cdot n-1$ are generated,

and those can be stored in a strictly lower triangular matrix L or in the stricly lower triangular part of A.

If we extend L by setting 1's on the main diagonal, that is to produce a unit lower triangular matrix L, making L just lower triangular but not strictly, then we can show that

$$A = L \cdot U \tag{3}$$

Suppose we have the matrices

$$M^{(1)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -l_{21} & 1 & 0 & 0 \\ -l_{31} & 0 & 1 & 0 \\ -l_{41} & 0 & 0 & 1 \end{bmatrix} \qquad M^{(2)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -l_{32} & 1 & 0 \\ 0 & -l_{42} & 0 & 1 \end{bmatrix} \qquad M^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -l_{43} & 1 \end{bmatrix}$$

We will notice that $M^{(k)}A^{(k-1)} = A^{(k)}$. We have the following steps.

$$\begin{aligned} Ax &= b \\ M^{(1)}Ax &= M^{(1)}b \qquad \text{or } A^{(1)}x = b^{(1)} \\ M^{(2)}M^{(1)}Ax &= M^{(2)}M^{(1)}b \qquad \text{or } A^{(2)}x = b^{(2)} \\ M^{(3)}M^{(2)}M^{(1)}Ax &= M^{(3)}M^{(2)}M^{(1)}b \qquad \text{or } A^{(3)}x = b^{(3)} \text{ or } Ux = c \end{aligned}$$

Given the above, we have $M^{(3)}M^{(2)}M^{(1)}A = U$. Taking into account that $M^{(k)}, k = 1, 2, 3, ...$ are unit lower triangulars and therefore non-singular, we have $A = (M^{(1)})^{-1}(M^{(2)})^{-1}(M^{(3)})^{-1}U$ and $A = (M^{(3)}M^{(2)}M^{(1)})^{-1}U$.

Also knowing that

The inverse of a unit lower triangular matrix is a unit lower triangular matrix, and

The product of unit lower triangular matrices is a unit lower triangular matrix,

we have that $(M^{(1)})^{-1}(M^{(2)})^{-1}(M^{(3)})^{-1}$, equivalentlt M^{-1} is unit lower triangular. Let $L = (M^{(1)})^{-1}(M^{(2)})^{-1}(M^{(3)})^{-1}$, then $A = LU$. Hence L is the product of the inverses of the $M^{(k)}$ matrices. The matrices $M^{(k)}$ are called elementary Gauss transformations. The properties of triangular matrices are included in Appendix B. Then, it can be observed that an elementary Gauss transformation is a matrix with the following properties:

(i) It is unit lower triangular

(ii) Its only non-zero elements are the 1's on the diagonal, and possibly the elements of one column below the diagonal.

(iii) The inverse of an elementary Gauss transformation is a matrix like itself, with the signs of the non-zero off-diagonal elements reversed.

These can be generalized to $n \times n$ matrices. Let $L = (M^{(1)})^{-1} \cdots (M^{(n-1)})^{-1}$, then $A = LU$. For computational purposes, the matrices $M^{(k)}$ and their inverses are never stored individually.

Let $Ax = b$ and assume we apply Gauss elimination algorithm to A. Then we obtain the L and U factors of A such that $A = LU$. Then the solution of $Ax = b$ is reduced to the solutions of $Lc = b$ where $Ux = c$. Therefore, one forward and one backward substition are needed. The cost is listed as follows

LU factorization / GE: $n^3/3$ flops and $n^2/2$ divisions

Forward substitution for finding c: $n^2/2$ flops (The n divisions are not needed, since L has 1's on the main diagonal)

Backward substition for finding x: $n^2/2$ flops and n divisions

Total: $n^3/3 + 2 \times n^2/2$ flops and $n^2/2 + n$ divisions

Here are some properties of LU factorization. The L, U factors of the LU decomposition of a given matrix A are unique. That is, if $A = LU$ and $A = \tilde{L}\tilde{U}$ where $L, \tilde{L}$ are unit lower triangular and $U, \tilde{U}$ are upper triangular, then $L = \tilde{L}$ and $U = \tilde{U}$,

The LU decomposition can also be written in the form $A = LD\hat{U}$ where D is a diagonal matrix and $\hat{U}$ is a unit upper triangular matrix. More specifically, if $A = LU$, where L is unit lower triangular and U is upper triangular, then $A = LD\hat{U}$, where $d_{ii} = u_{ii}, i = 1, \cdots, n$ (i.e. $D = \text{ diag}(u_{11}, u_{22}, \cdots, u_{nn})$) and $\hat{u}_{ij} = u_{ij}/u_{ii}, i = 1, \cdots, n, j = 1, \cdots, n$.

The cost of LU factorization can be reduced for specific types of matrices such as symmetric matrices, symmetric positive definite matrices, and banded matrices.

3.2.1 Symmetric Matrices

Assume that A is symmetric. Each step if GE preserves symmetry of the submatrix $A(k + 1 \cdots n, k + 1 \cdots n)$, that is, step k of GE produces a symmetric $(n - k) \times (n - k)$ submatrix. This happens because the operation

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)}/a_{kk}^{(k-1)} \cdot a_{kj}^{(k-1)}$$

and the operation

$$a_{ji}^{(k)} = a_{ji}^{(k-1)} - a_{jk}^{(k-1)}/a_{kk}^{(k-1)} \cdot a_{ki}^{(k-1)}$$

end up to be the same, since A is symmetric. Thus, we can obtain the LU factorization of A by doing only half of the operations (either those corresponding to the upper triangular part, or those corresponding to the lower triangular part). This reduces the work of LU factorization of symmetruc matrices to $n^3/6$ flops. The LU factorization of a symmetric matrix then takes the form $A = LDL^\intercal$, i.e. $\hat{U} = L^\intercal$, $U = DL^\intercal$, where D is a diagonal matrix.

3.2.2 Symmetric Positive Definite Matrices

Assume A is symmetric positive definite, i.e. $A = A^\intercal$ and $x^\intercal Ax > 0$ for every non-zero vector x.

It can be shown that the elemenst of D of the factorization $A = LDL^\intercal$ are positive, i.e. $d_{ii} > 1, i = 1, \cdots, n$. The LU factorization of a symmetric positive definite matrix takes the form $A = CC^\intercal$ where $C = LD^{1/2}$ and $D^{1/2}$ is a matrix such that $D^{1/2} \cdot D^{1/2} = D$. (In this case, since D is diagonal, $D^{1/2}$ is also diagonal and we have $(D^{1/2})_{ii} = (d_{ii})^{1/2}$)

The factorization $A = CC^\intercal$ is called the **Choleski factorization** of A, and C is the **Choleski factor** of A. The Choleski algorithm is an algorithm based on GE, which computes the entries of the Choleski factor C of a symmetric positive definite matrix A. C is lower triangular, not unit lower triangular.

3.2.3 Banded Matrices

Recall that a square matrix A is banded with lower bandwith l and upper bandwidth u, i.e. (l, u)-banded, if $a_{ij} = 0$ when $i - j > l$ and $j - i > u$. In other words, in a (l, u)-banded matrix, all entries below the l-th subdiagonal and above the u-th super diagonal are 0. The total bandwidth is $l + u + 1$.

Each step of GE preserves bandedness of the matrix. That is, if A is (l, u)-banded, the L and U matrices arising from GE are $(l, 0)$ and $(0, u)$-banded respectively. Note that L is both unit lower triangular and $(l, 0)$-banded, and U is both upper triangular and $(0, u)$-banded.

Thus, we can obtain the LU factorization of A by doing only operations within the band of non-zero entries. This reduces the work of LU factorization of (l, u)-banded matrices to $l \cdot u \cdot n$ flops approximately.

Each step of GE processes a rectanglular array (submatrix) of size $(l+1) \times (u+1)$. In each of the last $l-1$ or $u-1$ (precisely max $\{l-1u-1\}$) steps the size of the submatrix decreases by 1, so that it does not go out of bounds. The algorithm requires $\sum_{k=1}^{n-1}(l+1)(u+1) \approx \sum_{k=1}^{n} lu = (n-1)lu \approx nlu$ flops.

Algorithm 6 LU Factorization by Gauss Elimination for (l, u)-banded Matrices

```
Given matrix A
for k = 1 to n − 1 do                                   ▷ For each A^(k)
    for i = k + 1 to min{k + l, n} do
        a_ik = a_ik / a_kk
        for j = k + 1 to min{k + u, n} do
            a_ij = a_ij − a_ik a_kj
```

Algorithm 7 Forward Substitution ($Ly = b$ where L is $(l, 0)$-banded) for Banded Matrices

```
for i = 1 to n do
    for j = max{i − l, 1} to i − 1 do
        b_i = b_i − l_ij b_j
    b_i = b_i / l_ii
```

Algorithm 8 Backward Substitution ($Ux = y$ where U is (o, u)-banded) for Banded Matrices

```
for i = n down to 1 do
    for j = i + 1 to min{i + u, n} do
        y_i = y_i − u_ij y_j
    y_i = y_i / u_ii
```

The forward substitution algorithm requires $\sum_{i=1}^{n}(l+1) = n(l+1) \approx nl$ flops. The backward substitution algorithm requries $\sum_{i=1}^{n}(u+1) = n(u+1) \approx nu$ flops. Thus the solution of an (l, u)-banded linear system by GE/LU and f/b/s requires $nlu + n(l + u)$ flops.

3.3 LU Factorization for Computing the Inverse of a Matrix

Recall that given a square matrix $A \in \mathbb{R}^{n\times n}$, if there exists a matirx $X \in \mathbb{R}^{n\times n}$ for which $A \cdot X = X \cdot A = I$, then X is the inverse of A and is denoted by A^{-1}.

Note that X is computed from the relation $A \cdot X = I$. Let X_j denote the j-th column of X, and $e_j = [0, 0, ..., 0, 1, 0, ..., 0]^\intercal$ be the unit vector with "1" in the jth row. Note that $X_j \in \mathbb{R}^{n\times 1}$ and $e_j \in \mathbb{R}^{n\times 1}$. The relation $A \cdot X = I$ then consists of the relations

$$A \cdot X_j = e_j \qquad\qquad j = 1, ..., n \tag{4}$$

For each j, relation $A \cdot X_j = e_j$ forms a linear system with matrix A, which is same for each j, and right-hand size e_j, which is different for each j.

Assume we have computed the LU factorization of A, and let L, U be the associated factors. Then the solution of the systems in Equation (4) reduces to the solution of the triangular systems

$$L \cdot Y_j = e_j, j = 1, ..., n \tag{5}$$

$$U \cdot X_j = Y_j, j = 1, ..., n \tag{6}$$

Algorithm 9 Algorithm for Computing the Inverse of a Matrix

Compute the L, U factors of the LU factorization of A by GE
for $j = 1, ..., n$ **do**
    Solve $L \cdot Y_j = e_j$ using f/s
    Solve $U \cdot X_j = Y_j$ using b/s
$A^{-1} = [X_1|X_2|\cdots|X_n]$

According to the cost for solving m linear systems each of size $n \times n$, with the same matrix, in this case, with $m = n$, the coset is $n^3/3 + n(n^2/2 + n^2/2) = n^3/3 + n^3 = 4n^3/3$ flops and $n^2/2 + n \cdot n = 3n^2/2$ divisions.

However, it can b shown that this cost can be reduced to n^3 flops and $3n^2/2$ divisions, if we take advantage of the particular form of the right-hand side vectors e_j. Thus, the cost of computing the inverse of a matrix is n^3 flops.

An important note is that the solution of $Ax = b$ can be obtained by $x = A^{-1}b$. However, the cost of this procedure is n^3 flops, which is 3 times as much as the cost of applying LU/GE and back and forward substitutions for one right-hand side vector. Therefore, inverses of matrices are not computed, unless they are explicitly needed.

The LU factorization of a symmetric matrix involves some symmetry of the factors: $A = LDL^\intercal$. The inverse of a symmetric matrix is a symmetric matrix. The LU factorization of a banded matrix involves some bandedness of the factors: If A is (l, u)-banded and no pivoting is used, then L is $(l, 0)$-banded and U is $(0, u)$-banded. The inverse of a banded matrix is in genral not a banded matrix.

3.4 Summary

We have seen two ways of solving a linear system $Ax = b$, both based on Gauss Elimination.

The first applies GE to A and b simultaneously, and obtains an upper triangular matrix U and a transformed vector $c = b^{(n-1)}$, such that $Ax = b$ is equivalent to $Ux = c$, then applies back substitution to $Ux = c$ to compute x. In this case, the multipliers are computed but do not need to be stored.

The second applies GE to A only and obtains L and U factors, thus $A = LU$, then applies forward substitution to $Lc = b$ to compute an intermediate vector c, and then applies backward substitution to $Ux = c$ to compute x. In this casem the multipliers are computed and stored in the strictly lower triangular part of A.

The two ways are mathematically equivalent and involve the same computational cost. However, when we need to solve several linear systems with the same matrix and different right-hand side vectors, we should adopt the second way, apply GE / LU once, then store the L and U factors, then apply a pair of forward substitution and backward substitution for each right-hand side vector. Then the cost of solving m linear systems of size $n \times n$ with the same matrix is $n^3/3 + m(n^2/2 + n^2/2) = n^3/3 + mn^2$ flops and $n^2/2 + mn$ divisions.

3.5 Gauss Elimination with Pivoting

Recall that a point in the Gauss Elimination algorithm, we have that if $a_{kk} \neq 0, a_{ik} = a_{ik}/a_{kk}$, else quit. Clearly, if $a_{kk} = 0$, this algorithm cannot be applied in the form it was given. When the denominator in the multiplier is small and we also have limited number of decimal digits floating-point arithmetic.

Pivoting in GE is a technique according to which rows (or columns or both rows and columns) are interchanged, so that zero or very small in absolute value denominators in multipliers are avoided. Thus the applicability or stability of GE is enhanced. Through GE witn pivoting, we either able to solve systems not zolvable due to zero denominators, or able to solve systems with better accuracy than without pivoting.

Row pivoting: Reorder rows of the matrix ($P_rA = LU$. P_r permutation matrix)

Column pivoting: Reorder columns of the matrix ($AP_c = LU$, P_c permutation matrix)

Partial pivoting: Row or column pivoting (one of the two)

Complete pivoting: Reorder both rows and columns of the matrix ($P_rAP_c = LU$)

Symmetric pivoting: Reorder both rows and columns of the matrix, but when rows k and s are interchanged, then columns k and s are also interchanged ($PAP^\intercal = LU$)

The most common form of pivoting is row pivoting, so we often omit the term "row" or "partial".

The strategy followed in (row) pivoting is summarized as follows:

At the kth GE, before the multiplier at column k, rows $k + 1, \cdots, n$, are computed, a search along the kth column from row k to row n is performed, to identify the largest in absolute value element. This element becomes the pivot. Assume the pivot belongs to row s, i.e. $|a_{sk}| = \max\{|a_{sk}|, i = k, \cdots, n\}$. If $s \neq k$, rows k and s are interchanged.

In most standard implementations, this interchange is done by indirect indexing. That is, an integer vector, say $ipiv$, of size n or $n - 1$ is used to refer to the indices of the rows. For example, we can define $ipiv$ (size $n - 1$) by using the following idea: $ipiv(k) = s$ means that rows k and s were interchanged during the kth elimination step. If $ipiv(k) = k$, then no interchange took place at the kth elimination step. (We could also keep reflecting the interchanges.) The result of one or more interchanges of rows of A is a reordering of the rows of A. After possible interchange of rows, the multipliers are computed as usual, and the elimination step proceeds.

Algorithm 10 Gauss Elimination with Partial Pivoting Algorithm for General $n \times n$ matrices

```
for k = 1 to n − 1 do
    Find row s with max_{i=k}^{n} {a_ik} (s = arg max_{i=k}^{n} {|a_ik|})
    if a_sk = 0 then
        Matrix is singular, quit
    Interchange rows k and s (all columns)
    for i = k + 1 to n do
        a_ik = a_ik / a_kk
        for j = k + 1 to n do
            a_ij = a_ij − a_ik a_kj
```

The algorithm requires $\sum_{k=1}^{n-1}(n-k) = \sum_{k=1}^{n-1} k = n(n-1)/2 \approx n^2/2$ comparisons in addition to the flops of the algorithm without pivoting. Asymptotically, it has the same cost as the no pivoting algorithm, i.e. $n^3/3$. An example of GE and LU factorization without pivoting is shown in Appendix C. The steps of GE with pivoting can be expressed as

$$\begin{aligned} Ax &= b \\ A^{(1)}x = M^{(1)}P_1Ax &= M^{(1)}P_1b = b^{(1)} \\ A^{(2)}x = M^{(2)}P_2M^{(1)}P_1Ax &= M^{(2)}P_2M^{(1)}P_1b = b^{(2)} \\ A^{(3)}x = M^{(3)}P_3M^{(2)}P_2M^{(1)}P_1Ax &= M^{(3)}P_3M^{(2)}P_2M^{(1)}P_1b = b^{(3)} \end{aligned}$$

Hence $M^{(3)}P_3M^{(2)}P_2M^{(1)}P_1A = U$. It can be shown that this relation is equivalent to $PA = LU$ where $P = P_3P_2P_1$, L is unit lower triangular and U is upper triangular. Note that $M^{(3)}P_3M^{(2)}P_2M^{(1)}P_1$ is not necessarily lower triangular. However, L^{-1} and $M^{(3)}P_3M^{(2)}P_2M^{(1)}P_1$ are the same only with some or all columns in different order. We can show that $L^{-1} = M^{(3)}P_3M^{(2)}P_2M^{(1)}P_1P^{-1}$.

Generalizing this to $n \times n$ matrices, at the kth step of GE with pivoting, we have

$$A^{(k)}x = M^{(k)}P_kM^{(k-1)}P_{k-1}\cdots M^{(1)}P_1Ax = M^{(k)}P_kM^{(k-1)}P_{k-1}\cdots M^{(1)}P_1b = b^{(k)} \tag{7}$$

From above, we have

$$A^{(k)}x = M^{(k)}P_kM^{(k-1)}P_{k-1}\cdots M^{(1)}P_1A = U \tag{8}$$

It can be shown that this is equivalent to

$$PA = LU, \text{ where } P = P_{n-1}P_{n-2}\cdots P_1 \tag{9}$$

and L is unit lower triangular. It can also be shown that

$$L = PP_1(M^{(1)})^{-1}P_2(M^{(2)})^{-1}\cdots P_{n-1}(M^{(n-1)})^{-1} \tag{10}$$

$$L^{-1} = M^{(n-1)}P_{n-1}M^{(n-2)}P_{n-2}\cdots M^{(1)}P_1P^{-1} \tag{11}$$

The matrices P_k are derived from I by interchanging two rows and are called **elementary permutation matrices**. As permutation matrices, they are also orthogonal: $P_k^{-1} = P_k^\intercal$. As elementary permutation matrices, they are also symmetric: $P_k = P_k^\intercal$.

Thus we have $P_k = P_k^{-1}, P_k P_k = I$, and so P_k are **idempotent** matrices. The matrix $P = P_{n-1}P_{n-2}\cdots P_1$ is a permutation matrix and therefore orthogonal. However, it si neither necessarily elementary permutation matrix nor necessarily symmetric.

There are two ways of solving $Ax = b$ on GE with pivoting (GEpiv). The first applies GEpiv to A and b simultaneously, and obtains an upper triangular matrix U and a transformed vector $c = b^{(n-1)}$, such that $Ax = b$ (or $[A : b]$) is equivalent to $Ux = c$ (or $[U : c]$), then applies back substitution to $Ux = c$ to compute x. In this case, the multipliers are computed, but do not need to be stored. Note that when GEpiv is applied to A and b, both the row permutations and the elimination operations are applied to both A and b. The permutation matrix P (or the *ipiv* vector) does not need to be stored for the solution process.

The second applies GEpiv to A, and obtains the L and U factors and the permutation matrix P such that $PA = LU$, then applies forward substitution to $Lc = Pb$ to compute an intermediate vector c, and then applies backward substitution to $Ux = c$ to compute x. In this case, the multipliers are computed and stored in the strictly lower triangular part of A. The permutation matrix P is not explicitly stored, but the vector *ipiv* is, and from that the relevant information can be extracted.

Similar to our discussion before, the two ways are mathematically equivalent and involve the same computational cost. However, when we need to solve linear systems with the same matrix and different right-hand side vectors, we should adopt the second way, apply GE/LU once, store the L and U factors and the *ipiv* vector, then apply row interchanges and a pair of f/s and b/s to each right-hand side vector. Then the cost for solving m linear systems of size $n \times n$ with the same matrix is $n^3/3 + m(n^2/2 + n^2/2) = n^3/3 + mn^2$ flops, $n^2/2 + mn$ divisions and $n^2/2$ comparisons.

3.5.1 Scaling and GE/LU with Partial Pivoting

Consider now using GE/LU with pivoting to solve $Ax = b$ with

$$\begin{bmatrix} -1 & 1000 \\ 1 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} 1000 \\ 2 \end{bmatrix}$$

It is clear that GEpiv applied to $Ax = b$ will not interchange the rows and GEpiv will produce the same results as no pivoting GE. This discrepancy comes from bad scaling. If we scale each equation so that the largest element in each row is equal to 1, and then apply GEpiv in three decimal digits floating-point arithmetic, we will get reasonably accurate results of GEpiv applied to $Ax = b$ with A and b given above. The operation requires $n(n-1) \approx n^2$ comparisons and equal

Algorithm 11 Gauss Elimination with Scaled Partial Pivoting Algorithm for General $n \times n$ Matrices

```
for i = 1 to n do
    t = max_{j=1}^{n} {|a_ij|}
    if t = 0 then
        The system is singular, quit
    for j = 1 to n do
        a_ij = a_ij / t
Apply GEpiv
```

number of divisions in addition to flops and comparisons required by GEpiv. There are variations

of this algorithm that save about half of the divisions by scaling only the multipliers as they are generated during the elimination process. The bottom-line is that it requires approximately the same amount of work as the no-pivoting algorithm ($n^3/3$). Thus scaled partial pivoting (though it does not always improve the results as magically as in the example), is considered a useful technique for improving the accuracy of GE.

3.5.2 Complete Pivoting

The complete pivoting strategy is that, at the kth GE step, before the multipliers at column k, rows $k+1, ..., n$ are computed, a search in the submatrix of size $(n-k+1) \times (n-k+1)$ is performed, to identify the largest in absolute value element. This element becomes the pivot. Assume the pivot belongs to row l and column m, i.e. $|a_{lm}| = \max\{|a_{ij}|, i = l, ..., n, j = k, ..., n\}$. If $l \neq k$, rows k and l are interchanged, and if $m \neq k$, columns k and m are interchanged.

Algorithm 12 Gauss Elimination with Complete Pivoting Algorithm for General $n \times n$ Matrices

```
for k = 1 to n − 1 do
    (l, m) = arg max_{i=k,j=k}^{n} {|a_ij|}
    if a_lm = 0 then
        The system is singular, quit
    Interchange rows k and l, columns k and m
    for i = k + 1 to n do
        a_ik = a_ik / a_kk
        for j = k + 1 to n do
            a_ij = a_ij − a_ik a_kj
```

The algorithm requires $\sum_{k=1}^{n-1}(n-k)^2 = \sum_{k=1}^{n-1} k^2 = n(n-1)(2n-1)/6 \approx n^3/3 = \mathcal{O}(n^3)$ comparisons in addition to the flops required by the no-pivoting algorithm. Asymptotically, it requires approximately twice the amount of work of the no-pivoting algorithm ($n^3/3$). For this reason, although complete pivoting can improve the accuracy of GE on certain pathological cases futher than scaled partial pivoting, it's rarely used.

3.5.3 Effect of Pivoting to Special Matrices

For symmetric matrices, row (or column or complete) pivoting may destroy the symmetry of a matrix. Symmetric pivoting (same reordering to both rows and columns) preserves symmetry.

For banded matrices ((l,u)-banded), partial (row or column) pivoting may alther the bandwidth, but preserves *some* bandedness, more specifically, row pivoting applied to an (l,u)-banded matric generates (at most) l additional non-zero superdiagonals, i.e. U is $(0, u+l)$-banded, while L has at most $l+1$ non-zero elements per column. Column pivoting applied to an (l,u)-banded matrix generates (at most) u additional non-zero subdiagonals, i.e. L is $(u+l, 0)$-banded. Complete pivoting may destroy any bandedness.

A Gauss Elimination algorithms

Algorithm 13 Gauss Elimination (Detailed Version)

Initialize $A^{(0)} = A$
for $k = 1$ to $n - 1$ **do** ▷ For each $A^{(k)}$
  for $i = k + 1$ to n **do** ▷ For each row i below the pivot
    if $a_{kk}^{(k-1)} \neq 0$ **then** ▷ Calculate the multiplier if the pivot is not 0
      $t = a_{ik}^{(k-1)} / a_{kk}^{(k-1)}$
    else
      quit
    for $j = 1$ to n **do** ▷ For each column j in row i of the new matrix
      $a_{ij}^{k} = a_{ij}^{(k-1)} - t \cdot a_{kj}^{(k-1)}$ ▷ Compute the new element a_{ij} of $A^{(k)}$ (Row operation)

Algorithm 14 Gauss Elimination (Saving Memory)

Given matrix A
for $k = 1$ to $n - 1$ **do**
  for $i = k + 1$ to n **do**
    if $a_{kk} \neq 0$ **then**
      $t = a_{ik} / a_{kk}$
    else
      quit
    for $j = 1$ tp n **do**
      $a_{ij} = a_{ij} - t \cdot a_{kj}$

Algorithm 15 Gauss Elimination (Saving Time)

Given matrix A
for $k = 1$ to $n - 1$ **do** ▷ For each $A^{(k)}$
  for $i = k + 1$ to n **do** ▷ For each row i below the pivot
    if $a_{kk} \neq 0$ **then** ▷ Calculate the multiplier if the pivot is not 0
      $t = a_{ik} / a_{kk}$
    else
      quit
    for $j = k + 1$ tp n **do** ▷ No need to compute the zeros, update a_{ij}
      $a_{ij} = a_{ij} - t \cdot a_{kj}$

B Properties of Triangular Matrices

(i) The product of lower (upper) triangular matrices is a lower (upper) triangular matrix.

(ii) The product of unit lower (upper) triangular matrices is a unit lower (upper) triangular matrix.

(iii) The inverse of a non-singular lower (upper) triangular matrix is a lower (upper) triangular matrix

(iv) The inverse of a unit lower (upper) triangular matrix is a unit lower (upper) triangular matrix.

C Gauss Elimination (GE) and LU factorization with Pivoting Example

Consider the linear system $Ax = b$, where

$$A = \begin{bmatrix} 1 & -2 & -4 & -3 \\ 2 & 0 & -1 & 2 \\ -1 & 2 & 2 & 2 \\ 3 & 0 & -3 & 6 \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}, b = \begin{bmatrix} 2 \\ -1 \\ 4 \\ 9 \end{bmatrix}$$

The system can also be described with the augmented matrix

$$[A : b] = \begin{bmatrix} 1 & -2 & -4 & -3 & : & 2 \\ 2 & 0 & -1 & 2 & : & -1 \\ -1 & 2 & 2 & -1 & : & 4 \\ 3 & 0 & -3 & 6 & : & 9 \end{bmatrix}$$

Let perm_vec $p = [1, 2, 3, 4]$.

Recall that relation $ipiv(k) = s$ denotes that rows k and s were interchanged in the kth step of the algorithm. The part of A below the stair step belongs to L, but we overlay the elements of L within $A^{(k)}$ for being concise and for indicating that we save memory when doing the related computation.

For $k = 1$, Find along column 1 (rows 1 to 4) the maximum in absolute value element, and interchange its row with row 1.

$$\begin{bmatrix} 1 & -2 & -4 & -3 & : & 2 \\ 2 & 0 & -1 & 2 & : & -1 \\ -1 & 2 & 2 & -1 & : & 4 \\ 3 & 0 & -3 & 6 & : & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 0 & -3 & 6 & : & 9 \\ 2 & 0 & -1 & 2 & : & -1 \\ -1 & 2 & 2 & -1 & : & 4 \\ 1 & -2 & -4 & -3 & : & 2 \end{bmatrix} \qquad P_1 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$ipiv = [4, \cdot, \cdot] (\text{perm\_vec } p = [4, 2, 3, 1])$$

$$\rho_2^{(1)} \leftarrow \rho_2 - 2/3\rho_1$$
$$\rho_3^{(1)} \leftarrow \rho_3 + 1/3\rho_1$$
$$\rho_4^{(1)} \leftarrow \rho_4 - 1/3\rho_1$$

$$[A^{(1)} : b^{(1)}] = \begin{bmatrix} 3 & 0 & -3 & 6 & : & 9 \\ 2/3 & 0 & 1 & -2 & : & -7 \\ -1/3 & 2 & 1 & 1 & : & 7 \\ 1/3 & -2 & -3 & -5 & : & -1 \end{bmatrix}$$

For $k = 2$, Find along column 2 (rows 2 to 4) the maximum in absolute value element, and interchange its row with row 2.

$$\left[\begin{array}{cccc:c} 3 & 0 & -3 & 6 & 9 \\ 2/3 & 0 & 1 & -2 & -7 \\ -1/3 & 2 & 1 & 1 & 7 \\ 1/3 & -2 & -3 & -5 & -1 \end{array}\right] \rightarrow \left[\begin{array}{cccc:c} 3 & 0 & -3 & 6 & 9 \\ -1/3 & 2 & 1 & 1 & 7 \\ 2/3 & 0 & 1 & -2 & -7 \\ 1/3 & -2 & -3 & -5 & -1 \end{array}\right] \qquad P_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$ipiv = [4, 3, \cdot](\text{perm\_vec } p = [4, 3, 2, 1])$$

$$\rho_3^{(2)} \leftarrow \rho_3^{(1)} - 0\rho_2^{(1)}$$
$$\rho_4^{(2)} \leftarrow \rho_4^{(1)} + 2/2\rho_2^{(1)}$$

$$[A^{(2)} : b^{(2)}] = \left[\begin{array}{cccc:c} 3 & 0 & -3 & 6 & 9 \\ -1/3 & 2 & 1 & 1 & 7 \\ 2/3 & 0 & 1 & -2 & -7 \\ 1/3 & -1 & -2 & -4 & 6 \end{array}\right]$$

For $k = 3$, Find along column 3 (rows 3 to 4) the maximum in absolute value element, and interchange its row with row 3.

$$\left[\begin{array}{cccc:c} 3 & 0 & -3 & 6 & 9 \\ -1/3 & 2 & 1 & 1 & 7 \\ 2/3 & 0 & 1 & -2 & -7 \\ 1/3 & -1 & -2 & -4 & 6 \end{array}\right] \rightarrow \left[\begin{array}{cccc:c} 3 & 0 & -3 & 6 & 9 \\ -1/3 & 2 & 1 & 1 & 7 \\ 1/3 & -1 & -2 & -4 & 6 \\ 2/3 & 0 & 1 & -2 & -7 \end{array}\right] \qquad P_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$ipiv = [4, 3, 4](\text{perm\_vec } p = [4, 3, 1, 2])$$

$$\rho_4^{(3)} \leftarrow \rho_4^{(2)} + 1/2\rho_3^{(2)}$$

$$[A^{(3)} : b^{(3)}] = \left[\begin{array}{cccc:c} 3 & 0 & -3 & 6 & 9 \\ -1/3 & 2 & 1 & 1 & 7 \\ 1/3 & -1 & -2 & -4 & 6 \\ 2/3 & 0 & -1/2 & -4 & -4 \end{array}\right]$$

Then we have

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1/3 & 1 & 0 & 0 \\ 1/3 & -1 & 1 & 0 \\ 2/3 & 0 & -1/2 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 3 & 0 & -3 & 6 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & -2 & -4 \\ 0 & 0 & 0 & -4 \end{bmatrix}$$

The relation $A = LU$ no longer holds but we have $PA = LU$, where $P = P_3P_2P_1$.

D Computer Arithmetic, Gauss Elimination (GE) and LU factorization Abridged

D.1 Computer Arithmetic

$\mathbb{R}_b(t,s)$ represents the floating-point numbers of the form $f \times b^e$ where f the mantissa has t bits including the sign (If the mantissa is normalized, 1 does not need to be stored and is not included), and e the exponent has s bits including the sign. The distribution of numbers is denser towards 0 while the distribution in the same exponent is uniform.

$\varepsilon_{\text{mach}}$ is the smallest non-normalized floating-point number of the form b^{-i} for i some positive integer, with the property $fl(1+\varepsilon_{\text{mach}}) > 1$. To find $\varepsilon_{\text{mach}}$ of a floating-point system with t base b digits precision, the floating point representation of 1 with base b mantissa is $1 = 0.\underbrace{100\cdots0}_{t \text{ digits}} \times b^1$. Then construct $\varepsilon_{\text{mach}}$ with the smallest possible mantissa and the smallest possible exponent so that $1+\varepsilon_{\text{mach}} > 1$. Based on the type of rounding used, $\varepsilon_{\text{mach}}$ is b^{-t}, b^{-t-1}, or b^{-t+1}. Add the exact value of these guesses to 1 and after rounding, check if the sum is greater than 1. If so, the corresponding guess is $\varepsilon_{\text{mach}}$. In general, $\varepsilon_{\text{mach}} = b^{1-t}$ if chopping is assumed, and $\varepsilon_{\text{mach}} = (1/2)b^{1-t}$ if traditional rounding is assumed. $(1/2)b^{1-t} < \varepsilon_{\text{mach}} \leq b^{1-t} (\varepsilon_{\text{mach}} = (1/2)b^{1-t} + b^{-t})$ if proper rounding is assumed.

For a computer system that uses decimal floating-point arithmetic with t digits mantissa (plus one digit for the mantissa sign) and s digit exponent (plus one digit for the exponent sign), overflow and underflow levels are computed as follows.

$$N_{\max} = 0.(\underbrace{aa\cdots a}_{t \text{ digits}})_b \times b^{\overbrace{(aa\cdots a)}^{s \text{ digits}}{}_b} \tag{12}$$

$$N_{\max} = 0.(\underbrace{10\cdots00}_{t \text{ digits}})_b \times b^{-\overbrace{(aa\cdots a)}^{s \text{ digits}}{}_b} \tag{13}$$

where $a = b-1$. We have that $\mathbb{R}_b(t,s) \subset [-OFL, -UFL] \cup \{0\} \cup [UFL, OFL]$.

A common problem in computation is catastrophic cancellation. Adding nearly opposite numbers may result in having no correct digits at all. To avoid this, the conjugate of square roots can be multiplied to the original expression, or the original expression can be rewritten with the rules of trigonometry, including the following

$$\sin(a \pm b) = \sin(a)\cos(b) \pm \cos(a)\sin(b)$$
$$\cos(a \pm b) = \cos(a)\cos(b) \mp \sin(a)\sin b$$
$$\sin^2(a) = \frac{1-\cos(2a)}{a}$$
$$\cos^2(a) = \frac{1+\cos(2a)}{a}$$

The factor $\kappa_f = \left|\frac{xf'(x)}{f(x)}\right|$ is the condition number of f and is a measure of the relative sensitivity of the computation of $f(x)$ on relatively small changes in x. Check cases where the numerator is large and when the denominator is small. L'Hopital's rule is sometimes useful when checking limits.

D.2 GE/LU and Operation Counts

The general procedure for different kinds of computations

Forward substitution	$x_i = \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j\right)/a_{ii}$
Backward substitution	$x_i = \left(b_i - \sum_{j=i+1}^{n} a_{ij}x_j\right)/a_{ii}$
GE / LU	To compute $A^{(k+1)}$ from $A^{(k)}$, $\rho_i^{(k+1)} \leftarrow \rho_i^{(k)} - (a_{ik}^{(k)}/a_{kk}^{(k)})\rho_{k+1}^{(k)}$ where $i \geq k+2$. Then $A = LU$ and hence $Lc = b$ and $Ux = c$
GE with partial (row) pivoting	To compute $A^{(k)}$, find along column k the largest element in absolute value and exchange that row with row k. Update $ipiv$, where $ipiv(k) = s$ denotes that row s is exchanged with row k at the kth step. Also create P_k which is the identity matrix but with rows s and k exchanged. Then apply GE. $PA = LU$ where $P = P_{n-1}\cdots P_1$. Solve $PAx = Pb$ by $Ux = c$ and $Pc = Lb$.
GE with scaled partial (row) pivoting	For each row i, find the maximum element in absolute value $\lvert a_{\max,\,i}\rvert$. Divide each element in row i by $\lvert a_{\max,\,i}\rvert$. Also create $D = \mathrm{diag}\{1/\lvert a_{\max,\,i}\rvert\}$. Then apply GEpiv. $PDA = LU$ where $P = P_{n-1}\cdots P_1$. Solve $PDAx = PDb$ by $Lc = PDb$ and $Ux = c$
Obtaining Choleski factor C of symmetric matrix A	Apply GE to A and obtain its L and U factors. $D = \mathrm{diag}\{U_{ii}\}$. Let $E = D^{1/2}$ then $C = LE$ such that $A = CC^{\intercal}$. C is the Choleski factor of A.

Table 3: Operation counts of different type of computations

The operation counts of different type of computations are summarized in the following table

Standard matrix-matrix multiplication $C = AB$ where $A \in \mathbb{R}^{l\times m}$ and $B \in \mathbb{R}^{m\times n}$	lnm flops
Matrix inverse	n^3 flops and $3n^2/2$ divisions
Forward substitution	$n^2/2$ flops and n divisions
Backward substitution	$n^2/2$ flops and n divisions
Solving a general linear system by GE	GE: $n^3/3$ flops and $n^2/2$ divisions + simultaneous processing of RHS vector: $n^2/2$ + b/s = $n^3/3 + n^2$ flops and $n^2/2 + n$ divisions
Solving a general linear system by LU	GE: $n^3/3$ flops and $n^2/2$ divisions + f/s + b/s = $n^3/3 + n^2$ flops and $n^2/2 + n$ divisons
GEpiv	$n^2/2$ comparisons + GE

Table 4: Operation counts of different type of computations