MATH 612
Computational methods for equation solving and function minimization – Week # 6

F.J.S.

Spring 2014 – University of Delaware

Plan for this week

- Discuss any problems you couldn't solve from previous lectures
- We will cover Lectures 17, part of 16 and move on to 20
- The second coding assignment is due Friday. Radio silence has been decreed for this assignment: you can only discuss it with your coding partner.

Remember that...

... I'll keep on updating, correcting, and modifying the slides until the end of each week.

MORE ON BACKWARD STABILITY

Review

Arithmetic

For every real number

$$\mathrm{fl}(x) = x(1+\epsilon), \qquad |\epsilon| \le \epsilon_{\text{machine}}.$$

For every pair of floating point numbers and $* \in \{+, -, \times, /\}$,

$$x \circledast y = (x * y)(1+\epsilon), \qquad |\epsilon| \le \epsilon_{\text{machine}}$$

Backward stability

An algorithm $\widetilde{f} : X \to Y$ to approximate a problem $f : X \to Y$ is backward stable, when for all $x \in X$, there exists $\widetilde{x} \in X$ such that

$$f(\widetilde{x}) = \widetilde{f}(x), \qquad \|\widetilde{x} - x\| \le C\epsilon_{\text{machine}}\|x\|.$$

The dot product

We want to show that the dot product in floating point arithmetic is backward stable. Note that algorithms have to be defined in a fully deterministic way. For instance, given two vectors $(x_1, \dots, x_m), (y_1, \dots, y_m)$, our problem is $f : X \to \mathbb{R}$ (where $X = \mathbb{R}^n \times \mathbb{R}^n$) given by

$$f((x, y)) = f(x, y) = \sum_j x_j y_j .$$

The algorithm is, for instance

$$\begin{aligned} \widetilde{f}(x, y) \;\; &= \;\; \Bigg(\dots \Big(((\mathrm{fl}(x_1) \otimes \mathrm{fl}(y_1)) \oplus (\mathrm{fl}(x_2) \otimes \mathrm{fl}(y_2))) \\ &\qquad \oplus (\mathrm{fl}(x_3) \otimes \mathrm{fl}(y_3)) \Big) \dots \oplus (\mathrm{fl}(x_m) \otimes \mathrm{fl}(y_m)) \Bigg) \end{aligned}$$

Vectors with two components

Parturbations arise in floating point representations and arithmetic operations

$$\begin{aligned}
\widetilde{f}(x,y) &= (\mathrm{fl}(x_1) \otimes \mathrm{fl}(y_1)) \oplus (\mathrm{fl}(x_2) \otimes \mathrm{fl}(y_2)) \\
&= \Big(\underbrace{x_1(1+\epsilon_1)}_{\text{due to fl}(x_1)} \; y_1(1+\epsilon_2) \; \underbrace{(1+\epsilon_3)}_{\text{due to } \times} \\
&\qquad + x_2(1+\epsilon_4) y_2(1+\epsilon_5)(1+\epsilon_6)\Big) \; \underbrace{(1+\epsilon_7)}_{\text{due to } +} \\
&= \underbrace{x_1(1+\epsilon_1)(1+\epsilon_7)}_{\widetilde{x}_1} \underbrace{y_1(1+\epsilon_2)(1+\epsilon_3)}_{\widetilde{y}_1} \\
&\quad + \underbrace{x_2(1+\epsilon_4)(1+\epsilon_7)}_{\widetilde{x}_2} \underbrace{y_2(1+\epsilon_5)(1+\epsilon_6)}_{\widetilde{y}_2} \\
&= f(\widetilde{x}, \widetilde{y})
\end{aligned}$$

Vectors with two components (cont'd)

Using that $|\epsilon_j| \leq \epsilon_{\text{machine}}$, we cand bound

$$|(1+\epsilon_i)(1+\epsilon_j) - 1| \leq 2\epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2).$$

so

$$\begin{aligned} |\widetilde{x}_i - x_i| &\leq |x_i|(2\epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2)) \\ |\widetilde{y}_i - y_i| &\leq |y_i|(2\epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2)) \end{aligned}$$

We can wrap up the bounds with the formula

$$\|(\widetilde{x}, \widetilde{y}) - (x, y)\|_\infty \leq \|(x, y)\|_\infty (2\epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2))$$

Let's see with three-vectors (11 different ϵ)

$$
\begin{aligned}
\widetilde{f}(x,y) &= \Big((\mathrm{fl}(x_1) \otimes \mathrm{fl}(y_1)) \oplus (\mathrm{fl}(x_2) \otimes \mathrm{fl}(y_2)) \Big) \oplus (\mathrm{fl}(x_3) \otimes \mathrm{fl}(y_3)) \\
&= \Bigg(\Big(x_1(1+\epsilon_1) y_1 (1+\epsilon_2)(1+\epsilon_3) \\
&\qquad + x_2(1+\epsilon_4) y_2(1+\epsilon_5)(1+\epsilon_6) \Big) (1+\epsilon_7) \\
&\qquad\qquad + x_3(1+\epsilon_8) y_3(1+\epsilon_9)(1+\epsilon_{10}) \Bigg) (1+\epsilon_{11}) \\
&= \underbrace{x_1(1+\epsilon_1)(1+\epsilon_7)}_{\widetilde{x}_1} \underbrace{y_1(1+\epsilon_2)(1+\epsilon_3)(1+\epsilon_{11})}_{\widetilde{y}_1} \\
&\quad + \dots
\end{aligned}
$$

and, collecting carefully,

$$
\begin{aligned}
|\widetilde{x}_i - x_i| &\le |x_i|(2\epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}})) \\
|\widetilde{y}_i - y_i| &\le |y_i|(3\epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}}))
\end{aligned}
$$

UPPER TRIANGULAR SYSTEMS

Back substitution

Solve the equations backwards

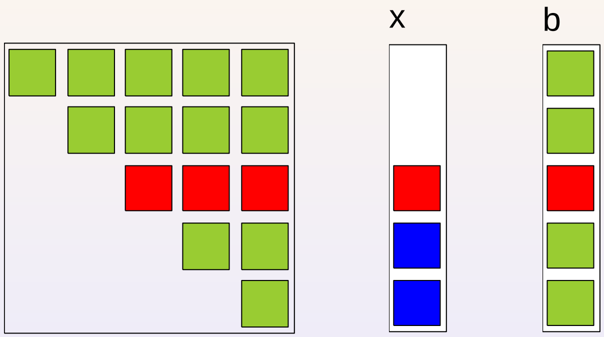
$$\sum_{j=i}^{m} r_{i,j} x_j = b_i, \qquad i = m, m-1, \ldots, 1$$

by computing

$$x_i = \Big(b_i - \sum_{j=i+1}^{m} r_{i,j} x_j\Big) / r_{i,i}, \qquad i = m, m-1, \ldots, 1.$$

```
x=zeros(m,1);
for i=m:-1:1
    x(i)=(b(i)-r(i,i+1:end)*x(i+1:end))/r(i,i);
                                    % row times column
end
```

With a picture



In **red** what is being used when $i = 3$. In **blue** the elements of x that have already been computed, *but they are being used in this step.* Entries in **green** are inactive.

Recursive back substitution

In the previous algorithm, we work equation by equation. Now, we modify the right-hand-side in each step and reduce the size of the system by one. The idea is that once x_i is computed, its contribution to all the preceding equations is subtracted.

Initial steps: $b_i^{(0)} = b_i$

$$\begin{aligned} & x_m = b_m^{(0)}/r_{m,m} \\ & b_i^{(1)} = b_i^{(0)} - r_{i,m}x_m \qquad i = 1, \ldots, m-1, \\ & x_{m-1} = b_m^{(1)}/r_{m-1,m-1} \\ & b_i^{(2)} = b_i^{(1)} - r_{i,m-1}x_{m-1} \qquad i = 1, \ldots, m-2, \\ & x_{m-2} = \ldots \end{aligned}$$

In this algorithm we access the upper triangular matrix by columns. Back substitution uses it by rows.

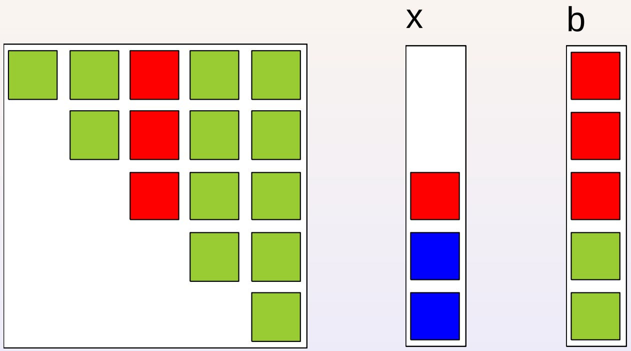
Recursive back substitution (2)

```
for i=m:-1:1
    x(i)=b(i)/r(i,i);
    b(1:i-1)=b(1:i-1)-r(1:i-1,i)*x(i);
end
```

The flop count for both algorithms is $\sim m^2$.

Remark. Forward susbtitution for a lower triangular matrix is back substitution for a system where we count equations and unknowns in the reverse order. Therefore, everything you show for back susbtitution is automatically shown for forward substitution.

With a picture



In **red** what is being used when $i = 3$. In **blue** the elements of x that have already been computed. Entries in **green** are inactive.

BKWD STABILITY OF BACK SUBS

In what sense?

We have an invertible upper triangular system

$$Rx = b, \qquad x = R^{-1}b$$

and solve it using back susbtitution and floating point operations. The operator is

$$f(R) = x = R^{-1}b,$$

that is we consider the matrix to be the input and we fix b. The algorithm is the result of solving with back susbtitution and floating point operations. *For simplicity*, we will assume that the entries of R and b are already floating point numbers, so we will only care about how the floating point operations in back susbtitution affect the result.

In what sense? (2)

Backward stability means: given R we can find another upper triangular matrix $\widetilde{R}$ such that

$$\widetilde{R}^{-1}b = f(\widetilde{R}) = \widetilde{f}(R) = \widetilde{x},$$

where $\widetilde{x}$ is the computed solution, and

$$\|\widetilde{R} - R\| \leq C\epsilon_{\text{machine}}\|R\|.$$

In other words, we can find $\widetilde{R}$ such that

$$\widetilde{R}\widetilde{x} = b = Rx \qquad \|\widetilde{R} - R\| \leq C\epsilon_{\text{machine}}\|R\|,$$

or also

$$(R + \delta R)\widetilde{x} = b, \qquad \|\delta R\| \leq C\epsilon_{\text{machine}}\|R\|.$$

The 2×2 case

$$\begin{array}{rl} \text{(Exact)} & \text{(Computed)} \\ x_2 = b_2/r_{22} & \tilde{x}_2 = b_2 \oslash r_{22} \\ x_1 = (b_1 - r_{12}x_2)/r_{11} & \tilde{x}_1 = (b_1 \ominus (r_{12} \otimes \tilde{x}_2)) \oslash r_{11} \end{array}$$

Two tricks to handle ϵ...

- if $|\epsilon| \le \epsilon_{\text{machine}}$, then

$$1 + \epsilon = \frac{1}{1+\epsilon'} \qquad |\epsilon'| \le \epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2),$$

- if $|\epsilon_1|, |\epsilon_2| \le \epsilon_{\text{machine}}$, then

$$(1+\epsilon_1)(1+\epsilon_2) = 1 + \epsilon_3, \qquad |\epsilon_3| \le 2\epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2).$$

The 2×2 case (2)

$$\begin{array}{rl} \text{(Exact)} & \text{(Computed)} \\ x_2 = b_2/r_{22} & \widetilde{x}_2 = b_2 \oslash r_{22} \\ x_1 = (b_1 - r_{12}x_2)/r_{11} & \widetilde{x}_1 = (b_1 \ominus (r_{12} \otimes \widetilde{x}_2)) \oslash r_{11} \end{array}$$

$$\widetilde{x}_2 = \frac{b_2}{r_{22}}(1+\epsilon_1) = \frac{b_2}{r_{22}(1+\epsilon_1')} = \frac{b_2}{\widetilde{r}_{22}}$$

$$\widetilde{x}_1 = \frac{(b_1 - r_{12}\widetilde{x}_2(1+\epsilon_2))(1+\epsilon_3)}{r_{11}}(1+\epsilon_4)$$

$$= \frac{b_1 - r_{12}(1+\epsilon_2)\widetilde{x}_2}{r_{11}(1+\epsilon_3')(1+\epsilon_4')} = \frac{b_1 - \widetilde{r}_{12}\widetilde{x}_2}{\widetilde{r}_{11}}$$

This is basically it. We have $\widetilde{R}\widetilde{x} = b$ and

$$|\widetilde{r}_{ij} - r_{ij}| = |r_{ij}|(c\epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}})), \qquad c = 1 \text{ or } 2.$$

The general result

Given an $m \times m$ invertible upper triangular matrix R and the system $Rx = b$, let $\widetilde{x}$ be the solution computed using back substitution with floating point operations (on a computer satisfying the usual hypotheses). Then there exists an upper triangular matrix $\widetilde{R}$ such that

$$\widetilde{R}\widetilde{x} = b, \qquad |\widetilde{r}_{ij} - r_{ij}| = |r_{ij}|(m\epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2)).$$

In particular back substitution is backward stable.

The general result, made more general

What we showed (only for the 2×2 case, but extendable for larger cases), does not involve the floating point representation of right-hand-side and matrix. In general, our problem would be

$$f(R, b) = R^{-1} b.$$

The algorithm computes for decreasing values of i

$$\widetilde{x}_i = \Big(\mathrm{fl}(b_i) \ominus (\mathrm{fl}(r_{i,i+1}) \otimes \widetilde{x}_{i+1}) \ominus \dots \ominus (\mathrm{fl}(r_{i,m}) \otimes \widetilde{x}_m)\Big) \oslash \mathrm{fl}(r_{i,i})$$

Backward stability would mean that we can find a triangular matrix $\widetilde{R}$ and a right-hand-side $\widetilde{b}$ such that

$$\widetilde{R}\widetilde{x} = \widetilde{b}, \quad \frac{\|\widetilde{R} - R\|}{\|R\|} = O(\epsilon_{\text{machine}}), \quad \frac{\|\widetilde{b} - b\|}{\|b\|} = O(\epsilon_{\text{machine}}),$$

where $\widetilde{x}$ is the computed solution. The proof for the 2×2 case is quite simple. You should try and do it in full detail.

HOUSEHOLDER REVISITED

What Householder triangularization produces

Let us start with an invertible matrix A. The Householder method delivers a QR decomposition

$$A = QR,$$

with R given as an upper triangular matrix and Q stored as a lower triangular matrix U, whose columns are the vectors originating the Householder reflectors:

$$Q = (I - 2u_1 u_1^*)(I - 2u_2 u_2^*) \dots (I - 2u_m u_m^*),$$

$$u_j = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ v_j \end{bmatrix}, \quad v_j \in \mathbb{C}^{m-j+1}.$$

Framing the algorithm

The matrix Q is never stored or computed. To compute

$$y = Q^* b = (I - 2u_m u_m^*) \dots (I - 2u_2 u_2^*)(I - 2u_1 u_1^*) b$$

we can proceed as follows:

```
for k=1:m
    b(k:m)=b(k:m)-2*U(k:m)*(U(k:m)'*b(k:m));
end
```

Solving systems

The algorithm consists of three steps:

1. Compute the QR decomposition $A = QR$, with Q stored as U (reflection vectors)
2. Compute $y = Q^* b$ using the multiplication algorithm
3. Solve $Rx = y$ with back substitution

All three steps are computed using floating point approximations.
We are next going to show the steps needed to prove that this algorithm is backward stable. Some of the details are not easy, so we'll just give a sketch of the entire process.

In today's lecture the norm will be the 2-norm. This is the best norm to interact with unitary matrices.

The three steps: Householder

Given A, we compute $\widetilde{Q}$ and $\widetilde{R}$ using the triangularization method. The matrix $\widetilde{Q}$ is never produced, but $\widetilde{Q}$ is unitary because it's constructed from vectors with norm exactly equal to one.

For all invertible A, the matrix $\widetilde{A}$ given by

$$\widetilde{A} = \widetilde{Q}\widetilde{R},$$

satisfies

$$\frac{\|\widetilde{A} - A\|}{\|A\|} = O(\epsilon_{\text{machine}})$$

We can also write

$$\widetilde{Q}\widetilde{R} = A + \delta A, \qquad \frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{machine}}).$$

The three steps: unitary transf. and back subs.

Now given $\widetilde{Q}$, we compute $\widetilde{y} = \widetilde{Q}^* b$, using floating point arithmetic and the Householder multiplication algorithm.

There exists a matrix δQ such that

$$(\widetilde{Q} + \delta Q)\widetilde{y} = b, \qquad \frac{\|\delta Q\|}{\|\widetilde{Q}\|} = \|\delta Q\| = O(\epsilon_{\text{machine}}).$$

There exists δR (upper triangular) such that

$$(\widetilde{R} + \delta R)\widetilde{x} = \widetilde{y}, \qquad \frac{\|\delta R\|}{\|\widetilde{R}\|} = O(\epsilon_{\text{machine}})$$

All together now

$$\widetilde{Q}\widetilde{R} = A + \delta A, \qquad \frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{machine}})$$

$$(\widetilde{Q} + \delta Q)\widetilde{y} = b, \qquad \frac{\|\delta Q\|}{\|\widetilde{Q}\|} = \|\delta Q\| = O(\epsilon_{\text{machine}}).$$

$$(\widetilde{R} + \delta R)\widetilde{x} = \widetilde{y}, \qquad \frac{\|\delta R\|}{\|\widetilde{R}\|} = O(\epsilon_{\text{machine}})$$

Backward stability of the entire process

Susbtituting the effect of the three steps

$$\begin{aligned} b &= (\widetilde{Q} + \delta Q)\widetilde{y} \\ &= (\widetilde{Q} + \delta Q)(\widetilde{R} + \delta R)\widetilde{x} \\ &= (\widetilde{Q}\widetilde{R} + (\delta Q)\widetilde{R} + \widetilde{Q}(\delta R) + (\delta Q)(\delta R))\widetilde{x} \\ &= (A + \underbrace{\delta A + (\delta Q)\widetilde{R} + \widetilde{Q}(\delta R) + (\delta Q)(\delta R)}_{\Delta A})\widetilde{x}, \end{aligned}$$

we just need to see that

$$\frac{\|\Delta A\|}{\|A\|} = O(\epsilon_{\text{machine}}).$$

Four matrices: (1) and (2)

$$\Delta A = \delta A + (\delta Q)\widetilde{R} + \widetilde{Q}(\delta R) + (\delta Q)(\delta R)$$

$$\|\delta A\| = O(\epsilon_{\text{machine}})\|A\|$$

$$\begin{aligned}
\|(\delta Q)\widetilde{R}\| &\le \|\delta Q\|\|\widetilde{R}\| = \|\delta Q\| \, \underbrace{\|\widetilde{Q}^*(A + \delta Q)\|}_{\widetilde{Q}\widetilde{R}=A+\delta A} \\
&\le O(\epsilon_{\text{machine}})\|\widetilde{Q}\|\|\widetilde{Q}^*\|\|A + \delta A\| \\
&\le O(\epsilon_{\text{machine}})(\|A\| + O(\epsilon_{\text{machine}})\|A\|) \\
&\le O(\epsilon_{\text{machine}})\|A\|
\end{aligned}$$

Note that even when $\widetilde{Q}$ is never constructed, in the way it is used, we know that $\|\widetilde{Q}\|_2 = 1$.

Four matrices: (3) and (4)

$$\Delta A = \delta A + (\delta Q)\widetilde{R} + \widetilde{Q}(\delta R) + (\delta Q)(\delta R)$$

$$\begin{aligned}\|\widetilde{Q}(\delta R)\| &\leq \|\widetilde{Q}\|\|\delta R\| \leq O(\epsilon_{\text{machine}})\|\widetilde{R}\| \\ &= O(\epsilon_{\text{machine}})\|\widetilde{Q}^*(A+\delta Q)\| \leq \ldots \leq O(\epsilon_{\text{machine}})\|A\|\end{aligned}$$

$$\begin{aligned}\|(\delta Q)(\delta R)\| &\leq \|\delta Q\|\|\delta R\| = O(\epsilon_{\text{machine}})\|\widetilde{Q}\|O(\epsilon_{\text{machine}})\|\widetilde{R}\| \\ &\leq O(\epsilon^2_{\text{machine}})\|A\|\end{aligned}$$

It is easy now to verify that we have finished the proof of backward stability

GAUSS, THE ELIMINATOR

The rules of the game

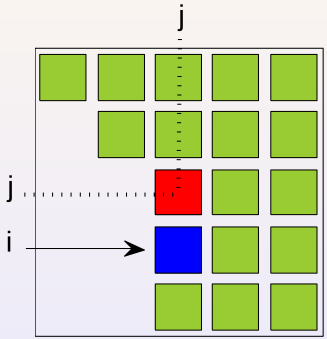
In a first stage, Gaussian elimination (applied to the augmented matrix for a system $Ax = b$) is applied as follows:

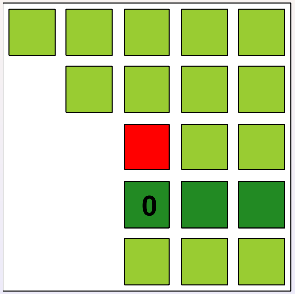
- Move from the first column to the last (of A)
- In the column j, pick a_{jj} and use it to make elimination of the elements below it in the matrix
- Keep all the changes in the same matrix
- At the end you have an upper triangular system $Ux = y$
- If at a certain moment $a_{jj} = 0$ do not do anythig yet

An elimination step

Subtract $m_{ij} = a_{ij}/a_{jj}$ times the j-th row from the i-th row, for $i = j + 1$ to m. The element m_{ij} is called the **multiplier.** The element a_{jj} is called the **pivot**.

Is a picture worth a thousand words?





1st version

```
m=size(A,1);
A=[A,b];
for j=1:m-1
    for i=j+1:m
        mult=A(i,j)/A(j,j);
        A(i,:)=A(i,:)-mult*A(j,:);    % (*)
    end
end
y=A(:,end);         % new r.h.s.
U=A(:,1:end-1);     % upper triangular
```

I hope you are seeing the waste of time in (*)

2nd and 3rd versions

Much better...

```
for j=1:m-1
    for i=j+1:m
        mult=A(i,j)/A(j,j);
        A(i,j:end)=A(i,j:end)-mult*A(j,j:end);
    end
end
```

All rows at once (in this case there's a column vector of multipliers)...

```
for j=1:m-1
    mult=A(j+1:end,j)/A(j,j);
    A(j+1:end,j:end)=A(j+1:end,j:end)-mult*A(j,j:end);
end
```

Finding a non-zero pivot

```
>> v=[0 1 3 4 0 2]'
>> find(v~=0)    % locations of non-zeros
ans =
     2
     3
     4
     6
>> find(v~=0,1)    % location of the first non-zero
ans =
     2
>> find(v,1)     % v is understood as v~=0
ans =
     2
```

4th version

We locate the first non-zero in what's relevant in the j-th column. The `find` command delivers a number k between 1 and $m - j + 1$. The actual number of row is $k + j - 1$ (which is between j and m)

```
for j=1:m-1
    k=find(A(j:end,j)~=0,1);
    k=k+j-1;   % k is relative
    A([k j],j:end)=A([j k],j:end);
    mult=A(j+1:end,j)/A(j,j);
    A(j+1:end,j:end)=A(j+1:end,j:end)-mult*A(j,j:end);
end
```

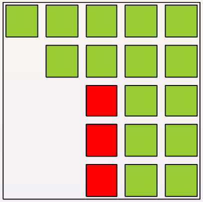
Partial pivoting

We will now look for the largest (in absolute value) possible pivot. Here's a very cool way of using the maximum finder in Matlab...

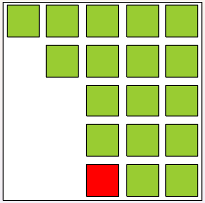
```
>> v=[0 1 -3 -5.5 0 5.5]';
>> max(abs(v))
ans =
   5.500000000000000
>> [what,where]=max(abs(v))
what =
   5.500000000000000
where =
     4
>> [~,k]=max(abs(v))
k =
     4
```

In case of ties (many maxima), we get the first one.

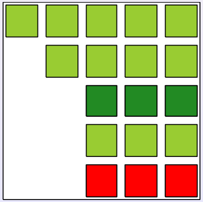
A diagram of partial pivoting



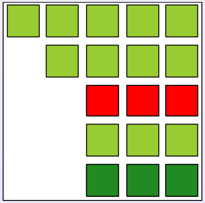
Search for a pivot



Pivot is located



These rows have to be swapped



Swapping complete

5th version: with partial pivoting

We look for the largest possible pivot in the active column. The `max` command delivers a number k between 1 and $m-j+1$. The actual number of row is $k+j-1$ (which is between j and m)

```
for j=1:m-1
    [~,k]=max(abs(A(j:end,j)));
    k=k+j-1;   % k is relative
    A([k j],j:end)=A([j k],j:end);
    mult=A(j+1:end,j)/A(j,j);
    A(j+1:end,j:end)=A(j+1:end,j:end)-mult*A(j,j:end);
end
```

ELIMINATOR, THE SEQUEL

Two versions of Gaussian elimination

```
for j=1:m-1
    k=find(A(j:end,j)~=0,1);
    k=k+j-1;
    A([k j],j:end)=A([j k],j:end);
    mult=A(j+1:end,j)/A(j,j);
    A(j+1:end,j:end)=A(j+1:end,j:end)-mult*A(j,j:end);
end

for j=1:m-1
    [~,k]=max(abs(A(j:end,j)));
    k=k+j-1;
    A([k j],j:end)=A([j k],j:end);
    mult=A(j+1:end,j)/A(j,j);
    A(j+1:end,j:end)=A(j+1:end,j:end)-mult*A(j,j:end);
end
```

Two versions of Gaussian elimination

The only difference is in the computation of the row swapping criterion:

- swap pivots when needed (unlikely to happen, not the best idea)
- partial pivoting (recommended)

Both strategies share a single idea:

At the step j (elimination in column number j), we swap row j with row k with $k \geq j$.

Before we adress this (and the storage of multipliers), let's start by saving some 'precious time' by faking the row swaps.

A simple idea

The element `A(i,j)` will be addressed as `A(row(i),j)`. Initially

$$\texttt{row(i)=i}, \qquad \texttt{i=1,...,m}.$$

When we swap rows, we will only swap the `row` vector. This is a permutation of the rows, carried out by permuting the vector that tells us where they actually are stored.

After several permutations, `row(i)` tells us the location in the matrix of the row that we use as row number *i*.

A simple example for the simple idea

This is like the shell game. Can you follow the rows?

```
>> A=[1 2 3;4 5 6;7 8 9;10 11 12];
>> row=1:4;
>> row([1 3])=row([3 1]);
>> row([2 4])=row([4 2]);
>> row([3 4])=row([4 3]);
>> row
row =
     3     4     2     1
>> A(row,:)
ans =
     7     8     9
    10    11    12
     4     5     6
     1     2     3
```

Version 6.0

At the end `U` is not upper triangular, but `U(row,:)` is. Note how `row` has to be used **always** in the first index of `A`.

```
m=size(A,1);
A=[A,b];
row=1:m;
for j=1:m-1
    [~,k]=max(abs(A(row(j:end),j)));
    k=k+j-1;
    row([k j])=row([j k]);
    mult=A(row(j+1:end),j)/A(row(j),j);
    A(row(j+1:end),j:end)=A(row(j+1:end),j:end)...
                         -mult*A(row(j),j:end);
end
c=A(:,end);
U=A(:,1:end-1);
```

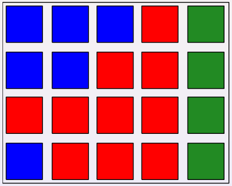
Storing the multipliers (version 6.1)

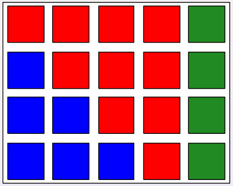
At the step j, we create $m-j$ zeros in the rows $j+1,\ldots,m$ (column j). Instead of making these zeros, we store the multipliers there. Note how `A` is still accessed with `row` to refer to its rows.

```
for j=1:m
    [~,k]=max(abs(A(row(j:end),j)));
    k=k+j-1;
    row([k j])=row([j k]);
    mult=A(row(j+1:end),j)/A(row(j),j);
    A(row(j+1:end),j+1:end)=A(row(j+1:end),j+1:end)...
                             -mult*A(row(j),j+1:end);
    A(row(j+1:end),j)=mult;
end
```

What's in A at the end

After this fake elimination, with storage of the multipliers and lots and lots of row swaps, `A` is a complete mess. However, `A(row,:)` is better.





The r.h.s is in **green**. In **red** the upper triangular matrix. In **blue**, the multipliers. The row permutation is row=`[3 4 2 1]`, `A` is on the left, `A(row,:)` on the right.

A little bit more Matlab

These two functions are not that clever, but they save some time in building a loop.

```
>> A=[1 2 3;4 5 6;7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
>> triu(A)    % upper triangular part
ans =
     1     2     3
     0     5     6
     0     0     9
>> tril(A,-1)  % lower tr, starting in -1 diagonal
ans =
     0     0     0
     4     0     0
     7     8     0
```

Wrap up and mysterious conclusion

... after doing everything we did, we unpack `A` and the right hand side in the following way:

```
y=A(row,end);
A(:,end)=[];
U=triu(A(row,:));
L=tril(A(row,:),-1)+eye(m);
```

```
L*U=A(row,:)
```

LU DECOMPOSITIONS

Permutations first – a wordy slide

Assume we are using the partial pivoting or the swap-only-when-needed strategy for Gaussian elimination. At the end we have a vector with the final permutations of the rows of the matrix. Remember that at the step j we only permute row j with one row which is under it. Therefore, the row that is placed in row j at the step j does not move from there in future steps.

It is not difficult to see that if we perform all the permutations at the beginning (before doing any elimination), the algorithm would proceed without any row changes.

Therefore, instead of applying Gaussian elimination with row changes to the matrix A, we can think of applying Gaussian elimination **with no row changes** to PA, where P is the permutation matrix associated to the final disposition of the rows due to row swaps.

Elimination

The operations performed in the j-th step of the Gaussian elimination method are equivalent to left-multiplying by the matrix

$$L_j = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -l_{j+1,j} & 1 & & \\ & & \vdots & & \ddots & \\ & & -l_{m,j} & & & 1 \end{bmatrix}$$

where $l_{j+1,j}, \ldots, l_{m,j}$ are the multipliers. (Unfortunately we are using m for the number of rows, so we need to change the letter for the multiplier.)

Matrix form of Gaussian elimination

Permutations can be moved to be the first step of the process (in theory; in practice we are figuring them out as we proceed). Elimination is carried out by the matrices L_j of the previous slide.

$$L_{m-1} \dots L_2 L_1 PA = U.$$

Therefore

$$PA = L_1^{-1} L_2^{-1} \dots L_{m-1}^{-1} U$$

Finding L (part one)

Looking at the formula

$$L_j = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -l_{j+1,j} & 1 & & \\ & & \vdots & & \ddots & \\ & & -l_{m,j} & & & 1 \end{bmatrix} \quad L_j^{-1} = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & l_{j+1,j} & 1 & & \\ & & \vdots & & \ddots & \\ & & l_{m,j} & & & 1 \end{bmatrix}$$

we can see how L_j^{-1} is the matrix that undoes the j-th elimination step: to each row, it adds back the multiplier times the j-th row.

Finding L (part two)

Therefore

$$L_1^{-1}L_2^{-1}\dots L_{m-1}^{-1} = L_1^{-1}L_2^{-1}\dots L_{m-1}^{-1}I$$

is the result of applying the opposite of the Gaussian elimination steps, in the reverse order, to the identity matrix. How difficult is it now to verify that

$$L_1^{-1}L_2^{-1}\dots L_{m-1}^{-1} = \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ l_{m1} & l_{m2} & \dots & l_{m,m-1} & 1 \end{bmatrix} \quad ?$$

Give you answer in the form of a theorem

Let P be the permutation matrix that corresponds to the final permutation of rows after the elimination process is ended. Let U be the resulting upper triangular matrix. Let L be the lower triangular matrix storing the multipliers of Gaussian elimination in the location they have been used[1] with ones in the diagonal. Then

$$LU = PA.$$

[1] This means that if the row changes place during one of the next elimination steps, the multiplier has to change place as well. Our algorithm does this automatically.