# CHAPTER 3 LECTURE NOTES: BASICS OF NUMERICAL ANALYSIS

**Thomas Shores**
**Department of Mathematics**
**University of Nebraska**
**Spring 2007**

## Contents



## 1. Nature of Numerical Computation

What does "finite precision" arithmetic mean? It means a finite number of bits of information are used to store an approximation to a given real number.

Specifically: Matlab uses IEEE (Institute of Electrical and Electronic Engineers) floating point standards, and all arithmetic is done is double precision. Thus, eight bytes are used per number, and roughly this gives about $15-16$ decimal digits of precision in a mantissa with exponents roughly in the range $10^{-308}-10^{308}$. We have already seen this in our introduction to Matlab:

```
  > eps
> x=1+eps
> y = 10*x
> y-10
> format hex
> 1
> x
> format
> realmax
> realmin
> realmin/2
> realmax/realmin
> realmin/realmax
> 0/0
```

As a matter of fact, numbers are stored in binary format as a string of 1's and 0's. The hex notation is to make binary more human readable. Fortunately, we don't need to worry too much about these details, but we do need to be aware of the potential inaccuracies of finite precision computation.

### Error Definitions and Sources

**Roundoff error**. A common notation for this floating point representation of a real number is $\mathrm{fl}(x)$ and in general

$$\mathrm{fl}(x) \neq x.$$

The error introduced by mere storage of a real number as floating point is called **roundoff (or rounding) error**.

When we perform an arithmetic operation such as $+$ we don't get exactly what we want, namely $x+y$, but rather

$$\mathrm{fl}(x) \oplus \mathrm{fl}(y)$$

where $\oplus$ is the machine operation. In spite of everything, IEEE standards can give pretty good results. Try this:

```
> format hex
> y = 0
> x = 1/3
> for ii=1:30, y=y+x; end
> y
> 10
```

Try this with $x = 1/7$, loop to 70.

Before we continue with more examples of floating point error, let's discuss the notion of error a little more systematically:

**Error:** Suppose the exact quantity that we want to estimate is $x_T$ and we end up calculating the quantity $x_A$. The **absolute error** of our calculation is

$$\mathrm{Error}(x_A) = |x_A - x_T|$$

and the **relative error** is

$$\mathrm{Error}(x_A) = \frac{|x_A - x_T|}{|x_T|},$$

provided $x_T \neq 0$.

This leads to a *formal* definition of significant digits: we say that $x_A$ approximates $x_T$ to $m$ significant (decimal) digits if

$$\frac{|x_A - x_T|}{|x_T|} \leq 5 \times 10^{-m}.$$

This is almost the same as the traditional *informal* definition that you get by eyeballing the numbers: find the leading digit $d$ of the error $|x_A - x_T|$, say it is in the $(m+1)$th place, counting to the right from the first nonzero digit in $x_T$. Then $x_A$ approximates $x_T$ to $m$ significant (decimal) digits if $d \leq 5$, otherwise the approximation is to $m-1$ significant digits.

Before we continue with more examples of floating point error, let's discuss the notion of error a little more systematically:

**Example 1.** How many significant digits does $x_A = 23.494$ have as an approximation to $x_T = 23.497$ using the formal or informal definition. What about $x_A = 23.491$?

**Solution.** Discuss.

**Example 2.** Try calculating

$$((2/7 + 1000) - 1000) - 2/7$$

and

$$((2/7 + 1000) - 2/7) - 1000$$

**Example 3**. Try using Matlab and the quadratic formula on the polynomial equation

$$x^2 - 10^9 x + 1 = 0$$

to obtain roots $x_1$ and $x_2$. Now notice the simple identity

$$(x - x_1)(x - x_2) = x^2 - (x_1 + x_2)x + x_1 x_2.$$

Check this out. Discuss.

**Solution.**

```
  > a = 1
> b = -1e9
> c = 1
> x1 = (-b+sqrt(b^2-4*a*c))/(2*a)
> x2 = (-b-sqrt(b^2-4*a*c))/(2*a)
> -(x1+x2)
> x1*x2
```

**Moral:** avoid catastrophic cancellation in arranging arithmetic of an algorithm.

**Example 4.** Plot the polynomial

$$p(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$$

on the interval $[0,2]$ with steps of $0.00001$. Then plot the same graph on the interval $[0.999, 1.001]$ using steps of $0.00001$. We'll see function evaluation noise.

**Solution.**

```
  >x = 0:0.00001:2;
>y = x.^5-5*x.^4+10*x.^3-10*x.^2+5*x-1;
>plot(x,y)
>% now repeat with new interval
```

**Example 5.** Solve the linear system

$$H_n x = b$$

where $x$ is a vector of $1$'s and $H_n$ is the Hilbert matrix of dimension $n$ for $n = 4, 8, 12, 16$.

**Solution.** The Hilbert matrix is a famous "bad boy" of numerical linear algebra. It turns up in an important approximation problem.

```
  >n = 4
>H = hilb(n);
>x = ones(n,1);
>b = H*x;
>x = H\b
>% now repeat this experiment
```

**Example 6.** Try calculating $f'(0)$ where $f(x) = exp(x)$ using finite differences

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

and get as accurate an answer as you can. (Of course, the correct answer is $1$.)

**Solution.** Use Matlab as follows

```
  >h=1e-4
>fp = (exp(0+h)-exp(0-h))/(2*h)
>% now repeat this experiment
```

**Example 7.** Let $p_n = 1/3^n$, $n = 0,1,2,\ldots$. Observe that this sequence obeys the rule

$$p_{n+1} = p_{n-1} - \frac{8}{3}p_n$$

with $p_0 = 1$ and $p_1 = 1/3$.Similarly, we see that

$$p_{n+1} = \frac{1}{3}p_n$$

with $p_0 = 1$. Use Matlab to plot the sequence $\{p_n\}_{n=0}^{50}$ directly, and then using the above recursion algorithms with $p_0$ and $p_1$ given and overlay the plot of those results. Repeat the plot with the last 11 of the points.

**Solution.** Use Matlab as follows

```
  >N=50
>p1 = (1/3).^(0:N);
>p2 = p1; p3 = p1;
>for n = 1:N,p(n+1) = (1/3)*p(n);end
>for n = 2:N,p(n+1) = p(n-1)-8/3*p(n);end
>plot([p1',p2',p3'])
>plot([p1(N-11:N)',p2(N-11:N)',p3(N-11,N)'])
```

### Sources of Error in Numerical Computation

(1) **Inaccurate model:** Implied volatility observations suggest that Black-Scholes might not be entirely accurate. Hence, no matter how refined we make our calculations, we can expect some error when we compare to reality.

(2) **Inaccurate data:** suppose we solve a Black-Scholes equation with $r = 0.065$ instead of the correct $r = 0.06$. Nothing we do short of getting exact data will save us from error. In computer science, the principle is GIGO.

(3) **Blunders:** these can be both hardware and software. Hardware problems are relatively rare nowadays, but software errors flourish.

(4) **Machine error:** this means rounding error and the error of finite precision floating point computation as in our first few examples.

(5) **Mathematical truncation:** consider the formula

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}, \text{ for } h > 0.$$

No matter how small we make $h$, we will not get the exact answer because mathematically the formula is not an exact equality. This is a bit like "mathematical roundoff."

(6) **Algorithmic instability:** we saw an example of this in Example 7, where we compute the sequence $1/3^n$ by a stable algorithm and an unstable algorithm. The problem is not in the sequence itself, but how we try to compute it. This is also an example of **error propagation**.

(7) **Mathematical instability:** this is more subtle. We saw an example of this in Example 5. The problem is not with algorithms for solving the linear system. It's deeper than that,

namely, the problem is that the Hilbert matrix is extremely sensitive to change. A bit of explanation and terminology is in order.

**Condition number of a problem:** Suppose the mathematical problem that we want to solve numerical can be expressed in functional form as

$$y = f(x).$$

Here, $x$ is the input data, and based on it, we compute $f(y)$. For the moment, forget algorithms and numerical calculations. Suppose we can do this calculation *exactly*, but the catch is that our input datum is not $x$ but a nearby value $\widetilde{x}$. The relative error in our input is

$$\frac{|\widetilde{x}-x|}{|x|}$$

while the relative error in our calculation is

$$\frac{|f(\widetilde{x})-f(x)|}{|f(x)|}.$$

Wouldn't it be nice if they were the same order of magnitude? If so, we say the mathematical problem is **well-conditioned**. If not, we say the problem is **ill-conditioned**.

Solving the system $H_n y = x$, that is,

$$y = H_n^{-1} x = f(x),$$

is an ill-conditioned problem for large $n$. A measure of the conditioning of our $y = f(x)$ problem is the smallest positive number $M$ such that

$$\frac{|f(\widetilde{x})-f(x)|}{|f(x)|} \leq M \frac{|\widetilde{x}-x|}{|x|}$$

for $\widetilde{x}$ near $x$. This can be given a more precise definition in terms of derivatives, but this will do for us. We'll call $M$ the **condition number** of the problem. There is such a measure for matrices. Try this in Matlab

```
>n = 4
>cond(hilb(n))
>% now repeat this experiment
```

### Big Oh Notation and Applications

Here's the definition:

We say a function $f(x)$ is big oh of $g(x)$ as $x \to a$ if there exists a positive number $M$ such that for $x$ sufficiently near to $a$,

$$|f(x)| \leq M\,|g(x)|\,.$$

We write

$$f(x) = \mathcal{O}(g(x)),\ x \to a.$$

When $x = n$ is a positive integer, we say $f(n)$ is big oh of $g(n)$ as $n \to \infty$ if there exists a positive number $M$ such that for $n \geq M$,

$$|f(n)| \leq M\,|g(n)|$$

and, of course, we write

$$f(n) = \mathcal{O}(g(n)),\ n \to \infty$$

or simply $f(n) = \mathcal{O}(g(n))$. There are ($\geq$) 3 major types of applications in numerical analysis of the big oh notation.

In all cases, the number $M$ is called the **order constant**. Let's look at some examples of each kind.

**Example 8.** (Behavior in the small) One can use the Taylor series with remainder term to show that if $f(x)$ has continuous second derivatives, then for a fixed $x$,

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathscr{O}(h),\ h \to 0,$$

and

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathscr{O}\left(h^2\right),\ h \to 0.$$

Think about what this says when we halve the step size $h$ in each case. Discuss.

**Example 9.** (Complexity) It can be shown that if Gaussian elimination is used to solve the linear system $Ax = b$, where $A$ is an $n \times n$ matrix, then the number of floating point operations (flops) needed is approximately $\frac{4}{3}n^3$. This is a measure of the **complexity** of the algorithm, i.e., a number proportional to its cost in computer time. More precisely, one can show that the complexity is

$$\frac{4}{3}n^3 + an^2 + bn + d = \frac{4}{3}n^3 + \text{l.o.t.} = \mathscr{O}\left(n^3\right).$$

We call this polynomial order, because that is what it is in terms of problem size $n$.

By way of contrast, there are exotic examples that show that in the worst cases, the linear programming problem of minimizing $c^T x$ subject to $Ax \geq b$, with $A$ $n \times n$ and using the simplex method, is $\mathscr{O}(2^n)$ . This is exponential order, which is BAD!

**Example 10.** (Order of convergence.) If a sequence of iterates $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)}, \ldots$ produced by some algorithm converges to the desired point $\mathbf{x}^*$, then there arises the question of how fast? We say that the sequence converges at a rate $q$ (an integer greater than or equal to 1) if

$$\left\|\mathbf{x}^{(n+1)} - \mathbf{x}^*\right\| = \mathscr{O}\left(\left\|\mathbf{x}^{(n)} - \mathbf{x}^*\right\|^q\right), n \to \infty,$$

or, equivalently,

$$\left\|\mathbf{x}^{(n+1)} - \mathbf{x}^*\right\| \leq M\left\|\mathbf{x}^{(n)} - \mathbf{x}^*\right\|^q, n \to \infty,$$

for some order constant $q$. The larger $q$, the faster the convergence. In the case $q = 1$ (linear convergence), it is required that the order constant $M < 1$. In other cases, like quadratic convergence ($q = 2$), any $M$ will do.

A few more specific examples:

**Example 11.** Show the following sequences have the corresponding convergence rates:

(1) $\left\{\frac{1}{2^n}\right\}_{n=0}^{\infty}$ converges linearly to zero.

(2) $\left\{\frac{1}{2^{2^n}}\right\}_{n=0}^{\infty}$ converges quadratically to zero

## 2. Solving Linear Systems of Equations

### Applications of the norm idea

**Error:** Suppose we set out to to compute a vector (or matrix) quantity $\mathbf{x}_T$, but due to machine error we obtain the approximate answer $\mathbf{x}_A$. The **absolute error** of our calculation is

$$\text{Error}(x_A) = \|\mathbf{x}_A - \mathbf{x}_T\|$$

and the **relative error** is

$$\text{Error}(x_A) = \frac{\|\mathbf{x}_A - \mathbf{x}_T\|}{\|\mathbf{x}_T\|},$$

provided $\mathbf{x}_T \neq \mathbf{0}$.

**Convergence**. Suppose we have a sequence of vectors (or matrices) $\{\mathbf{x}_n\}_{n=1}^{\infty}$. What does it mean to say that the sequence converges to a vector $\mathbf{x}^*$?

**Answer:** It means that the limit of the distance between $\mathbf{x}_n$ and $\mathbf{x}^*$ tends to zero as $n \to \infty$. Formally, we say that

$$\lim_{n\to\infty} \mathbf{x}_n = \mathbf{x}^* \text{ if } \lim_{n\to\infty} \|\mathbf{x}_n - \mathbf{x}^*\| = 0.$$

**Example.** Let $\mathbf{x}_n = \left[\frac{\sin n}{n}; 1 + \frac{1}{n^2}\right], n = 1, 2 \ldots$. This sequence converges to $\mathbf{x}^* = [0; 1]$. Why?

There is a very important number associated with a square invertible matrix, called its **condition number**:

$$\text{cond}(A) = \|A\| \left\|A^{-1}\right\|.$$

We'll see an application of it shortly.

Let's do some examples in Matlab, which knows all of these norms:

```
>x=[1;-3;2;-1]
>A = [2 1 3 1; 0 2 1 1;0 3 1 -1; -4 0 0 2]
>norm(A)
>norm(A*x), norm(A)*norm(x)
>norm(A,1)
>norm(A,2)
>norm(A,inf)
>eig(A'*A)
>sqrt(ans)
>cond(A),norm(A)*norm(A^(-1))
```

**An Application of condition number:**

Suppose we set out to solve a system of $n$ equations in $n$ unknowns with a theoretical unique solution $\mathbf{x}$ (so that $A$ will necessarily be invertible), of the form

$$A\mathbf{x} = \mathbf{b}.$$

However, error is introduced in both $A$ and $b$. Hence, even if we do exact arithmetic, our solution has error as well. So we end up solving the system

$$(A + \delta A)\,\mathbf{y} = \mathbf{b} + \delta\mathbf{b}.$$

Write $\mathbf{y} = \mathbf{x} + \delta\mathbf{x}$. Is there an estimate of our error?

**Fundamental Error Estimate:** With notation as above, if $r = \|\delta A\| \left\|A^{-1}\right\| < 1$ in any induced norm, then

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\text{cond}(A)}{1-r} \left\{ \frac{\|\delta A\|}{\|A\|} + \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} \right\}$$

**Moral:** relative error in our solution may grow as much as condition number of $A$ times the relative error in $A$ and $\mathbf{b}$. This strongly suggests that the condition number of the matrix $A$ is the correct measure of the sensitivity of solving the system $A\mathbf{x} = \mathbf{b}$ to error in input data. Thus, it is the condition number of the problem.

**Example 17.** Verify the fundamental error estimate in the case that $\delta A = 0$.

**Solution.** Solve for $\delta\mathbf{x}$ and use inequality $\|\mathbf{b}\| \leq \|A\|\,\|\mathbf{x}\|$.

**Example 18.** Interpret the fundamental error estimate in terms of significant digits, assuming $r \ll 1$ and that the inequality is roughly an equality.

**Solution.** Discuss. (This leads to the heuristic that if $A$ and $\mathbf{b}$ are represented to $m$ significant digits and $\text{cond}(A) \approx 10^r$ then the computed $x+\delta x$ may only approximate $x$ to $m-r$ significant digits.)

(2.2.2-3): **Direct Methods**

These are methods where one directly solves for the variables in a finite number of steps. The main players: Gaussian elimination and LU-factorization. As a matter of fact, these are really equivalent methods. We'll illustrate Gaussian elimination by a simple example.

**Example 19.** Express the following system in matrix language and solve it by Gaussian elimination.

$$\begin{aligned} x_1+x_2+x_3 &= 4 \\ 2x_1+2x_2-x_3 &= 5 \\ 4x_1+6x_2+8x_3 &= 24 \end{aligned}$$

**Solution.** It's $A\mathbf{x}=\mathbf{b}$.There are two phases. The first is "forward solve" by using each equation to eliminate a variable from subsequent equations. We'll employ a shorthand for the system called the augmented matrix and perform row operations on the equations implicitly:

$$[A\,|\,\mathbf{b}] = \begin{bmatrix} 1 & 1 & 1 & 4 \\ 2 & 2 & -1 & 5 \\ 4 & 6 & 8 & 24 \end{bmatrix} \xrightarrow{E_{21}(-2)} \begin{bmatrix} 1 & 1 & 1 & 4 \\ 0 & 0 & -3 & -3 \\ 4 & 6 & 8 & 24 \end{bmatrix}$$

$$\xrightarrow{E_{31}(-4)} \begin{bmatrix} 1 & 1 & 1 & 4 \\ 0 & 0 & -3 & -3 \\ 0 & 2 & 4 & 8 \end{bmatrix} \xrightarrow{E_{23}} \begin{bmatrix} 1 & 1 & 1 & 4 \\ 0 & 2 & 4 & 8 \\ 0 & 0 & -3 & -3 \end{bmatrix}$$

Now "back solve" to get the answer:

$$-3x_3=-3,\ x_3=1,$$

$$2x_2+4x_3=8,\ x_2=(8-4x_3)/2=2,$$

$$x_1+x_2+x_3=4,\ x_1=4-x_1-x_2=1.$$

This is the basic idea of Gaussian elimination.

Now notice that we had to swap rows of $A$ in the forward solve phase. This could be accomplished by pre-multiplication of $A$ and $\mathbf{b}$ by a "permutation matrix" $P$. Had we done this first we would have gotten

$$[PA\,|\,P\mathbf{b}] = \begin{bmatrix} 1 & 1 & 1 & 4 \\ 4 & 6 & 8 & 24 \\ 2 & 2 & -1 & 5 \end{bmatrix} \xrightarrow{E_{21}(-4)} \begin{bmatrix} 1 & 1 & 1 & 4 \\ 0 & 2 & 4 & 8 \\ 2 & 2 & -1 & 5 \end{bmatrix}$$

$$\xrightarrow{E_{31}(-2)} \begin{bmatrix} 1 & 1 & 1 & 4 \\ 0 & 2 & 4 & 8 \\ 0 & 0 & -3 & -3 \end{bmatrix}$$

Now save the negatives of the multipliers and the upper final form in the "$A$" part of the augmented matrix to obtain:

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

Try this experiment in Matlab:

```
  >b=[4;5;24]
>A = [1 1 1; 2 2 -1;4 6 8]
>A\b
>P = [1 0 0; 0 0 1;0 1 0]
>L = [1 0 0; 4 1 0;2 0 1]
>U = [1 1 1; 0 2 4;0 0 -3]
>L*U
>P*A
>[l,u] = lu(A)
>l*u
>[l,u,p]=lu(A)
>l*u
>p*A
```

**Remarks:** (1) This illustrates the so-called *LU-factorization* of $A$. Here's how it is used:
To solve the system

$$A\mathbf{x} = \mathbf{b},$$

compute an $LU$-factorization of $A$ by essentially doing Gaussian elimination on the coefficient matrix $A$ to obtain $LU = PA$. Now to solve the system, multiply on the left by the permutation matrix $P$:

$$LUx = PAx = Pb.$$

Now solve $Ly = Pb$, then $Ux = y$. Notice that each of these is a triangular solve, no worse then the back solve phase of Gaussian elimination. This is useful when the same system with new right hand sides has to be solved over and over, since the $LU$ work is only done once.

(2) Why didn't Matlab get what we got for $L, U$?

Answer is that Matlab marches to a different drummer: chooses pivot entries by scaling and "partial pivoting" techniques that make the whole operation more numerically stable.

(2.2.4) **Iterative Methods**

The basic idea is called **fixed point iteration**. Put a problem we are trying to solve into a "fixed point" form:

$$x = G(x).$$

The problem with this is that it gives $x$ *implicitly*. So turn it into an *explicit* problem by starting with an initial guess for $\mathbf{x}$ and then iterating the equation

$$\mathbf{x}^{(k+1)} = G\left(\mathbf{x}^{(k)}\right), k = 0, 1, 2, \ldots.$$

Now cross your fingers and hope that it works, that is, $\mathbf{x}^{(k)} \to \mathbf{x}$, as $k \to \infty$. If so, the iterative scheme is **convergent**, otherwise it is **divergent**.

**Example 20.** Use Matlab to find the (unique real) solution to

$$x^3 - x - 6 = 0.$$

**Solution.** Try two "splittings" of the original equation:

$$x = x^3 - 6 \text{ and } x = (6 + x)^{1/3}.$$

```
  >x = 0
>x = x^3 - 6
>% iterate the above line a few times
>x=0
>x = (6 + x)^(1/3)
>% iterate the above line a few times
  (02/28/05)
```

So how do we use this idea for a linear system

$$A\mathbf{x} = \mathbf{b}$$

**Answer**: "Split" the matrix $A$ as in

$$A = B - C,$$

then write

$$A\mathbf{x} = (B - C)\mathbf{x} = \mathbf{b},$$

so that

$$B\mathbf{x} = C\mathbf{x} + \mathbf{b},$$

and we derive an iteration scheme

$$B\mathbf{x}^{(k+1)} = C\mathbf{x}^{(k)} + \mathbf{b}.$$

**NB**: The whole idea here is that solving this system for $\mathbf{x}^{(k+1)}$ should be EASY. If not, why go to all this extra work???

**Classical Examples**: splittings for $A\mathbf{x} = \mathbf{b}$. In the following, we write

$$A = D + L + U,$$

where $D$ is the diagonal part of $A$, $L$ is the lower triangular part of $A$ and $U$ is the upper triangular part of $A$. (Note: NOT the same as $LU$-factorization. For example, with our matrix

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & -1 \\ 4 & 6 & 8 \end{bmatrix}, D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 8 \end{bmatrix},$$

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

(1) Jacobi Iteration: $A = D - (-L - U)$

(2) Gauss-Seidel (GS) Iteration:

$A = U - (-D - L)$

(3) GS-SOR Iteration: this can be applied to any iterative method (not just GS) to possibly accelerate it.

The basis of SOR is that the iteration can be written

$$\mathbf{x}^{(k+1)} = G\left(\mathbf{x}^{(k)}\right) = \mathbf{x}^{(k)} + \left(G\left(\mathbf{x}^{(k)}\right) - \mathbf{x}^{(k)}\right) = \mathbf{x}^{(k)} + \mathbf{r}^{(k)}$$

We can think of $\mathbf{r}^{(k)}$ as a "relaxation" term. The idea is to accelerate convergence by increasing or decreasing this "relaxation" term using a relaxation factor $\omega$ :

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega\mathbf{r}^{(k)}$$

In case $\omega \geq 1$, the method is called *over-relaxation*, otherwise *under-relaxation*.

So when can we be sure that these methods converge? There is a general theoretical condition that is used to show all kinds of convergence theorems.

**Convergence Test:** The iterative scheme for solving $A\mathbf{x} = \mathbf{b}$ given by the splitting

$$A = B - C,$$

namely,

$$\mathbf{x}^{(k+1)} = B^{-1}\left(C\mathbf{x}^{(k)} + \mathbf{b}\right),$$

converges for all initial guesses $\mathbf{x}^{(0)}$ and right hand sides $b$ to a unique solution if and only if $\rho\left(B^{-1}C\right) < 1$, in which case convergence is linear with order coefficient approaching $\rho(B^{-1}C)$.

For example, on can use this general theorem and some extra work to show

**Fact:** If GS-SOR converges for some $\omega$, then $0 < \omega < 2$. If $A$ is symmetric with positive diagonal entries, then the converse is true.

One final observation: the three classical methods we exhibited are not normally used in the matrix form of a splitting algorithm like $\mathbf{x}^{(k+1)} = B^{-1}\left(C\mathbf{x}^{(k)} + \mathbf{b}\right)$ – that would be extremely inefficient in conventional programming. This form is mainly used for theoretical purposes, like analyzing the behavior of the algorithm.

Rather, they take on a much simpler form as follows. Let $A = \left[a_{ij}\right]$ and $\mathbf{b} = [b_i]$ (this means that the element in the $i$th row and $j$th column of $A$ is $a_{ij}$ and the $i$th element of $\mathbf{b}$ is $b_i$.) Now let the column vector $\mathbf{x}^{(k)}$ have coordinates given by

$$\mathbf{x}^{(k)} = \left[x_1^{(k)}; x_2^{(k)}; \ldots; x_n^{(k)}\right]$$

Here are the practical forms for one iteration of these algorithms:

**Jacobi:** For $i = 1, 2, \ldots, n$,

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left\{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right\}.$$

For Gauss-Seidel we get a little clever and use new information as soon as we get it.

**Gauss-Seidel:** For $i = 1, 2, \ldots, n$,

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left\{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right\}.$$

Finally, we have

**GS-SOR:** For $i = 1, 2, \ldots, n$,

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left\{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right\}.$$

Just for the record, there is a Jacobi SOR, though in most cases it will be inferior to GS-SOR:

**J-SOR:** For $i = 1, 2, \ldots, n$,

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left\{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right\}.$$

**Matlab programming paradox:** Coding up the algorithms in Matlab using "more efficient" for loops is actually slower that coding it in "inefficient" matrix form, because Matlab is incredibly good at fast matrix arithmetic, and relatively slow as an interpreted language

## 3. Function Approximation and Interpolation

A common problem in computations: we have a function in a rather complicated theoretical format, or only a limited number of data points that represent values of the function.

**Question:** How can we approximate this function economically and accurately?

**Answer:** It depends...In the math department, we have an entire course dedicated to this question (CSCE/Math 441: Approximation Theory.)

**A Classical Method:** Polynomials! There are many ways we can use them. For example, we have Taylor series. These are good for highly local approximations.

**Example 21.** Try approximating

$$f(x) = e^x, \ -1 \le x \le 1$$

by Taylor polynomials of first through third degree polynomials.

**Solution.** First recall that

$$e^x \ = \ \left(1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots\right)$$

Now let's try plotting the function and its Taylor approximations of degree 1 to 3.

```
>x=-1:.01:2;
>plot(x,exp(x));
>hold on
>% goto graphics window and play...
>;
>x = H\b
>% now repeat this experiment
```

**Polynomials and Matlab:** Matlab has a number of built-in functions for dealing with polynomials. Note particularly polyfit, polyval and conv. Discuss the format of a "polynomial" in Matlab at the board.

**Example 22.** Same as the previous, but with interpolation.

**Solution.** First the definition: a function $f(x)$ **interpolates** $g(x)$ at nodes $x_1, x_2, \ldots, x_n$ if

$$f(x_j) = g(x_j), \ j = 1, 2, \ldots, n.$$

What can be done:

**Fact:** Given a function $f(x)$ and nodes $x_0 < x_1 < \cdots < x_n$ there is one and only one polynomial of degree $p(x)$ at degree most $n$ that interpolates $f(x)$.

**Reason:** Write it out as a linear system in the case of 3 points. The coefficient is a Vandermonde matrix, which is always nonsingular provided the $x_j$ are distinct. Discuss this point and return to the example.

Here is the calculation we need:

```
>% close the plot window and replot error of cubic
>plot(x,1+x+x.^2/2+x.^3/6-exp(x))
>grid,hold on
```

```
>xnodes = linspace(-1,1,4)
>ynodes = exp(xnodes)
poly = polyfit(xnodes,ynodes,3)
>plot(x,polyval(poly,x)-exp(x))
```

**Example 23.** Try interpolating the data set

$$x = [1,2,3,4,5,6,7,8,9,10]$$
$$y = [4,2.5,-2,-1,2,5,4,6,4.5,3].$$

**Solution.** We'll do this in Matlab with polyfit. Try larger degrees and exact fits. Then back off the degree. What is Matlab doing with something like
polyfit(x,y,n)
where $n$ is smaller then the length of $x$?

**Solution.** Try this

```
>x=1:10
>y=[4 2.5 -2 -1 2 5 4 6 4.5 3]
>plot(x,y,'o'),grid,hold on
>% now edit it a bit...
>poly = polyfit(x,y,3)
>xx = 1:.01:10;
plot(xx,polyval(poly,xx))
```

There are other ways to approximate that fit data much more smoothly and accurately. For example, one could use rational functions instead of polynomials, that is $p(x)/q(x)$, where $p(x)$ and $q(x)$ are polynomials. Or one could break the interpolating interval into pieces and approximate on each piece. This is the idea behind splines.

**Splines:** We create a curve by "knotting" together continuously or smoothly, polynomials of a given degree that interpolate given data points at the knots. Linear and cubic splines are the most common. Three basic cubic splines:

- Natural cubic splines: these minimize the "wiggle" in a curve, but are not the most accurate cubic spline. In particular, second derivatives at the endpoints will be zero, which could be far from correct.
- clamped cubic spline: in addition to interpolation at knots, they are clamped in the correct direction at the endpoints. We get this in Matlab if we create a spline with two more ordinates than abscissas. The extreme ordinates are assumed to have the extra derivative data.
- not-a-knot: these fake clamping by using two points next to the boundary. This is the default in Matlab, if ordinates and abscissas have same length.

**Example 24.** Create a cubic spline for the data of the previous example.

**Solution.** The functions we need from Matlab are spline and ppeval.

```
>plot(x,y),grid,hold on
>% now edit it a bit...
>spln = spline(x,y)
plot(xx,ppval(spln,xx))
```

**Example 25.** Use splines to approximate $\sin x$, $0 \le x \le 7$, using integer abscissas. Plot the error function.

## 4. Solving Nonlinear Equations

The basic problem is the following:

**Question:** How can we solve for the root(s) of and equation

$$f(x) = 0$$

in situations where we cannot derive an explicit analytical formula? Of course, this means that we will be satisfied with an approximate numerical answer.

Matlab has a very simple function in the basic package called `fzero`. It does not use any derivative information like Newton's method, and is based on a bisection-type idea. Matlab also has a function fminbnd for solving the optimization problem of minimizing the function $f(x)$ on a specified interval. Discuss this.

**Example 26.** Use Matlab's function fzero and fminbnd to find the zeros and minimum values of the function

$$f(x) = x - 2\sin x$$

on the interval $[0,3]$.

**Solution.** Start by getting help on these functions. Then create an anonymous function $f(x)$ and graph it on the interval. Sample calculations:

```
> myfcn = @(x) x-2*sin(x)
> fzero(myfcn,3)
> fzero(myfcn,0.5)
> fminbnd(myfcn,0,3)
> [x,y,exitflag,output] = fzero(myfcn,3)
> [x,y,exitflag,output] = fminbnd(myfcn,0,3)
```

**Example 27.** Use Matlab to solve this problem: A home buyer can afford monthly payments of at most $\$900$. What is the maximum interest rate that the buyer can afford to pay on a $\$100000$ house (after the down) with a 15 year mortgage.

**Solution.** We have a standard formula for this situation (the *ordinary annuity equation*)

$$A = \frac{P}{i}\left(1 - (1+i)^{-n}\right)$$

where $A$ is the mortgage amount, $P$ the monthly payment and $i$ is the interest rate per period over the $n$ payment periods. The unknown is $i$ . Now apply Matlab.

**Example 28.** Use Matlab to solve this problem: a European call option has strike price $\$54$ on a stock with current price $\$50$, expires in five months, with a risk-free rate of 7%. Its current price is $\$2.85$. What is the implied volatility?

**Solution.** We have a standard formula for this situation that is stored in the function bseurcall. Get help on it and set up a function of $\sigma$.

**Systems of Nonlinear Equations**

**Example 29.** The nonlinear system

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

has solution(s). Find them.

How can we solve such a system? Motivate this by Newton's method in one variable, then discuss general method.

**Chapter 4: Numerical Integration**

Here the basic question is the following:

**Question:** How can we obtain an approximate solution to a definite integral

$$I[f] = \int_a^b f(x)\,dx$$

in situations where an explicit analytic answer is not known.

**Example 30.** It is desired to approximate

$$\int_0^3 \sin(x)\, e^{x^2-x} dx.$$

We learned a few methods back in calculus. The idea is to divide the interval $[a,b]$ up into equal subintervals of width

$$h = \frac{b-a}{N}$$

by introducing points $x_0 = a, x_1 = x_0 + h$, ..., $x_N = x_0 + Nh = b$, called **nodes**, shorthand $f(x_k) = f_k$, and do something like:

- (Trapezoidal Rule) Integrate a trapezoid on each subinterval interpolating $f(x)$ at each node (draw a picture) then sum the integrals to get

$$I[f] \approx \frac{h}{2}\left(f_0 + 2\sum_{k=1}^{N-1} f_k + f_N\right)$$

- (Simpson's Rule) Integrate a parabola interpolating $f$ on double subintervals (so $N = 2M$ has to be even) (draw a picture) then sum the integrals to get

$$I[f] \approx \frac{h}{3}\left(f_0 + 4\sum_{k=1}^{M} f_{2k-1} + 2\sum_{k=1}^{M-1} f_{2k} + f_N\right)$$

Matlab has a built-in method quad that is quite powerful. It uses an "adaptive" strategy blended with a Simpson rule. Use it to solve Example 30.