1.2.3 Pivoting Techniques in Gaussian Elimination

Let us consider again the 1st row operation in Gaussian Elimination, where we start with the original augmented matrix of the system

$$(A, \underline{b}) = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1N} & b_1 \\ a_{21} & a_{22} & a_{23} & \dots & a_{2N} & b_2 \\ a_{31} & a_{32} & a_{33} & \dots & a_{3N} & b_3 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \\ a_{N1} & a_{N2} & a_{N3} & \dots & a_{NN} & b_N \end{bmatrix} \qquad \textbf{(1.2.3-1)}$$

and perform the following row operation,

$$(A^{(2,1)}, \underline{b}^{(2,1)}) = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1N} & b_1 \\ (a_{21} - \lambda_{21} a_{11}) & (a_{22} - \lambda_{21} a_{12}) & (a_{23} - \lambda_{21} a_{13}) & \dots & (a_{2N} - \lambda_{21} a_{1N}) & (b_2 - \lambda_{21} b_1) \\ a_{31} & a_{32} & a_{33} & \dots & a_{3N} & b_3 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & \dots & a_{NN} & b_N \end{bmatrix}$$

(1.2.3-2)

To place a zero at the (2,1) position as desired, we want to define λ_{21} as

$$\lambda_{21} = \frac{a_{21}}{a_{11}} \qquad \textbf{(1.2.3-3)}$$

but what happens if $a_{11} = 0$? => λ_{21} blows up to $\pm\infty$!

The technique of <u>partial pivoting</u> is designed to avoid such problems and make Gaussian Elimination a more robust method.

Let us first examine the elements of the 1st column of A,

$$A(:,1) = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ \vdots \\ \vdots \\ a_{N1} \end{bmatrix} \qquad \textbf{(1.2.3-4)}$$

Let us search all the elements of this column to find the row #j that contains the value with the largest magnitude, i.e.

$$|a_{ji}| \geq |a_{k1}| \text{ for all } k = 1, 2, ..., N \qquad \textbf{(1.2.3-5)}$$

or

$$|a_{j1}| = \max_{k\in[1,N]}\{|a_{k1}|\} \qquad \textbf{(1.2.3-6)}$$

Since the order of the equations does not matter, we are perfectly free to exchange rows # 1 and j to form the system

$$(\bar{A}, \bar{b}) = \begin{bmatrix} a_{j1} & a_{j2} & a_{j3} & ... & a_{jN} & b_j \\ a_{21} & a_{22} & a_{23} & ... & a_{2N} & b_2 \\ a_{11} & a_{12} & a_{13} & ... & a_{1N} & b1 \\ : & : & : & : & : & \\ a_{N1} & a_{N2} & a_{N3} & ... & a_{NN} & b_N \end{bmatrix} \begin{matrix} \\ \\ \leftarrow \text{row \# j} \\ \\ \\ \end{matrix} \qquad \text{row \# 1}$$

(1.2.3-7)

Now, as long as any of the elements of the 1st column of A are non-zero, a_{j1} is non-zero and we are safe to begin eliminating the values below the diagonal in the 1st column.

If all the elements of the 1st column are zero, we immediately see that no equation in the system makes reference to the unknown x, and so there is no unique solution. We therefore stop the elimination process at this point and "give up".

The row-swapping procedure outlined in **(1.2.3-1), (1.2.3-6), (1.2.3-7)** is known as a partial pivoting operation.

For every new column in a Gaussian Elimination process, we 1st perform a partial pivot to ensure a non-zero value in the diagonal element before zeroing the values below.

The Gaussian Elimination algorithm, modified to include partial pivoting, is

For i= 1, 2, …, N-1 % iterate over columns

- select row $j \geq i$ such that $|a_{ji}| = \max_{j \geq i} \{|a_{ii}|, |a_{i+1,i}|, \ldots, |a_{N,i}|\}$
- if $a_{ji} = 0$, no unique solution exists, STOP
- if $j \neq i$, interchange rows i and j

For j = i+1, i+2, …, N % rows in column i below diagonal

$$> \lambda \leftarrow \frac{a_{ji}}{a_{ii}}$$

For k = i, i+1, …, N % elements in row j from left → right

$> a_{jk} \leftarrow a_{jk} - \lambda a_{ik}$

end

$> b_j \leftarrow b_j - \lambda b_i$

end

end

Backward substitution then proceeds, in the same manner as before.

Even if rows must be swapped at each column, computational overhead of partial pivoting is low, and gain in robustness is large!

To demonstrate how Gaussian Elimination with partial pivoting is performed, let us consider the system of equations with the augmented matrix

$$(A, \underline{b}) = \begin{bmatrix} 1 & 1 & 1 & 4 \\ 2 & 1 & 3 & 7 \\ 3 & 1 & 6 & 2 \end{bmatrix} \text{pivot} \qquad \textbf{(1.2.3-8)}$$

First, we examine the elements in the 1st column to see that the element of largest magnitude is found in row #3.
We therefore perform a partial pivot to interchange rows 1 and 3.

$$(\overline{A}, \overline{b}) = \begin{bmatrix} 3 & 1 & 6 & 2 \\ 2 & 1 & 3 & 7 \\ 1 & 1 & 1 & 4 \end{bmatrix} \qquad \textbf{(1.2.3-9)}$$

We now perform a row operation to zero the (2,1) element

$$\lambda_{21} = \frac{\overline{a}_{21}}{\overline{a}_{11}} = \frac{2}{3} \qquad \textbf{(1.2.3-10)}$$

$$(A^{(2,1)}, \underline{b}^{(2,1)}) = \begin{bmatrix} 3 & 1 & 6 & 2 \\ 2-\left(\frac{2}{3}\right)3 & 1-\left(\frac{2}{3}\right)1 & 3-\left(\frac{2}{3}\right)6 & 7-\left(\frac{2}{3}\right)2 \\ 1 & 1 & 1 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 1 & 6 & 2 \\ 0 & \frac{1}{3} & 1 & 5\frac{2}{3} \\ 1 & 1 & 1 & 4 \end{bmatrix} \qquad \textbf{(1.2.3-11)}$$

We now perform another row operation to zero the (3,1) element

$$\lambda_{31} = \frac{a_{31}^{(2,1)}}{a_{11}^{(2,1)}} = \frac{1}{3} \qquad \textbf{(1.2.3-12)}$$

$$(A^{(3,1)}, \underline{b}^{(3,1)}) = \begin{bmatrix} 3 & 1 & 6 & 2 \\ 0 & \frac{1}{3} & -1 & 5\frac{2}{3} \\ 1-\left(\frac{1}{3}\right)3 & 1-\left(\frac{1}{3}\right)1 & 1-\left(\frac{1}{3}\right)6 & 4-\left(\frac{1}{3}\right)2 \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 1 & 6 & 2 \\ 0 & \frac{1}{3} & -1 & 5\frac{2}{3} \\ 0 & \frac{2}{3} & -1 & 3\frac{1}{3} \end{bmatrix} \quad \textbf{(1.2.3-13)}$$

We now move to the 2nd column, and note that the element of largest magnitude appears in the 3rd row. We therefore perform a partial pivot to swap rows 2 and 3.

$$(A^{-(3,1)}, \underline{b}^{-(3,1)}) = \begin{bmatrix} 3 & 1 & 6 & 2 \\ 0 & \frac{2}{3} & -1 & 3\frac{1}{3} \\ 0 & \frac{1}{3} & -1 & 5\frac{2}{3} \end{bmatrix} \quad \textbf{(1.2.3-14)}$$

We now perform a row operation to zero the (3,2) element.

$$\lambda_{32} = \frac{\frac{1}{3}}{\frac{2}{3}} = \frac{1}{2} \quad \textbf{(1.2.3-15)}$$

$$(A^{(3,2)}, \underline{b}^{(3,2)}) = \begin{bmatrix} 3 & 1 & 6 & 2 \\ 0 & \frac{2}{3} & -1 & 3\frac{1}{3} \\ 0 & \frac{1}{3}-\frac{1}{2}\left(\frac{2}{3}\right) & -1-\frac{1}{2}(-1) & 5\frac{2}{3}-\frac{1}{2}\left(3\frac{1}{3}\right) \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 1 & 6 & 2 \\ 0 & \frac{2}{3} & -1 & 3\frac{1}{3} \\ 0 & 0 & -\frac{1}{2} & 4 \end{bmatrix} \quad \textbf{(1.2.3-16)}$$

After the elimination method, we have an upper triangular form that is easy to solve by backward substitution.

We write out the system of equations,

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

(1.2.3-17)

First, we find

$$x_3 = -8$$

(1.2.3-18)

Then, from the 2nd equation,

$$x_2 = \frac{(3\frac{1}{3} + x_3)}{\frac{2}{3}} = -7$$

(1.2.3-19)

And finally from the 1st equation

$$x_1 = \frac{(2 - 6x_3 - x_2)}{3} = 19$$

(1.2.3-20)

The solution is therefore

$$x = \begin{bmatrix} 19 \\ -7 \\ -8 \end{bmatrix}$$

(1.2.3-21)

Note that in our partial pivoting algorithm, we swap rows to make sure that the largest magnitude element in each column at and below the diagonal is found in the diagonal position. We do this even if the diagonal element is non-zero.

This may seem like wasted effort, but there is a very good reason to do so. It reduces the "round-off error" in the final answer. To see why, we must consider briefly how numbers are stored in a computer's memory.

If we were to look at the memory in a computer, we would find data represented digitally as a sequence of 0'a and 1's

```
00100101   00101011 , …..
----------   -------------
byte # i       byte # i+1
```

To store a real number in memory, we need to represent it in such format. This is done using floating point notation.

Let f be a real number that we want to store in memory. We do so by representing it as some value

$$\tilde{f} \approx f$$

That we write as

$$\tilde{f} = \pm[d_1 * 2^{e-1} + d_2 * 2^{e-2} + ...d_t * 2^{e-t}]$$

t= machine precision

(1.2.3-22)

Each

$d_i = 0$ or 1

And so is represented by one bit in memory. e is an integer exponent in the range

$$L \le e \le U$$

L ≡ underflow limit

U = overflow limit

(1.2.3-23)

e is also stored as a binary number, for example if we allocate a byte (8 bits) to storing e, then

$$\frac{\pm}{e_7}\frac{0}{e_6}\frac{0}{e_5}\frac{0}{e_4}\frac{0}{e_3}\frac{0}{e_2}\frac{1}{e_1}\frac{1}{e_0} = 3$$

(1.2.3-24)&(1.2.3-25)

The largest e is when

$$e = e_0 * 2^0 + e_1 * 2^1 + e_2 * 2^2 + e_3 * 2^3 + e_4 * 2^4 + e_5 * 2^5 + e_6 * 2^6$$

For which

$$e_{max} = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = \sum_{k=0}^{6} 2^k = 2^7 - 1$$

(1.2.3-26)

So, say in general

$$e_{max} \approx 2^k$$

Where k depends on number of bits allocated to store e. **(1.2.3-27)**

For the largest magnitude variable that can be stored in memory, M

$$M = \underset{\pm d_1=1}{\overset{0}{}} \frac{1}{d_2}\frac{1}{d_3}\frac{1}{d_4}\frac{1}{d_5}\frac{1}{d_6}\frac{1}{d_7}\frac{1}{d_8} \qquad \frac{0}{\pm e_6}\frac{1}{e_5}\frac{1}{e_4}\frac{1}{e_3}\frac{1}{e_2}\frac{1}{e_1} \qquad \textbf{(1.2.3-34)}$$

$$M = +\left[2^{e-1} + 2^{e-2} + 2^{e-3} + 2^{e-4} + 2^{e-5} + 2^{e-6} + 2^{e-7} + 2^{e-8}\right] \qquad \textbf{(1.2.3-35)}$$

where

$$e = +2^6 = +64 \qquad \textbf{(1.2.3-36)}$$

so

$$M = \sum_{k=1}^{8} 2^{e-k} = 1.8375\text{x}10^{19} \qquad \textbf{(1.2.3-37)}$$

In general, for machine precision t and $= 2^k$,

$$M = \sum_{i=1}^{t} 2^{U-i} = 2^U \sum_{i=1}^{t} 2^{-i} = 2^U \sum_{i=1}^{t} \left(\frac{1}{2}\right)^i = 2^U \left(\frac{1}{2}\right) \sum_{i=1}^{t} \left(\frac{1}{2}\right)^{i-1} \qquad \textbf{(1.2.3-38)}$$

We now use the identity for a geometric progression,

$$\sum_{i=1}^{N} x^{i-1} = \frac{x^N - 1}{x - 1}, x \neq 1 \qquad \textbf{(1.2.3-39)}$$

to write

$$M = 2^U \left(\frac{1}{2}\right) \frac{\left(\frac{1}{2}\right)^t - 1}{\left(\frac{1}{2} - 1\right)} = 2^U \left[1 - 2^{-t}\right] \qquad \textbf{(1.2.3-40)}$$

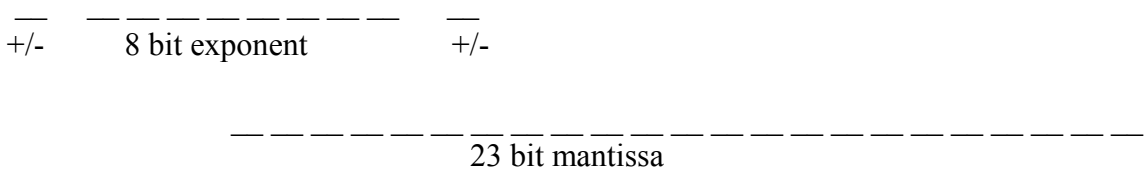
We see that, for a given t and k (i.e. how much memory we wish to allocate to storing each number), there is a maximum and minimum magnitude to the real numbers that can be represented.

$$m \leq \left|\tilde{f}\right| \leq M$$

For t = 8, for various k, $U = 2^k$, we have the following m and M,

k	$U=2^k=-L$	$m = 2^{L-1}$	$M = 2^U(1-2^{-t})$
4	16	$\sim 7.36\text{x}10^{-6}$	$\sim 6.53\text{x}10^{4}$
6	64	$\sim 2.71\text{x}10^{-20}$	$\sim 1.84\text{x}10^{19}$
8	256	$\sim 4.32\text{x}10^{-78}$	$\sim 1.15\text{x}10^{77}$

The typical representation on a 32-bit machine is



total = 32 bits for each real number

The important point to note is that when we wish to store a real number f in memory, in general f cannot be exactly represented by a finite set of bits in floating point notation; certainly this is true for $\frac{1}{3}, \pi, e$. Instead, we represent it with the closest possible value

$$\tilde{f} = \pm\left[d_1 * 2^{e-1} + d_2 * 2^{e-2} + ...d_t * 2^{e-t}\right] \quad \textbf{(1.2.3-22, repeated)}$$

so that the difference between the "true" value of f and the represented value $\tilde{f}$ is called the round-off error, rd(f)

$$rd(f) = f - \tilde{f} \quad \textbf{(1.2.3-41)}$$

For binary representation of a number f with $m \le \left|f\right| \le M$, from **(1.2.3-22)**, we see that the magnitude of the round-off error is

$$rd(f) \sim 2^{e-t} = 2^{-t} * 2^{e} = (eps) * 2^{e} \quad \textbf{(1.2.3-42)}$$

where we define the machine precision as

(eps) = 2^{-t}, see MATLAB command "eps" **(1.2.3-43)**

Let us write $rd(f) = r_f(eps)x2^{e}{}_f$ **(1.2.3-44)**

Where

r_f is some number of O(1) (i.e. is on the order of 1)

e_f = exponent of f

We write $\tilde{f} = m_f x2^{e_f}$, where m_f = mantissa of f, also O(1) **(1.2.3-45)**

Then,

$$\frac{rd(f)}{\tilde{f}} = \frac{r_f}{m_f}(eps)x\frac{2^{e_f}}{2^{e_f}} = \frac{r_f}{m_f}(eps) \qquad \textbf{(1.2.3-46)}$$

For $eps << 1, rd(f) << \tilde{f}$ **(1.2.3-47)**

So, when we initially assign a value in memory, the round-off error may be small. We want to make sure that this initial small error, as it propagates through our algorithms, does not "blow up" to become large.

For example, let us take the difference of two close, large numbers

$$f = 3.000\ 0001x10^6$$
$$g = 3.000\ 00009x10^6$$
(1.2.3-48)

$$f\text{-}g = 0.01 \text{ so } |f - g| << |f|, |g| \qquad \textbf{(1.2.3-49)}$$

If $f = \tilde{f} + rd(f),\ \ g = \tilde{g} + rd(g)$ **(1.2.3-50)**

$$f - g = \tilde{f} - \tilde{g} + [rd(f) - rd(g)] \qquad \textbf{(1.2.3-51)}$$

so

$$rd(f\text{-}g) = rd(f) - rd(g) \qquad \textbf{(1.2.3-52)}$$

Let us write

$$rd(f) = r_f(eps)x2^{e_f}, \qquad rd(g) = r_g(eps)x2^{e_g} \qquad \textbf{(1.2.3-53)}$$

$$\tilde{f} = m_f x2^{e}\ , \quad \tilde{g} = m_g x2^{e_g} \qquad \textbf{(1.2.3-54)}$$

then

$$\frac{rd(\tilde{f}-\tilde{g})}{\tilde{f}-\tilde{g}} = \frac{r_f(eps)2^{e_f} - r_g(eps)2^{e_g}}{m_f 2^{e_f} - m_g 2^{e_g}} \qquad \textbf{(1.2.3-55)}$$

Let us now take the case of numbers like

$$f = 3.000\ 0001x10^6$$

$$g = 3.000\ 00009x10^6 \quad \textbf{(1.2.3-48, repeated)}$$

for which, in binary or decimal notation, $e_f = e_g$ and $m_f - m_g << 1$

Then

$$\frac{rd(\tilde{f}-\tilde{g})}{\tilde{f}-\tilde{g}} = \frac{(r_f - r_g)(eps)}{m_f - m_g} \qquad \textbf{(1.2.3-56)}$$

as $(r_f - r_g) = O(1)$ $m_f - m_g << 1$, we see that compared to

$$\frac{rd(f)}{\tilde{f}} = \frac{r_f}{m_f}(eps) \qquad \textbf{(1.2.3-46, repeated)}$$

$$\left|\frac{rd(\tilde{f}-\tilde{g})}{\tilde{f}-\tilde{g}}\right| >> \left|\frac{rd(f)}{\tilde{f}}\right|, \left|\frac{rd(g)}{\tilde{g}}\right| \qquad \textbf{(1.2.3-57)}$$

Taking the difference between two large, similar numbers therefore is bad, from the view of propagation of error, since the accumulated round-off error in the result is much larger than it should be from a direct assignment.

We wish to design, and operate, our algorithms so that the accumulated round-off errors do not grow larger, and ideally decay to zero. If error "blows up", the errors become larger in magnitude than the values that we are trying to represent, and we get instability that crashes the program.

For example, let us say that we wish to perform the operation

$$a \leftarrow a - \lambda b \qquad \textbf{(1.2.3-58)}$$

We really perform the operation on their floating point representations

$$\tilde{a} \leftarrow \tilde{a} - \tilde{\lambda}\tilde{b} \qquad \textbf{(1.2.3-59)}$$

Since $rd(a) = a - \tilde{a}$, we subtract these equations

$$rd(a) \leftarrow rd(a) - \lambda b + \tilde{\lambda}\tilde{b} + e_{new} \qquad \textbf{(1.2.3-60)}$$

If $\tilde{\lambda} = \lambda$, we can write

$$\left|rd(a)\right| \leftarrow \left|rd(a) - \lambda rd(b) + e_{new}\right| \qquad \textbf{(1.2.3-61)}$$

If $|\lambda| > 1$, any round-off error in b is magnified during this operation, but if $|\lambda| < 1$, then error accumulated to date by b is decreased as it is "passed" to the new value of a.

In Gaussian elimination, we perform a number of operations

$$a_{jk} \leftarrow a_{jk} - \lambda a_{ik}, \lambda = \frac{a_{ji}}{a_{ii}} \qquad \textbf{(1.2.3-62)}$$

By performing partial pivoting, we ensure $|a_{ii}| > |a_{ji}|$, so $|\lambda| < 1$ and the algorithm has favorable error propagation characteristics.

We can further enhance this favorable property of error propagation by performing complete, or full, pivoting.

In complete pivoting, one searches for the maximum magnitude element not only in the current column, but in others as well. The pivoting involves not merely interchange of rows, but also of columns. This makes the book keeping more complex as column interchange implies an interchange of the values of the unknowns in their position in the solution vector $\underline{x}$. While full pivoting improves the accuracy of calculation, by more rapidly decaying the round-off error, it is not strictly necessary for systems that are well-balanced, i.e. all elements along any given row $a_{i1}, a_{i2}\ldots,a_{iN}$ are all of the same order of magnitude. We therefore do not discuss this technique further.

We now note that with the addition of partial pivoting, Gaussian elimination provides a robust method of solving linear equations that is easily implemented by a computer. It either returns a solution to the linear system, or, if no non-zero pivot element if found, it recognizes that there is no unique solution and STOP's.

We therefore have a dependable method that can be used in higher-level algorithms to solve non-linear algebraic equations, partial differential equations, etc. First, we must examine in closer detail the existence and uniqueness of solutions.