No more fiction for us, we calculate. But that we may calculate, we must make fiction first.

— Thus Spake Zarathustra

Friedrich Wilhelm Nietzsche (1844-1900).

1

PROBLEMS, ALGORITHMS & SOFTWARE

1.1 INTRODUCTION

There is a grain of truth in Nietzsche's aphorism. It could be restated as 'before we calculate we must approximate'. Approximation pervades all parts of numerical computation. In numeric digital computing we must approximate because :

1. Space is finite.
2. Time is finite.
3. Arithmetic operations are limited to $+, -$, $\times$, and $/$.

The subject of these chapters could be called *Approximate Calculation*, but for historical reasons this would be misleading. Instead, it goes by various names : *Numerical Analysis*, *Numerical Methods*, *Numerical Mathematics*, *Numerical Algorithms*, and *Scientific Computing*, are widely used.

We have chosen to call this course *Numerical Algorithms* because it is primarily about algorithms or methods that are used to calculate numbers. The algorithms are *numerical* or *arithmetical* as opposed to *algebraic* or *symbolic*. Hence, when we are asked to solve the equation $ax^2 + bx + c = 0$ we do not give the answer

$$x_1, x_2 = (-b \pm \sqrt{b^2 - 4ac})/2a.$$

Instead, we ask for values of a, b, c $(1, -10, 3$ say) and we might give the solution of $x^2 - 10x + 3 = 0$ as

$$x_1 = 5 + \sqrt{22}, \;\; x_2 = 5 - \sqrt{22}.$$

Even this answer is not fully numerical because we have not *calculated* $\sqrt{22}$. The correct numerical answer is

$$x_1 = 0.309584240, \;\; x_2 = 9.690415760.$$

or, more precisely,

$$x_1 = 0.3095842401765704454343698864 6, \;\; x_2 = 9.6904157598234295545656301135 4.$$

Some might argue that we need the symbolic answer before we can calculate the numeric answer. This is not true. We can guess at values for x and see how close they come to satisfying the quadratic equation. For example, choose $x = 10$. This gives $x^2 - 10x + 3 = (10)^2 - 10 \times 10 + 3 = 3 \neq 0$. Choose $x = 9$. This gives $x^2 - 10x + 3 = (9)^2 - 10 \times 9 + 3 = -6 \neq 0$. Neither guess satisfies the equation but we now know that a number between 9 and 10 does. [WHY?]

1.1.1 Finite Space and Time

Finite Space

Pythagoras's Constant $d = \sqrt{2}$ is the length of the diagonal of the unit square. The ancient Greeks were upset when they realized that this length was *incommensurable* with the sides of the square. In other words, $\sqrt{2}$ cannot be expressed as the ratio of two integers and is therefore called an *irrational*. Euclid later gave a very nice proof-by-contradiction of this fact.

Today $\sqrt{2}$ is just as incommensurable as it was 2500 years ago. By this we mean that it cannot be represented as the ratio of two integers and, therefore, cannot be represented by a finite base$-b$ expansion. Here are the first 100 digits of its base-10 expansion :

1.4142135623730950488016887242096980785696718753769480731766797379907324784621070388503875343276415 73...

It is obvious that we cannot hope to represent $\sqrt{2}$ exactly in finite memory (space). This is one of the reasons why most numerical computations are approximate.

Finite Time

Now let us consider a calculation that uses the rationals only : the calculation of *Euler's Number* e.[1] The function e^x can be represented by the power series expansion

$$e^x = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^k}{k!} + \cdots$$

which converges for all values of x. Letting $x = 1$ we get

$$e = 1 + 1 + \frac{1}{2!} + \cdots + \frac{1}{k!} + \cdots$$

It is obvious that all numbers in this formula are rationals and we use $+$, $*$, and / to calculate it. We can calculate this formula using the recurrence

$$S_k := S_{k-1} + \frac{1}{k!}, \quad k = 2, 3, \ldots, \quad \text{with} \quad S_1 = 2.$$

The first 6 terms of this recurrence are

k	1	2	3	4	5	6
S_k	2	$\frac{5}{2}$	$\frac{8}{3}$	$\frac{65}{24}$	$\frac{163}{60}$	$\frac{1957}{720}$
$\mathrm{fl}(S_k)$	2.000000000	2.500000000	2.666666667	2.708333333	2.716666667	2.718055556

[1] Euler's Number is $e = \lim_{n\to\infty}(1 + \frac{1}{n})^n = 2.71828\ldots$. Euler's Constant is $\gamma = \lim_{n\to\infty}(H_n - \ln n) = 0.57721\ldots$.

© Derek O'Connor, September 13, 2006

Even if we assume that all results can be calculated and stored exactly we still have a problem : we must truncate the infinite series for some finite k, i.e., after a finite time. In fact this recurrence is calculating increasingly accurate rational approximations to the transcendental number e. The numbers fl(S_k) are floating point approximations to the rationals S_k.

1.1.2 Numerical versus Symbolic Calculations

It is sometimes said, in popular computer magazines and in internet discussion groups, that with the easy availability of computer algebra systems such as MATHEMATICA and MAPLE there is no need for approximate calculations anymore. Let us examine this assertion briefly.

Software systems such as MATLAB and O-MATRIX are numerical in the sense that they give answers that are numerical approximations to the true answer. Computer Algebra systems such as MATHEMATICA, MAPLE, and MAXIMA are exact and symbolic. Now, if we type this `solve(a*x^2+b*x+c = 0,x)` in MAXIMA, we get

$$x = \frac{-b - \sqrt{b^2 - 4\,a\,c}}{2a}, \quad x = \frac{-b + \sqrt{b^2 - 4\,a\,c}}{2a}.$$

Notice that this result is exact and contains exact numbers along with symbols. Neither MATLAB nor O-MATRIX can solve such a system. Consider a slightly more difficult problem. Solve the equation

$$x^3 + x^2 + x + 1 = 0. \tag{1.1}$$

The roots of this equation are $x = -1, +i, -i$. Let us see what MATLAB and MAXIMA get for this problem. In MATLAB the function `roots` gives the result

```
>> x=roots([1 1 1 1])
x = -1
    -3.05311331771918e-016 + 0.999999999999999i
    -3.05311331771918e-016 - 0.999999999999999i
```

In MAXIMA, `solve(x^3+x^2+x+1=0,x)` gives the result `[x = -1, x = +%i,x = -%i]`. This answer is perfect, but the MATLAB answer is not : the second and third roots are not exact.

Consider a slight change in (1.1). Solve the equation

$$x^3 + x^2 + x + 1 + 10^{-7} = 0 \tag{1.2}$$

MATLAB gives the answer

```
>> x=roots([1 1 1 1+1/10^7])
x = -1.00000005
     2.49999988627403e-008 + 1.000000025i
     2.49999988627403e-008 - 1.000000025i
```

In MAXIMA, `solve(x^3+x^2+x+1+1/10^7=0,x)` gives the result

$$
\begin{aligned}
x &= -\frac{1}{3} - \frac{2\left(\frac{\sqrt{3}\,i}{2}-\frac{1}{2}\right)}{9\left(\frac{\sqrt{1600000400000027}}{60000000\sqrt{3}}-\frac{200000027}{540000000}\right)^{\frac{1}{3}}} + \left(\frac{\sqrt{1600000400000027}}{60000000\sqrt{3}}-\frac{200000027}{540000000}\right)^{\frac{1}{3}}\left(-\frac{\sqrt{3}\,i}{2}-\frac{1}{2}\right),\\
x &= -\frac{1}{3} + \left(\frac{\sqrt{1600000400000027}}{60000000\sqrt{3}}-\frac{200000027}{540000000}\right)^{\frac{1}{3}}\left(\frac{\sqrt{3}\,i}{2}-\frac{1}{2}\right) - \frac{2\left(-\frac{\sqrt{3}\,i}{2}-\frac{1}{2}\right)}{9\left(\frac{\sqrt{1600000400000027}}{60000000\sqrt{3}}-\frac{200000027}{540000000}\right)^{\frac{1}{3}}},\\
x &= -\frac{1}{3} + \left(\frac{\sqrt{1600000400000027}}{60000000\sqrt{3}}-\frac{200000027}{540000000}\right)^{\frac{1}{3}} - \frac{2}{9\left(\frac{\sqrt{1600000400000027}}{60000000\sqrt{3}}-\frac{200000027}{540000000}\right)^{\frac{1}{3}}}
\end{aligned}
$$

MAXIMA'S answer is exact but is it better than MATLAB'S? It is obvious from MATLAB'S answers that a small change in the polynomial gives a small change in the roots. This is certainly not obvious when we compare MAXIMA'S results. Yes, but if we simplify MAXIMA'S answers we will get a much better result. Who will simplify the results? When MAXIMA was asked to simplify its results, it tried and failed.

1.2 STANDARD NUMERICAL PROBLEMS

Lloyd Trefethen, in his essay, *Numerical Analysis*[2], lists the main areas of Numerical Analysis as these :

1. Approximation Theory
2. Numerical Linear Algebra.
3. Numerical Solution of Differential Equations.
4. Numerical Optimization.
5. Machine Arithmetic and Rounding Errors.

We give here a brief description of the standard problems of numerical computation. These problems arise in many areas of science, engineering, economics, and operations research. They have been extensively studied and many have been 'solved' in the sense that good, efficient algorithms and software are available for their solution.

1.2.1 Approximation & Interpolation

Interpolation

The interpolation problem is : given $n+1$ points $\{(x_i, f_i), i = 0, \ldots, n\}$, find the polynomial of degree n that passes through these points. This means calculating the $n+1$ coefficients of the polynomial

[2]See Section 1.7

© Derek O'Connor, September 13, 2006

$a_0 + a_1x + a_2x^2 + \cdots a_nx^n$ which satisfy the following equations :

$$\begin{aligned} a_0 + a_1x_0 + a_2x_0^2 + \cdots + a_nx_0^n &= f_0 \\ a_0 + a_1x_1 + a_2x_1^2 + \cdots + a_nx_1^n &= f_1 \\ &\vdots \\ a_0 + a_1x_n + a_2x_n^2 + \cdots + a_nx_n^n &= f_n \end{aligned}$$

In matrix form these equations are

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ & & & \ddots & \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix}, \quad \text{or} \quad Va = f.$$

This matrix V is called the *Vandermonde Matrix* whose determinant is

$$\det(V) = \prod_{i>j}(x_i - x_j).$$

This determinant is non-zero and V is non-singular if, and only if, $x_0 \neq x_1 \neq \cdots \neq x_n$, i.e., the points are distinct. Hence these equations have a unique solution for $(a_0, a_1, \ldots, a_n)$ and so the interpolating polynomial is unique (one, and only one, straight line can go through two points).

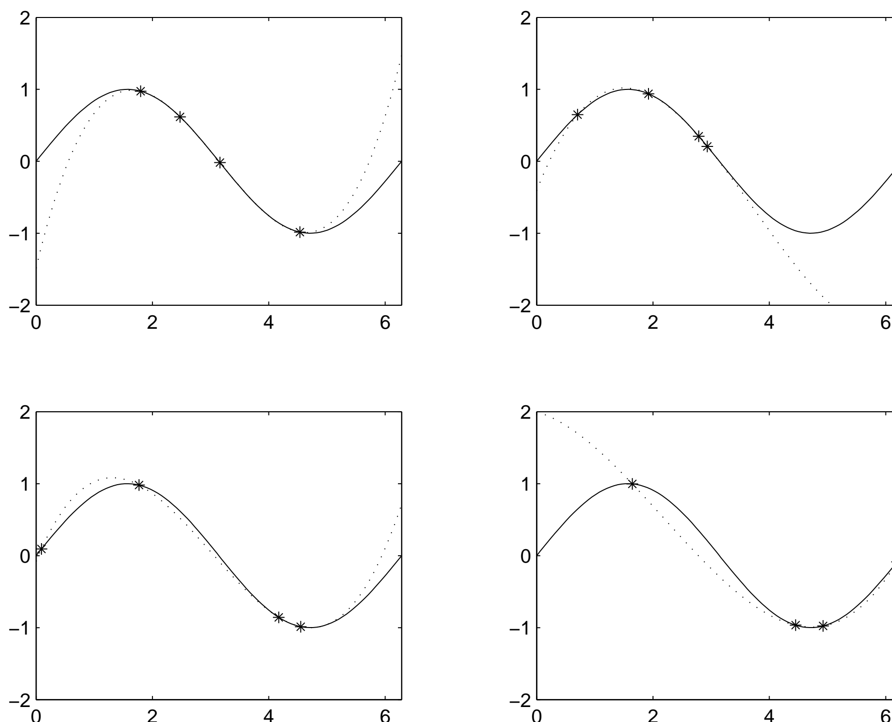

Figure 1.1 : Cubic Polynomial Interpolation of $\sin x$

Notice in Figure 1.1 that there are many different polynomials that interpolate a given function.

© Derek O'Connor, September 13, 2006

Approximation

Given a function $f(x) \in \mathscr{C}[a,b]$, and some $\epsilon > 0$, can we find a polynomial $p_n(x)$ of sufficiently high degree n for which

$$|f(x) - p_n(x)| \leq \epsilon, \quad a \leq x \leq b.$$

Weierstrass's approximation theorem (1885) shows that this is possible. This type of approximation is called *uniform*. Other types are *best* and *least squares*. Figure 1.2 shows that this is a different problem to interpolation because the polynomial is not constrained to pass through interpolation points. Also note that the shape of the approximation function $p_n(x)$ is unrestricted as long as it stays within the envelop $f(x) \pm \epsilon$.

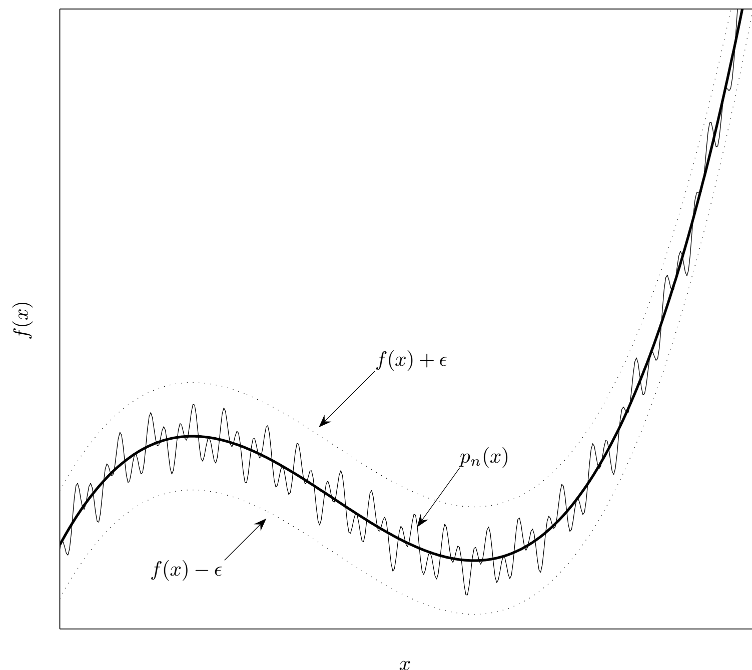

Figure 1.2 : Approximation of $f(x)$

Function Evaluation

How do we calculate tan(33.4) or $e^{-t^2/2}$? This is quite a tricky subject which uses interpolation, approximation, and other ideas. We will discuss these after the first Lab. Exercise.

1.2.2 Linear Algebra

Linear Equations

Find x that satisfies

$$Ax = b, \quad \text{where} \quad A \in \mathbb{R}^{n\times n} \quad \text{and} \quad x, b \in \mathbb{R}^n. \tag{1.3}$$

© Derek O'Connor, September 13, 2006

That is, given (A, b), find x.

This is perhaps *the* problem of numerical analysis. Some might say that this problem is trivial because we simply apply Gauss's elimination method and out pops the solution. This comment contains a grain of truth and a mountain of ignorance, as we will see later.

Algebraic Eigenvalues

Find x and λ that satisfy

$$Ax = \lambda x, \quad \text{where} \quad A \in \mathbb{R}^{n\times n},\ x \in \mathbb{R}^n, \quad \text{and} \quad \lambda \in \mathbb{C}. \tag{1.4}$$

Singular Value Decomposition

Given $A \in \mathbb{C}^{m\times n}$, find the matrices U, Σ, and V^* such that

$$A = U\Sigma V^*, \quad \text{where} \quad U \in \mathbb{C}^{m\times m},\ V \in \mathbb{C}^{n\times n} \quad \text{are unitary, and} \quad \Sigma \in \mathbb{R}^{m\times n} \quad \text{is diagonal.} \tag{1.5}$$

Furthermore, the diagonal elements of Σ are arranged as $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$, where $p = \min(m, n)$.

Singular value decomposition has become very important in the last 10–15 years because of its applications and insight it gives into matrix problems.

Least Squares

Solve $Ax = b$ for x, where $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$. With $m > n$ there are more equations than unknowns and so we solve the *linear least squares problem* :

$$\min_{x\in\mathbb{R}^n} \|b - Ax\|_2^2, \tag{1.6}$$

that is , we find x to minimize the difference between the left and right sides of $Ax = b$.

1.2.3 Optimization

Optimization is a vast subject that has many branches. Until quite recently the two main branches were Linear and Non-Linear optimization. Since the advent of Kachian's algorithm in 1979 and Karamarkar's re-discovery of interior-point algorithms for linear programming, the two branches have become closer.

Linear Programming

This, along with the Fourier Transform, is one of the most-solved problems.

$$\begin{aligned} \min_x \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i, \quad i = 1, 2, \ldots, m \\ & x_j \geq 0, \quad j = 1, 2, \ldots, n. \end{aligned} \tag{1.7}$$

© Derek O'Connor, September 13, 2006

In matrix-vector form the problem is

$$\begin{array}{rrcl} \min\limits_{x} & c^{\mathsf{T}}x & & \\ \text{subject to} & Ax & \geq & b \\ & x & \geq & 0, \end{array} \tag{1.8}$$

where A,b and c are given constants which constitute the data of the problem.

Thus we are minimizing a linear function $f:\mathbb{R}^n \to \mathbb{R}$ in n unknowns $x_1, x_2, \ldots, x_n$, subject to m linear constraints, along with n non-negativity constraints on the unknowns.

With $m < n$ there are more unknowns than equations and so there are many x's that satisfy $Ax = b$. The aim here is to choose the x that minimizes $c^T x$.

If x is constrained to be integer-valued then this is called the *Integer Programming* problem.

Unconstrained Optimization

$$\min_{x\in\mathbb{R}^n} f(x),$$

where $f:\mathbb{R}^n \to \mathbb{R}$ is a scalar-valued function of the n variables $x_1, x_2, \ldots, x_n$.

Constrained Optimization

$$\min_{x\in\mathbb{R}^n} f(x), \quad \text{subject to } \; g(x) = b,$$

where $f:\mathbb{R}^n \to \mathbb{R}$ is a scalar-valued function of the n variables $x_1, x_2, \ldots, x_n$ and $g:\mathbb{R}^n \to \mathbb{R}^m$ is a vector-valued function of the n variables $x_1, x_2, \ldots, x_n$. That is, $g(x)$ is a set of m constraint equations, $g_i(x) = b_i$, on the variables $x_1, x_2, \ldots, x_n$.

Non-Linear Equations

Solve $f(x) = 0$, where $f:\mathbb{R}^n \to \mathbb{R}$ with $0,\ x \in \mathbb{R}^n$. Here $f(.)$ is a vector-valued function and we wish to find the vector x for which the function is the 0-vector.

In component form this is

$$\begin{array}{rcl} f_1(x_1, x_2, \ldots, x_n) & = & 0, \\ f_2(x_1, x_2, \ldots, x_n) & = & 0, \\ & \vdots & \\ f_n(x_1, x_2, \ldots, x_n) & = & 0. \end{array}$$

We have included this problem in the optimization group because the theory and methods for solving it are very similar to those used in optimization.

© Derek O'Connor, September 13, 2006

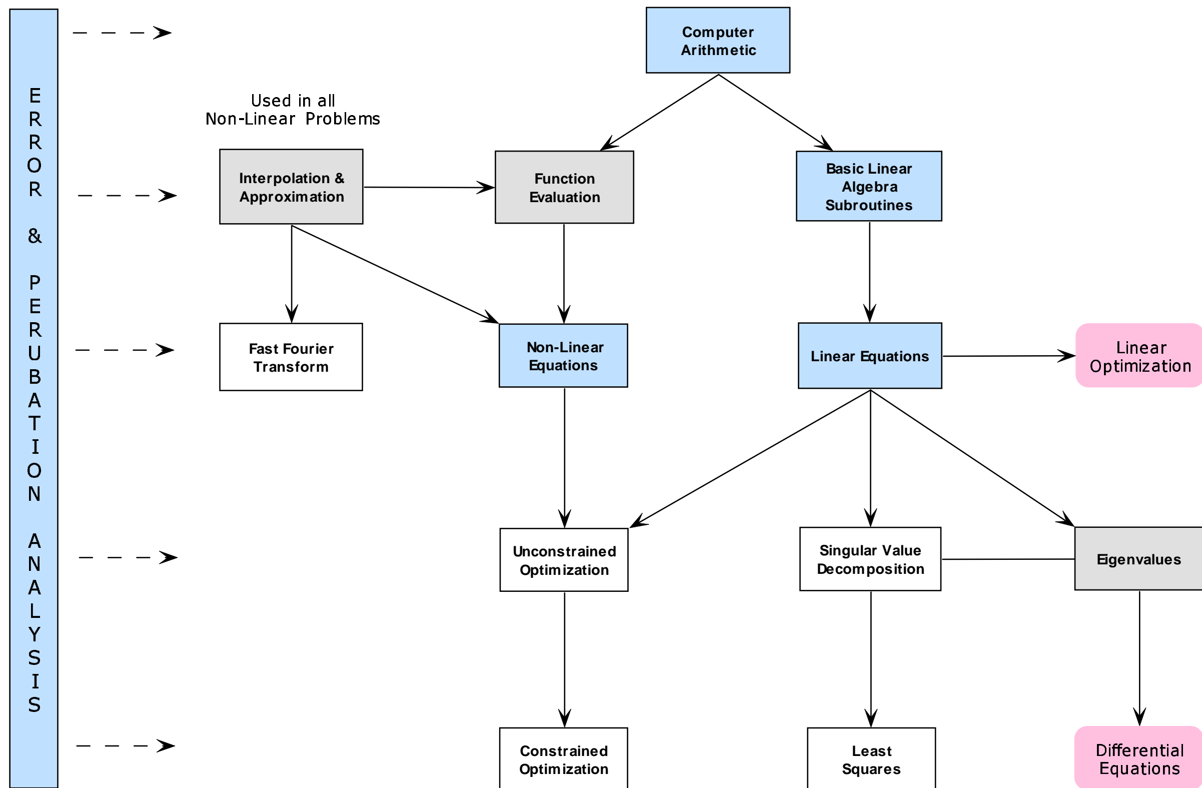

Figure 1.3 : Relationships between Problems

1.2.4 Integration & Differentiation

One example will suffice here :

$$N(\mu,\sigma) = F(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-(t-\mu)^2/2\sigma^2}\, dt,$$

has no closed form and must be evaluated numerically for all required values of x. This is why we need tables for the *Normal Distribution Function* $N(\mu,\sigma)$.

1.2.5 Differential and Integral Equations

1.2.6 Fourier Transforms

1.3 SOME TYPICAL PROBLEM AREAS

- Web Searching (Eigenvalues)
- Data Mining (SVD, Eigenvalues)
- Information Retrieval (SVD)
- Image Compression (SVD)
- Structural Engineering (Linear Equations, Eigenvalues)
- Electrical Engineering (Everything)
- Business-Operations Research (Linear, Integer, & Non-Linear Programs)
- Meteorology (Large-scale Partial Differential Equations)

Here is part of a discussion taken from the MATLAB users' group webpage :

> I am currently writing a program which involves computing the following for over 40,000 times:
>
> $$P = T \times P \times T^{\mathsf{T}} + R \times Q \times R^{\mathsf{T}}$$
>
> where T, P, R, and Q are matrix of dimensions (1740×1740), (1740×1740), (1740×1548) and (1548×1548) respectively.
>
> On my 2.4GHz PC (with 2GB RAM), each run of the above matrix computation would take 9.5 second. That means the whole thing would take almost a week to complete.
>
> Is there anyway to speed up this calculation?
>
> Any comment would be very much appreciated.

A further entry by the same author said this:

> Last night I converted all the T, P, R, and Q matrices from full to sparse and ran the code again. Before the conversion, it constantly takes 500 seconds for 50 iterations. Now, it takes only 2 seconds for the first 50 iterations. Then it quickly converges to 35 seconds/50 iterations for the next few thousands iterations. Now it's running the 18000th iteration and taking about 100 seconds/50 iterations.

© Derek O'Connor, September 13, 2006

1.4 GENERAL NUMERICAL ALGORITHMS

We may view the process of solving numerical problems as a mapping of a point $p \in P$, the *Problem Space*, to a point (or points) $s \in S$, the Solution Space. We regard p as an *instance* of a set of problems P, all of the same type, s a solution to p, and the mapping A as the algorithm or solution process, as shown in Figure 1.4.

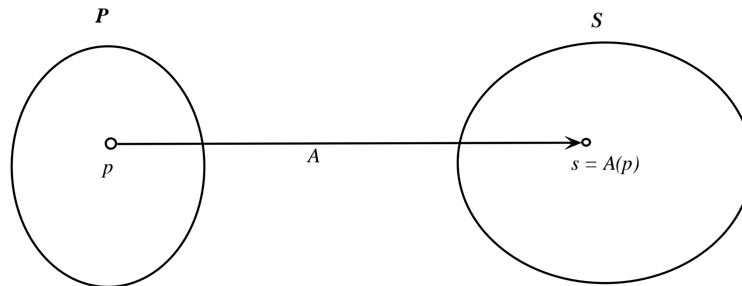

Figure 1.4 : Problem Solving

The process of problem-solving may include all aspects of problem finding, formulation, data collection, algorithm design or selection, and solution generation, interpretation, and implementation.

We will take a more restricted view of this process and limit ourselves to the application of numerical algorithms to problem data, to generate solutions. Hence we use algorithms to map points in the problem data space P to points in the solution data space S. Symbolically we have

$$A : P \longrightarrow S, \quad \text{or} \quad s \longleftarrow A(p).$$

We define an *algorithm* to be a finite set of instructions that can be carried out in a finite amount of time. Anything that is capable of carrying out these instructions is called a *computer*. Thus we may have human, biological, mechanical, digital, or even quantum computers.

There are three main ideas used in the construction of most (if not all) numerical algorithms :

- Approximation
- Iteration
- Transformation, Simplification, Decomposition.

© Derek O'Connor, September 13, 2006

1.4.1 Iterative Algorithms

These algorithms start with an initial guess at the solution and repeatedly refine this guess until a satisfactory answer is obtained.

```
algorithm  Iterate (p)
-----------------------------------------------------------
 1.  Choose s_old  initial guess at solution to p
      while s_old is not satisfactory do
 2.           s_new ← T(s_old)       refine the guess.
 3.           s_old ← s_new          use the new guess.
      endwhile
 4.  return s_old  a satisfactory solution to p
 endalg Iterate
```

We will see that all iterative algorithms have this form, no matter what problem we are solving. The heart of the algorithm is the transformation $T(\cdot)$, which must be specified for each algorithm and problem type.

Example 1.1. *Newton's Method for* $\sqrt{N}$. This is the method used in all calculators and computers. The transformation T is defined as

$$s_{\text{new}} \leftarrow (s_{\text{old}} + N/s_{\text{old}})/2.$$

With $N = 2$ and $s_{\text{old}} = 1.0$ we get

$$\begin{aligned}
s_{\text{old}} &= 1.000000 \\
s_{\text{new}} &= (1.00000000 + 2.0/1.00000000)/2.0 = 1.50000000 \\
s_{\text{old}} &= 1.500000 \\
s_{\text{new}} &= (1.50000000 + 2.0/1.50000000)/2.0 = 1.41666667 \\
s_{\text{old}} &= 1.41666667 \\
s_{\text{new}} &= (1.41666667 + 2.0/1.41666667)/2.0 = 1.41421569 \\
s_{\text{old}} &= 1.41421569 \\
s_{\text{new}} &= (1.41421569 + 2.0/1.41421569)/2.0 = 1.41421356 \\
s_{\text{old}} &= 1.414214 \\
s_{\text{new}} &= (1.41421356 + 2.0/1.41421356)/2.0 = 1.41421356
\end{aligned}$$

It can be seen that the `satisfactory` test is true when two successive iterations have the first 9 digits the same.

© Derek O'Connor, September 13, 2006

1.4.2 Direct or Transformational Algorithms

These algorithms apply a sequence of transformations to the original problem until it is easy to solve.

```
algorithm  Transform (p_orig)
 1. p ← p_orig
    while p is not easy do
 2.       p ← T(p)  --  simplify the old problem.
    endwhile
 3. s ← Solve(p)
 4. Transform s to obtain solution to the
     original problem, if necessary.
 5. return s
endalg Transform
```

Example 1.2. *Solving* $Ax = b$. In general, A is an $n \times n$ matrix and x and b are $n \times 1$ vectors. We solve the following 2×2 problem by transforming it into a trivial problem :

$$\begin{bmatrix} 1 & 1 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} \\ \frac{2}{3} \end{bmatrix}$$

We transform this matrix by adding multiples of one row to another as follows :

$$T_1 : R_2 = R_2 - 2R_1 \qquad \begin{bmatrix} 1 & 1 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$T_2 : R_2 = R_2/3 \qquad \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{2}{3} \end{bmatrix}$$

$$T_3 : R_1 = R_1 - R_2 \qquad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} \\ \frac{2}{3} \end{bmatrix}$$

The solution to this trivial problem is found by inspection : $x_1 = \frac{1}{3}$ and $x_2 = \frac{2}{3}$.

In general, A is an $n \times n$ matrix and x and b are $n \times 1$ vectors.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

© Derek O'Connor, September 13, 2006

We transform $A, b \rightarrow U, b'$ by adding multiples of one row to another to get:

$$\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}$$

This is easily solved by Back-Substitution :

$$x_n = b'_n / u_{nn}, \quad x_{n-1} = (b'_{n-1} - u_{n-1,n} x_n)/u_{n-1,n-1}, \ldots$$

Many of the algorithms in Numerical Linear Algebra are transformational in that they decompose a problem into simpler parts. Gaussian Elimination or LU Decomposition solves the linear equations problem $Ax = b$ by the decomposition $A \rightarrow LU$, where L is a lower- and U is an upper-triangular matrix. Solving $LUx = b$ is much simpler than solving $Ax = b$. Similar algorithms are QR Factorization, Singular Value Decomposition, and Spectral Decomposition.

1.5 NUMERICAL SOFTWARE

1.5.1 Introduction

Algorithms for solving numerical problems often have a deceptive simplicity. Translating such numerical algorithms into robust and reliable programs can be a difficult, error-prone task.

To illustrate these difficulties, let us write a MATLAB function to calculate the length of the vector $x = (x_1, x_2, \ldots, x_n)$. This length is defined as

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

The algorithm for this calculation is straight-forward :

```
algorithm  Length (x, n)
---------------------------------
   sum := 0
   for  i := 1  to  n  do
        sum := sum + x[i] × x[i]
   endfor
   return  √sum
endalg
```

The translation of this algorithm into MATLAB is easy :

```
function y = Length(x,n)
     sum = 0;
     for i = 1:n
        sum = sum + x(i)*x(i);
     end
     y = sqrt(sum);
```

This piece of code is not reliable. If any $x(i)$ is of the order 10^{200} then $x(i)*x(i)$ will cause **overflow**, i.e., it will not fit in a computer word and a fatal error may occur. If the $x(i)$ values are small then there is a danger of **underflow** or negative overflow, i.e., when a result becomes too small it is set to zero. For example, if $n = 10000$ and $x_i = 10^{-200}$ then the length of this vector is 10^{-180}. All of these numbers are perfectly valid computer numbers but the code above will give 0 as the result, because $x_i^2 = 10^{-400}$ will be set to 0 and usually without warning that underflow has occurred.

In fact this code is more unreliable than is suggested above : for half of all machine numbers $x \in \mathbb{F}$, the number x^2 will either underflow or overflow. The lesson to be learned from this example is

Don't write your own software unless you really have to.

Numerical software was the first software to be made available generally to the public. It pre-dates spreadsheets, wordprocessors, etc., by at least 20 years.

The first digital computing was numerical and the software written in the 10 years after WW II was either hard-wired or written in machine or assembly language for a specific machine. Obviously such software could not be distributed widely. In the mid 1950's FORTRAN was developed at IBM to automate the process of writing machine code for numerical algorithms. Hence the acronym FORmula TRANslation. From that time on most numerical software has been written in Fortran and today there are large libraries of excellent software available at nominal cost.

We now give some sources of numerical software. Some is in the public domain and is available at a nominal charge; other software is commercial or is distributed commercially.

1.5.2 Public Domain Packages

The packages below are available in FORTRAN and C and can be obtained from various sites on the Web or from some local source. (Do a Web search on NETLIB)

1. BLAS — Basic Linear Algebra Subroutines.
2. LINPACK — Linear Algebra Problems : Matrix Decomposition, Equations, Least Squares.
3. EISPACK — Eigenvalues.
4. Lapack 95 — The previous two packages have been superseded by this one. It has been the subject of an incredible amount research, code writing and testing. No package is more fundamental to scientific computing than this one. Originally written in Fortran, there is now a C version.
5. FUNPACK — Function evaluation.
6. DEPACK — Differential Equations.

© Derek O'Connor, September 13, 2006

1.5.3 Chip Manufacturers' Packages

Intel has a very active software research department. Their *Math Kernel Library* is a highly-tuned linkable package of BLAS, Lapack, and FFT routines for most of their processors. It can be linked with Digital (Compaq(Hewlett-Packard)) Fortran compiler and Intel's Fortran compiler. A *C* version is available, linkable with Microsoft Visual C. The latest version covers their Pentium III, Xeon, IV, Itanium, Itaniun 2 processors. It is downloadable free for academic users (50MB). Other manufacturers, such as IBM and AMD, have similar kernels.

1.5.4 Commercial Packages

The two sources below are similar commercial organizations that write and maintain their own software.

1. IMSL — International Mathematical & Statistical Library (USA) : Most types of numerical software; available in source or object code; price depends on amount and code type.
2. NAG — National Algorithms Group (UK).

1.5.5 Commercial Numerical Software Systems

Software Systems are integrated systems that have their own format and language for stating and solving numerical problems. Some of these are :

1. Matlab
2. Gauss
3. LINDO
4. O-Matrix
5. SPSS
6. SAS
7. and many more

1.5.6 Free Numerical Software Systems

1. Euler — Dutch. Similar to MATLAB but < 1MB, excellent)
2. Scilab — French. Similar to MATLAB.
3. Octave — GNU. Similar to MATLAB.

1.5.7 Computer Algebra Systems (CAS) and Special Topics

These are software systems that do exact (rational) arithmetic and symbolic computations. For example

$$1/23 + 5/57 \to \frac{172}{1311}$$

$$\text{integrate(1/(1+x**3),x);} \to -\frac{\log\left(x^2-x+1\right)}{6} + \frac{\arctan\left((2x-1)/\sqrt{3}\right)}{\sqrt{3}} + \frac{\log(x+1)}{3}$$

The main commercial CAS are :

1. Maple 9.0
2. Mathematica 5.0
3. MuPad 3.0

There seems to be just one free general CAS but many special purpose systems. Some of the are:

1. Maxima 5.9 (Originally the famous Macsymsa, which was the first true CAS.)
2. wxMaxima 0.6 is a very nice GUI for Maxima
3. PariGp 2.2.10 (Number Theory & high precision arithmetic)
4. Fermat 3.3 (Number Theory etc.)
5. Many more specialist systems.

1.5.8 Textbook Software

Many textbooks have software in them and some go as far as to provide a floppy disk. Generally speaking, such software is mediocre if not downright dangerous. Some exceptions are :

1. Hager, William W. : *Applied Numerical Linear Algebra*, Prentice-Hall, 1988.
2. Forsythe, G., Malcolm, M., and Moler, C. : *Computer Methods Mathematical Computations*, Prentice-Hall, 1977.
3. Kahaner, D., Moler, C, and Nash S. : *Numerical Methods and Software*, Prentice-Hall, 1989.

Numerical Recipes

This book, *Numerical Recipes*, Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., Cambridge University Press, 1986, 199?, 200?, deserves a section all to itself because it is so widely used and praised. It was one of the first books that included a floppy disk containing the source code, in Fortran 77 and Pascal, for all the algorithms in the book.[3] The book was an instant success with engineers and scientists because it gave short, snappy mathematical descriptions of the algorithms along with implementations in Fortran and Pascal. This saved people the tedium of reading specialized textbooks and papers on their particular problem, and it allowed them to 'solve' their problem right away, with the enclosed software.

COMMENT 1987.

> The book . . . is an encyclopædia of numerical methods and software. It has been much praised by users who seem to know little about writing numerical software. Testing of three of its subroutines revealed errors in each. Perhaps a 2nd edition will correct these errors. — D.O'C.

COMMENT 1997.

> The paragraph above was written around 1987. Since then a 2nd edition has appeared but the software has not improved. Indeed, it is so bad that there is a Web site devoted to pointing out its errors. This Web site is maintained at the Jet Propulsion Laboratory, California, by Fred T. Krogh. Read the handout *Why not use Numerical Recipes?* which was down-loaded from this site. This site also has links to sources of good, public-domain mathematical software.— D.O'C.

[3] It was also one of the first books to be typeset entirely in TeX.

COMMENT 2003.

Still being sold and widely cited. Fred Krogh has given up. See the newsgroup posting below — D.O'C.

From: nmm1@cus.cam.ac.uk (Nick Maclaren)
Newsgroups: comp.theory, sci.math, ucam.comp.misc
Subject: Practical algorithm references
Date: 31 Jan 1995 11:33:43 GMT

Press et al.: Numerical Recipes [in C etc.]

This is the computer equivalent of *Remove your Tree Stumps with Home-made Explosives and Save Dollars*. Most of the algorithms described are appropriate and reliable, though some are definitely unsafe, but almost all have a few restrictions on the input that they will handle correctly. The book neither warns about such restrictions, nor includes code to check for them - as is well-known to numerical analysts, this can easily lead to wrong answers without warning. These comments apply to both editions.

COMMENT 2006. *Numerical Recipes Strikes Again* :

S. Paskov in his paper 'Computing High Dimensional Integrals with Applications to Finance', at http://citeseer.ist.psu.edu/paskov94computing.html, used a random number generator from *Numerical Recipes in C*, First Ed., to generate long sequences of random numbers. These sequences were then used to obtain Monte Carlo estimates of high-dimensional integrals ($n = 360$). In a typical run 10^6 random vectors of length 360 were generated. He compared the results with those obtained using Halton and Sobol quasi-random or low-discrepancy sequences. The results did not match. He noted that there appeared to be something wrong with *NR*'s random number generator.

Some time later Akira Tajima, Syoiti Ninomiya, and Shu Tezuka wrote a paper, 'Analysis of the Anomaly of ran1() in Monte Carlo Pricing of Financial Derivatives', *Journal of the Operations Research Society of Japan*, Vol. 41, No.3, 1998, also published as IBM Research, TRL Research Report RT0187, http://www-imai.is.s.u-tokyo.ac.jp/~akira/papers/ran1.ps.gz.

They analysed the generator and found that it cycled after 3,888,000 iterations. Now $3,888,00 = 360 \times 10,800$ and this means that after generating the first 10,800 vectors of length 360, the process repeats itself, and the remaining $10^6 - 10,800 = 989,200$ or 99% of the random vectors are useless. It is well to remember that the IEEE double precision floating point number system can represent about 10^{19} different numbers and so random number generators have plenty of elbow-room. Finite precision is no excuse for short-period generators.

An where does the magic number 360 come from? Paskov was using Monte Carlo sampling to estimate the present value of Collateralized Mortgage Obligations. He used 30-year mortgages that were repaid monthly. [4]

Paskov's story is a classic example of *Faith-Based Computing* — 'it must be correct if it is published', or 'it it must be correct because many other people are using it', or, the most honest, 'I hope to God this works'. This is to be expected because the book (and implicitly the software) has been highly recommended by well-known mathematicians, scientists, engineers, and a Nobel Laureate, none of whom seems to know anything about numerical *software*.

The last-known website (Aug 2006) collecting NR war stories is at

http://www.stanford.edu/class/cme302/wnnr/nr.html

Never have so few wasted the time of so many.
— with apologies to Winston Churchill.

[4]When I got my 20-year mortgage in 1978 the bank manager used *Pear's Ready Reckoner*, 1910 Edition.

1.6 HISTORICAL NOTES

Hardware and Software Progress

We have seen impressive advances in computer hardware in the last 50 years. At present the speed of processors and the size of memory seem to increase by an order of magnitude every few years while the prices fall steadily. It is no exaggeration to say that a computer that cost $100,000 twenty years ago could be bought for $3,000 today (1992).

The impressive increase in hardware power can be seen in Table 1.1. The unit of measurement is the *flop*, or floating point operations. In MATLAB this operation is

$$\text{sum} := \text{sum} + \text{a(i, j)} * \text{x(j)},$$

and includes the most frequently-used operations in scientific computation, *viz*, floating point addition and multiplication, assignment and array indexing.[5]

Table 1.1: Super Computers (1998)

MACHINE	YEAR	MFLOPS PER SEC
Manchester Mark I	1947	0.0002
IBM 701	1954	0.003
IBM Stretch	1960	0.3
CDC 6600	1964	2
CDC 7600	1969	5
Cray-1	1976	50
Cray-2	1985	125
TMC	1992	1000
ASCI Red	1996	1,000,000
Projected	2010	1,000,000,000

ASCI Red contains 7,000 Pentium Pro processors

Thus we see that the speed has increased by a factor of more than 10^{10} in 50 years. Today (1998), desktop workstations costing $3,000 are roughly equivalent to a 1980's supercomputer costing $ millions.

Solving PDEs

Although hardware developments have been impressive there have been equally impressive improvements in algorithm design. John Rice in his book *Numerical Methods, Software and Analysis* discusses in some detail the *algorithmic* history of three-dimensional elliptic partial differential equations. The standard methods of solving such problems first *discretize* the equation. The *finite difference* method approximates the equation by finite differences at the points of a regular three-dimensional grid. In general this will give a set of $n \times n \times n$ linear equations in $n \times n \times n$ unknowns. The main computational problem is the solution of these equations.

[5] This operation is now counted as 2 flops.

© Derek O'Connor, September 13, 2006

Bentley, (see *More Programming Pearls*, 1988) has modified Rice's data and produced Table1.2 which shows the algorithmic improvements over the last 45 years.

Table 1.2: PDE Algorithms[Bentley 88]

Algorithm	Year	Flops
Gaussian Elim.	1945	n^7
SOR Iteration	1954	$8n^5$
SOR Opt ω	1960	$8n^4 \log n$
Cyclic Reduction	1970	$8n^3 \log n$
Multi-Grid	1978	$60n^3$

Notice that the Multi-Grid algorithm is essentially optimal because it is solving a set of n^3 equations in $O(n^3)$ time or flops. Just to read or write (view) n^3 pieces of data requires, of necessity, $O(n^3)$ time.

If we combine the hardware and software speed-ups we get Table 1.3

Table 1.3: Harware & Software Speed-Ups

		Algorithm		
		1950	1965	1985
Hardware	1950	206 cts	11 days	18 hrs
	1965	16 days	2 mins	7 secs
	1985	6 hrs	2 secs	$\frac{1}{10}$ secs

Remembering that a software speedup of S_s and a hardware speedup of S_h give a combined speedup of $S_{hs} = S_h \times S_s$, we get a combined speedup of roughly 10^{15}. Thus PDEs that would have taken *centuries* to solve in 1950 can be solved in fractions of a second today.

Solving Linear Programs

The following has been taken from Bixby, 2002.

> CPLEX 1.0 was introduced in 1988 and featured a strong implementation of the primal simplex method for linear programming. An implementation of the dual simplex method was introduced in CPLEX 1.2 in 1991, followed by the introduction of the CPLEX presolve algorithms in CPLEX 2.1 in 1993 and CPLEX Barrier Optimizer in CPLEX 3.0 in 1994. Table 1 illustrates the performance of different versions of CPLEX on one particular large linear program, with 49,944 rows, 177,628 variables, and 393,657 non-zeros. The CPU times given in the table were obtained on a 296 MHz Ultrasparc. Except for CPLEX 1.0, the table shows the times for the CPLEX dual simplex method to solve the problem.
>
> Coupled with the improvements in software, there have been dramatic improvements in the hardware available to solve linear programs. Table 2 illustrates the performance of the CPLEX 3.0 dual

© Derek O'Connor, September 13, 2006

Table 1.4: CPLEX on one linear program

Version	Year	Time(secs)
1.0	1988	57,840
3.0	1994	4,555
5.0	1996	3,835
6.5	1999	165
7.0	2000	161

UltraSparc 296 mhz, on pds.mps

Table 1.5: CPLEX 3.0 Dual Simplex on various machines

Machine/Chip	Year	Time(secs)
Sun 3/150	1985	44,064.0
Intel Pentium 60 MHz	1992	222.6
IBM RS/6000 Model 590	1993	65.0
SGI Power Challenge R8000 75 MHz	1994	44.8
Intel Pentium III 550 MHz	1999	31.2
AMD Athlon 650 MHz	1999	22.2
Compaq Alpha 21264 667 MHz	1999	6.8

CPLEX 3.0, on pilot.mps

simplex algorithm on the famous NETLIB problem PILOT.MPS on a range of computer hardware. Included with the table are estimates of the year that each machine was introduced by the vendor.

This table illustrates a performance improvement due to hardware innovation of a factor of 6,480, using the same code running on different hardware platforms.

Summary. Using these point samples that illustrate the performance improvement in software and the performance improvement in hardware, we can make the argument that there are linear programs that can be solved over 65,000,000 times faster today than was possible 15 years ago. We believe that there are problems that can be solved in seconds with CPLEX 7.0 today on desktop computers that would have taken 2 years to solve using the best linear programming codes and best hardware available in the late 1980s. This leads us to the following statements: Linear programming models that were impossible to solve last year can possibly be solved today.

© Derek O'Connor, September 13, 2006

1.6.1 History of my Personal Computers

Table 1.6: History of my Personal Computers

YEAR	MAKER	MODEL	OPERATING SYSTEM	PROCESSOR	SPEED MHz	MEMORY Kbytes	STORAGE Mbytes	COST Then
1980	SuperBrain	Quad	CPM 2.2	Zilog Z80	2	64	2×0.8 Fl	£4000
1985	Ericsson	PC XT	MS DOS 2.11	Intel 8088	5	256	10	£3000
1987	Quattro	PC AT	MS DOS 3.0	Intel 286	10	640	20	£3200
1992	Zenith	MasterSport	Windows 3.1	Intel 386SL	25	8000	40	£2500
1993	Quattro	PC AT	Windows 3.1	Intel 486	66	16000	500	£3200
1997	Dell	Optiplex GxPro	Windows NT 4	Pentium Pro	200	128000	4000	£4500
1998	Dell	Workstation 400	Windows NT 4	Pentium II	300	128000	4000	£3100
2000	Dell	Workstation 620	Windows 2000	Pentium III Xeon	800	640000	172000	£5600
2006?	Dell	Workstation 690	Windows Server	$2\times$ Xeon Dual Core	3400	4000000	800000	£5500

1.7 TREFETHEN'S ESSAYS

Lloyd N. Trefethen is one of the top teachers and researchers in Numerical Analysis. He has taught at Harvard, Stanford, NYU, MIT and Cornell. Since 1997 he has been a professor at Oxford University and is a Fellow of the Royal Society. Here are some of his essays on the subject that are short and not very technical. Any aspiring master of computational science should read these and heed what he has to say.

Maxims about Numerical Mathematics, Computers, Science, and Life (SIAM News, Jan/Feb 1998)

http://web.comlab.ox.ac.uk/oucl/work/nick.trefethen/maxims.html

Numerical Analysis (Princeton Companion to Mathematics, to appear), Oxford University, March 2006

http://web.comlab.ox.ac.uk/oucl/work/nick.trefethen/NAessay.pdf

The Definition of Numerical Analysis

http://web.comlab.ox.ac.uk/oucl/work/nick.trefethen/defn.ps.gz

Predictions for Scientific Computing 50 years from now.

http://web.comlab.ox.ac.uk/oucl/work/nick.trefethen/future.ps.gz

Who Invented the Great Numerical Algorithms.

http://web.comlab.ox.ac.uk/oucl/work/nick.trefethen/inventorstalk.pdf

Ten Digit Algorithms — "Ten digits, five seconds, and just one page".

http://web.comlab.ox.ac.uk/oucl/work/nick.trefethen/tda.html

© Derek O'Connor, September 13, 2006