LECTURES

LESSON I

1. Mathematics and Computer Science

1.1 Calculus

The principal topics in calculus are the real and complex number systems, the concept of limits and convergence, and the properties of functions.

Convergence of a sequence of numbers x_i is defined as follows:

The sequence x_i converges to the limit x^* if, given any tolerance $\varepsilon > 0$, there is an index $N = N(\varepsilon)$ so that for all $i \geq N$ we have $|x_i - x^*| \leq \varepsilon$. The notation for this is

$$\lim_{i \to \infty} x_i = x^*.$$

Convergence is also a principal topics of numerical computation, but with a different emphasis. In calculus one studies limits and convergence with analytic tools; one tries to obtain the limit or to show that convergence takes place. In computations, one has the same problem but little or no theoretical knowledge about the sequence. One is frequently reduced to using empirical intuitive tests for convergence; often the principal task is to actually estimate the value of the tolerance for a given x.

The study of functions in calculus revolves about continuity, derivatives, and integrals. A function $f(x)$ is continuous if

$$\lim_{x_i \to x^*} f(x_i) = f(x^*)$$

holds for all x^* and all ways for the x_i to converge to x^*. We list six theorems from calculus which are useful for estimating values that appear in numerical computation.

Theorem 1 (Mean value theorem for continuous functions). *Let $f(x)$ be continuous on the interval $[a, b]$. Consider points XHI and XLOW in $[a, b]$ and a value y so that $f(XLOW) \leq y \leq f(XHI)$.Then there is a point ρ in $[a, b]$ so that*

$$f(\rho) = y.$$

Theorem 2 (Mean value theorem for sums). *Let $f(x)$ be continuous on the interval $[a, b]$, let $x_1, x_2, \ldots, x_n$ be points in $[a, b]$ and let $w_1, w_2, \ldots, w_n$ be positive numbers. Then there is a point ρ in $[a, b]$ so that*

$$\sum_{i=1}^{n} w_i(x) f(x_i) = f(\rho) \sum_{i=1}^{n} w_i.$$

Theorem 3 (Mean value theorem for integrals). *Let $f(x)$ be continuous on the interval $[a, b]$ and let $w(x)$ be a nonnegative function $[w(x) \geq 0]$ on $[a, b]$. Then there is a point ρ in $[a, b]$ so that*

$$\int_a^b w(x) f(x) dx = f(\rho) \int_a^b w(x) dx.$$

Theorems 2 and 3 show the analogy that exists between sums and integrals. This fact derives from the definition of the integral as

$$\int_a^b f(x)dx = \lim_{\max|x_{i+1}-x_i|\to 0} \sum_i f(x_i)(x_{i+1}-x_i),$$

where the points x_i with $x_i < x_{i+1}$ are a partition of $[a,b]$. This analogy shows up for many numerical methods where one variation applies to sums and another applies to integrals. Theorem 2 is proved from Theorem 1, and then Theorem 3 is proved by a similar method. The assumption that $w(x) \geq 0$ ($w_i > 0$) may be replaced by $w(x) \leq 0$ ($w_i < 0$) in these theorems; it is essential that $w(x)$ be on one sign shown by the example $w(x) = f(x) = x$ and $[a,b] = [-1,1]$.

Theorem 4 (Continuous functions assume max/min values). *Let $f(x)$ be continuous on the interval $[a,b]$ with $|a|, |b| \leq \infty$. Then there are points XHI and XLOW in $[a,b]$ so that for all x in $[a,b]$*

$$f(XHI) \leq f(x) \leq f(XLOW).$$

The **derivative** of $f(x)$ is defined by

$$\frac{df}{dx} = f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}.$$

As an illustration of the difference between theory and practice, the quantity $[f(x+h) - f(x)]/h$ can be replaced by $f(x+h) - f(x-h)]/(2h)$ with no change in the theory but with dramatic improvement in the rate of convergence; that is, much more accurate estimates of $f'(x)$ are obtained for a given value of h. The $k-$th derivative is the derivative of the $(k-1)$th derivative; they are denoted by $d^k f/dx^k$ or $f''(x), f'''(x), f^{(4)}(x), f^{(5)}(x), \ldots$

Theorem 5 (Mean value theorem for derivatives). *Let $f(x)$ be continuous and differentiable in $[a,b]$, with $|a|, |b| < \infty$. Then there is a point ρ in $[a,b]$ so that*

$$\frac{f(b)-f(a)}{b-a} = f'(\rho)$$

$$f(x) = f(c) + f'(\rho)(x-c)$$

The special case of Theorem 5. with $f(a) = f(b) = 0$ is known as Rolle's theorem. It states that if $f(a) = f(b) = 0$, then there is a point ρ between a and b so that $f'(\rho) = 0$. This is derived from Theorem 5 by multiplying through by $b-a$, renaming a, b as x, c, and then applying the first form to the smaller interval $[x,c]$ or $[c,x]$, depending on the relation between x and c.

A very important tool in numerical analysis is the extension of the second part of Theorem 5 to use higher derivatives.

Theorem 6 (Tailor series with remainder). *Let $f(x)$ have $n+1$ continuous derivatives in $[a,b]$.*

Given points x and c in $[a,b]$ we have

$$f(x) = f(c) + f'(c)(x-c) + f''(c)\frac{(x-c)^2}{2!} + f'''\frac{(x-c)^3}{3!} + \cdots + f^{(n)}(c)\frac{(x-c)^n}{n!}$$

$$+R_{n+1}(x),$$

where R_{n+1} has either one of the following forms (ρ is a point between x and c):

$$R_{n+1}(x) = f^{(n+1)}(\rho)\frac{(x-c)^{n+1}}{(n+1)!}$$

$$R_{n+1}(x) = \frac{1}{n!}\int_c^x (x-t)^n f^{(n+1)}(t)dt$$

If a function f depends on several variables, one can differentiate it with respect to one variable, say x, while keeping all the rest fixed. This is a **partial derivative** of f and it is denoted by $\partial f/\partial x$ or f_x. Higher order and mixed derivatives are defined by successive differentiation. Taylor's series for functions of several variables is a direct extension of the formula in Theorem 6, although the number of terms in it grows rapidly. For two variables it is

$$f(x,y) = f(c,d) + f_x(x-c) + f_y(y-d) + \frac{1}{2}[f_{xx}(x-c)^2 + 2f_{xy}(x-c)(y-d)$$

$$+f_{yy}(y-d)^2] + \cdots,$$

where all the partial derivatives are evaluated at the point (c,d).

Theorem 7 (Chain rule for derivatives). *Let $f(x,y,\ldots,z)$ have continuous first partial derivatives with respect to all its variables. Let $x = x(t), y = y(t), \ldots, z = z(t)$ be continuous differentiable functions of t. Then*

$$g(t) = f(x(t), y(t), \ldots, z(t))$$

is continuously differentiable and

$$g'(t) = f_x x'(t) + f_y y'(t) + \cdots + f_z z'(t).$$

Finally, we state

Theorem 8 (Fundamental theorem of algebra). *Let $p(x)$ be a polynomial of degree $n \geq 1$, that is,*

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n,$$

where the a_i are real or complex numbers and $a_n \neq 0$. Then, there is a complex number ρ so that $p(\rho) = 0$.

1.2. Number representation

Numbers are represented in number systems. Any number of bases can be employed as the base of a number system, for example, the base 10 (decimal), 8 (octal), 12 (duodecimal), 16 (hexadecimal), or the base 2, (binary) system. The base 10, i.e. decimal system is the most common system used in human communication. In spite of not being optimal (optimal would be theoretical system with base e, base of natural logarithm, or technical system with base 3, trinary system), digital computers use, due to electronic technology, system with base 2, or binary system. In a digital computer, a binary number consists of a number of binary bits. The number of binary bits in a binary number determines the precision with which the binary number represents a decimal number. The most common size of binary number is a 32-bit number (we say, the machine word is 32 bits long, what defines the "32-bits word computer), what can represent approximately 7 digits of a decimal number. Some computer have 64 bits binary numbers, i.e. 64 bits machine word length, which can represent 13 to 14 decimal digits. For many engineering and scientific calculations, 32 bit arithmetic is good enough. But, for many other applications, 64 bit arithmetic is required. Higher precision (i.e. 64 bit, or even 128 bit) can be reached by software means, using Double precision or Quad precision, respectively. Of course, such software enhancement must be payed by even 10 times execution times of single precision calculation.

As already told, computers store numbers not with infinite precision but rather in some approximation that can be packed into a fixed number of bits (binary digits) or

bytes (groups of 8 bits). Almost all computers allow the programmer a choice among several different such representations or data types. Data types can differ in the number of bits utilized (the word-length), but also in the more fundamental respect of whether the stored number is represented in fixed-point (also called integer) or floating-point (also called real) format. A number in integer representation is exact. Arithmetic between numbers in integer representation is also exact, with the provisos that

(a) the answer is not outside the range of (usually signed) integers that can be represented, and
(b) division is interpreted as producing an integer result, throwing away any integer remainder.

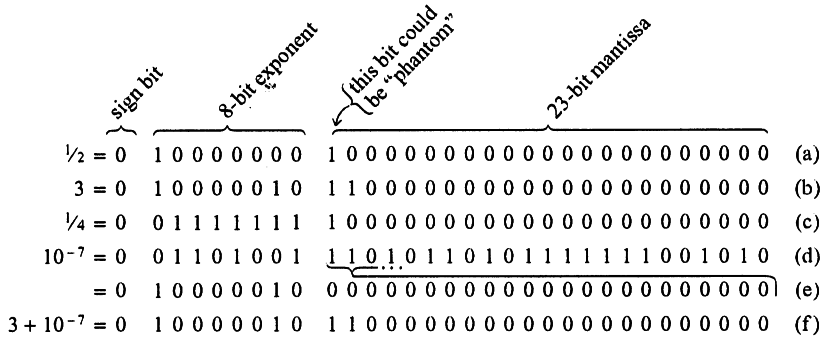

Figure 1.2.1.

In Fig. 1.2.1. are given floating point representations of numbers in a typical 32-bit (4-byte) format, with the following examples:

(a) The number $1/2$ (note the bias in the exponent);
(b) the number 3;
(c) the number $1/4$;
(d) the number 10^{-7}, represented to machine accuracy;
(e) the same number 10^{-7}, but shifted so as to have the same exponent as the number 3; with this shifting, all significance is lost and 10^{-7} becomes zero; shifting to a common exponent must occur before two numbers can be added;
(f) sum of the numbers $3+10^{-7}$, which equals 3 to machine accuracy. Even though 10^{-7} can be represented accurately by itself, it cannot accurately be added to a much larger number.

In floating-point representation, a number is represented internally by a sign bit s (interpreted as plus or minus), an exact integer exponent e, and an exact positive integer mantissa M. Taken together these represent the number

$$s \times M \times B^{e-E} \tag{1.2.1}$$

where B is the base of the representation (usually $B = 2$, but sometimes $B = 16$), and E is the bias of the exponent, a fixed integer constant for any given machine and representation.

Several floating-point bit patterns can represent the same number. If $B = 2$, for example, a mantissa with leading (high-order) zero bits can be left-shifted, i.e., multiplied by a power of 2, if the exponent is decreased by a compensating amount. Bit patterns that are "as left-shifted as they can be" are termed normalized. Most computers always produce normalized results, since these do not waste any bits of the mantissa and thus allow a greater accuracy of the representation. Since the high-order bit of a properly normalized mantissa (when $B = 2$)is always one, some computers do not store this bit at all, giving one extra bit of significance. Arithmetic among numbers in floating-point representation is not exact, even if the operands happen to be exactly represented (i.e.,

have exact values in the form of equation (1.2.1). For example, two floating numbers are added by first right-shifting (dividing by two) the mantissa of the smaller (in magnitude) one, simultaneously increasing its exponent, until the two operands have the same exponent. Low-order (least significant) bits of the smaller operand are lost by this shifting. If the two operands differ too greatly in magnitude, then the smaller operand is effectively replaced by zero, since it is right-shifted to oblivion. The smallest (in magnitude) floating-point number which, when added to the floating-point number 1.0, produces a floating-point result different from 1.0 is termed the machine accuracy m. A typical computer with $B = 2$ and a 32-bit word-length has m around 3×10^{-8}. Generally speaking, the machine accuracy m is the fractional accuracy to which floating-point numbers are represented, corresponding to a change of one in the least significant bit of the mantissa.

1.3. Error, accuracy, and stability

Except for integers and some fractions, all binary representations of decimal numbers are approximations, owing to the finite word length of binary numbers.Thus, some loss of precision in the binary representation of decimal number is unavoidable. Result of arithmetic operation among binary numbers is typically a longer binary number which cannot be represented with the number of available bits of the digital computer. Thus, the results are rounded off in the last available binary bit. This rounding-off is called round-off error. Well, pretty much any arithmetic operation among floating numbers should be thought of as introducing an additional fractional error of at least ε_m, called roundoff error. It is important to understand that ε_m is not the smallest floating-point number that can be represented on a machine. That number depends on how many bits there are in the exponent, while ε_m depends on how many bits there are in the mantissa. Roundoff errors accumulate with increasing amounts of calculation. If, in the course of obtaining a calculated value, one performs n such arithmetic operations, he might be satisfied with a total roundoff error on the order of $\sqrt{n}\varepsilon_m$, if the roundoff errors come in randomly up or down. (The square root comes from a random-walk.) However, this estimate can be very badly off the mark for two reasons:

(i) It very frequently happens that the regularities of calculation, or the peculiarities of computer, cause the roundoff errors to accumulate preferentially in one direction. In this case the total will be of order $n\varepsilon_m$.

(ii) Some especially unfavorable occurrences can vastly increase the roundoff error of single operations. Generally these can be traced to the subtraction of two very nearly equal numbers, giving a result whose only significant bits are those (few) low-order ones in which the operands differed. You might think that such a "coincidental" subtraction is unlikely to occur, what is not always true. Some mathematical expressions magnify its probability of occurrence tremendously. For example, in the familiar formula for the solution of a quadratic equation,

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \tag{1.3.1}$$

when $ac << b^2$ the addition becomes critical and round-off could ruin the calculation (see section 1.6).

Roundoff error is a characteristic of computer hardware. There is another, different, kind of error that is a characteristic of the program or algorithm used, independent of the hardware on which the program is executed. Many numerical algorithms compute "discrete" approximations to some desired "continuous" quantity. For example, an integral is evaluated numerically by computing a function at a discrete set of points, rather than at "every" point. Or, a function may be evaluated by summing a finite

number of leading terms in its infinite series, rather than all infinity terms. In cases like this, there is an adjustable parameter, e.g., the number of points or of terms, such that the "true" answer is obtained only when that parameter goes to infinity. Any practical calculation is done with a finite, but sufficiently large, choice of that parameter.

The discrepancy between the true answer and the answer obtained in a practical calculation is called the truncation error. Truncation error would persist even on a hypothetical, "perfect" computer that had an infinitely accurate representation and no roundoff error. As a general rule there is not much that a programmer can do about roundoff error, other than to choose algorithms that do not magnify it unnecessarily. Truncation error, on the other hand, is entirely under the programmers control. In fact, it is only a slight exaggeration to say that clever minimization of truncation error is practically the entire content of the field of numerical analysis.

Most of the time, truncation error and roundoff error do not strongly interact with one another. A calculation can be imagined as having, first, the truncation error that it would have if run on an infinite-precision computer, and in addition, the roundoff error associated with the number of operations performed.

Some computations are very sensitive to round-off and others are not. In some problems sensitivity to round-off can be eliminated by changing the formula or method. This is always possible; there are many problems which are inherently sensitive to round-off and any other uncertainties. Thus we must distinguish between sensitivity of *methods* and sensitivity inherent in *problems*.

The word *stability* appears during numerical computations and refers to continuous dependence of a solution on the *data* of the problem or *method*. If one says that a method is *numerically unstable*, one means that the round-off effects are grossly magnified by the method. Stability also has precise technical meaning (not always the same) in different areas as well as in this general one.

Solving differential equations usually leads to difference equations, like

$$x_{i+2} = -(13/6)x_{i+1} + (5/2)x_i.$$

Here, the sequence $x_1, x_2, \ldots$ is defined, and for given initial conditions x_1 and x_2 of differential equation, we get the initial conditions for difference equation. For example, $x_1 = 30,\ x_2 = 25$. Computing in succession for $4, 8, 16, 32, 64$ decimal digits gives the results that can be compared with the exact one, $x_i = 36/(5/6)^i$. (Compute in `Mathematica`, using $N[x[I+2], k]$, where $k = 4, 8, 16, 32, 64$ number of decimal digits).

i	4	8	16	True value
1	30.00	30.00	30.00	30.00
2	25.00	25.00	25.00	25.00
3	20.83	20.8333	20.8333	20.8333
4	17.36	17.3611	17.3611	17.3611
5	14.46	14.4676	14.4676	14.4676
6	12.07	12.0563	12.0563	12.0563
7	10.00	10.0470	10.0469	10.0469
8	8.518	8.3724	8.3724	8.3724
9	6.541	6.9773	6.9770	6.9770
10	7.121	5.8133	5.8142	5.8142
11	.925	4.8478	4.8452	4.8452
12	15.790	4.0296	4.0376	4.0376
13	−31.920	3.3888	3.3647	3.3647
14	108.700	2.7318	2.8039	2.8039
16	954.600	1.2978	1.9472	1.9472

18	8576.000	-4.4918	1.3522	1.3522
20	77170.000	-51.6565	.9390	.9390
22	6.9×10^5	-472.7080	.6521	.6521
25	-1.8×10^7	12781.1000	.3776	.3774
28	5.0×10^8	-345079.0000	.2134	.2184
30	4.5×10^9	-3.1×10^6	.1071	.1517
35	-1.1×10^{12}	7.5×10^8	10.8822	.0609
40	-1.1×10^{14}	-1.8×10^{11}	-2629.5300	.0245
50	1.5×10^{19}	-1.0×10^{16}	-1.5×10^8	.0039
75	1.3×10^{31}	9.2×10^{27}	1.3×10^{20}	.00

This difference equation is unstable and one can see that the computation quickly "blows up". One nice thing about unstable computation is that they usually produce huge, nonsense numbers that one is not tempted to accept as correct. However, imagine that one wanted only 30 terms of the x_i and was using the computer with 16 decimal digits. How would one know that the last term is in error by 50 percent ?

The word *condition* is used to describe the sensitivity of problems to uncertainty. Imagine the solution of a problem being obtained by evaluation a function $f(x)$. Then, if x is changed a little to $x+\delta x$, the value $f(x)$ also changes. The relative *condition number* of this change is

$$\frac{|f(x+\delta x)-f(x)|}{|f(x)|} / |\frac{\delta x}{x}|,$$

or

$$\frac{f(x+\delta x)-f(x)}{\delta x} \times \frac{x}{f(x)},$$

and, for δx very small, condition number c is

$$c \sim \frac{x f'(x)}{f(x)}.$$

This number estimates how much an uncertainty in the data x of a problem is magnified in its solution $f(x)$. If this number is large, then the problem is said to be *ill-conditioned* or poorly conditioned.

The given formula is for the simplest case of a function of a single variable; it is not easy to obtain such formulas for more complex problems that depend on many variables of different types. We can see three different ways that a problem can have a large condition number:

1. *$f'(x)$ may be large while x and $f(x)$ are not*;

   If we evaluate $1+\sqrt{|x-1|}$ for x very close to 1, then x and $f(x)$ are nearly 1, but $f'(x)$ is large and the computed value is highly sensitive to change in x.

2. *$f(x)$ may be small while x and $f'(x)$ are not*;

   The Taylor's series for $\sin x$ near π or e^{-x} with x large exhibit this form of ill conditioning.

3. *x may be large while $f'(x)$ and $f(x)$ are not*;

   The evaluation of $\sin x$ for x near 1000000π is poorly conditioned.

One can also say that computation is ill-conditioned and this is the same as saying it is numerically unstable. The condition number gives more information than just saying something is numerically unstable. It is rarely possible to obtain accurate values for condition numbers but one rarely needs much accuracy; an order of magnitude is often enough to know.

Note that is almost impossible for a method to be numerically stable for an ill-conditioned problem.

Example 1.3.1. An **ill-conditioned line intersection problem** consists in computing the point of intersection P of two nearly parallel lines. It is clear that a minor change in one line changes the point of intersection to $(P+\delta P)$ which is far from P. A mathematical model of this problem is obtained by introducing a coordinate system and writing equations

$$y = a_1 x + b_1$$
$$y = a_2 x + b_2$$

what leads to solving a system of equations

$$a_1 x - y = -b_1$$
$$a_2 x - y = -b_2$$

with the a_1 and a_2 nearly equal since the lines are nearly parallel. This numerical problem is unstable or ill-conditioned, as it reflects the ill-conditioning of the original problem.

A mathematical model is obtained by introducing a **coordinate system**. Any two vectors will do for a basis, and if we chose to use the unusual basis

$$\mathbf{b_1} = (0.5703958095, 0.8213701274)$$
$$\mathbf{b_2} = (0.5703955766, 0.8213701274)$$

then every vector $\mathbf{x}$ can be expressed as

$$\mathbf{x} = x\mathbf{b_1} + y\mathbf{b_2}$$

so that the equations of the two lines in this coordinate system are

$$y = -0.0000000513 + 0.9999998843x$$
$$y = -0.0000045753 + 1.000001596x$$

with the point of intersection P with coordinates $(-0.8903429339, 0.8903427796)$. Note that mathematical model is very ill-conditioned; a change of 0.0000017117 in the data makes the two lines parallel, with no solution.

The poor choice of a basis in the given example made the problem poorly conditioned. In more complex problems it is not so easy to see that a poor choice has been made. In fact, a poor choice is sometimes the most natural thing to do. For example, in problems involving the polynomials, one naturally takes vectors based on $1, x, x^2, \ldots, x^n$ as a basis, but there are terribly ill-conditioned even for n moderate in size.

Example 1.3.2. System of equations (input information)

$$2x + 6y = 8$$
$$2x + 6.0001y = 8.0001$$

have a solutions (output information) $x = 1,\ y = 1$. If the coefficients of second equation slightly change, i.e. if one takes the equation

$$2x + 5.99999y = 8.00002,$$

the solutions are $x = 10,\ y = -2$. This is typical round-off error.

Errors in methods occur usually because in numerical mathematics the problem to be solved is replaced by another one, closed to original, which is easier to solve.

Example 1.3.3. Integral $\int_a^b f(x)dx$ can be approximately calculated, for example, by replacing the function f by some polynomial P on segment $[a,b]$, which is in some sense close to given function. However, for approximative calculation it is possible to use the sum

$$\sum_{i=1}^{n} f(x_i)\Delta x_i.$$

In both cases the method error occurs.

In some sense, the round-off error are also method errors. Sum of all errors makes the total error.

Sometimes, however, an otherwise attractive method can be unstable. This means that any roundoff error that becomes "mixed into" the calculation at an early stage is successively magnified until it comes to swamp the true answer. An unstable method would be useful on a hypothetical, perfect computer; but in this imperfect world it is necessary for us to require that algorithms be stable or if unstable that we use them with great caution. Here is a simple, if somewhat artificial, example of an unstable algorithm (see [4], p.20).

Example 1.3.4. Suppose that it is desired to calculate all integer powers of the so-called "Golden Mean," the number given by

$$\Phi \equiv \frac{\sqrt{5}-1}{2} \approx 0.61803398 \tag{1.3.2}$$

Powers of Φ^n satisfy simple recurrence relation

$$\Phi^{n+1} = \Phi^{n-1} - \Phi^n\,. \tag{1.3.3}$$

Well, knowing the first two values $\Phi^0 = 1$ and $\Phi^1 = 0.61803398$, we can apply (1.3.3) by subtraction, rather than a slower multiplication by Φ, at each stage. Unfortunately, the recurrence (1.3.3) also has another solution, namely the value $-\frac{1}{2}(\sqrt{5}+1)$. Since the recurrence is linear, and since this undesired solution has magnitude greater than unity, any small admixture of it introduced by roundoff errors will grow exponentially. On a typical machine with 32-bit word-length, (1.3.3) starts to give completely wrong answers by about $n = 16$, at which point Φ^n is down to only 10^{-4}. Thus, the recurrence (1.3.3) is unstable, and cannot be used for the purpose stated.

On the end of this section, it remains the question: How to estimate errors and uncertainty ?

One almost newer knows the error in a computed result unless one already knows the true solution, and so one must settle for estimates of the error. There are three basic approaches to error estimates. The first is *forward error analysis*, when one uses the theory of the numerical method plus information about the uncertainty in the problem and attempts to predict the error in the computed result. The information one might use includes

- the size of round-off,
- the measurement errors in problem data,
- the truncation errors in obtaining the numerical model from the mathematical model,
- the differences between the mathematical model and the original physical model.

The second approach is *backward error analysis*, where one takes a computed solution and sees how close it comes to solving the original problem. The backward error is often called the the *residual* in equations. This approach requires that the problems involve satisfying some conditions (such as an equation) which can be tested with a

trial solution. This prevents it from being applicable to all numerical computations, e.g. numerically estimating the value of π or the value of an integral.

The third approach is *experimental error analysis*, where one experiments with changing the computations, the method, or the data to see the effect they have on the results. If one truly wants certainty about the accuracy of a computed value, then one should give the problem to two (or even more) different groups and ask to solve it. The groups are not allowed to talk together, preventing a wrong idea from being passing around.

The relationship between these three approaches could be illustrated graphically, as given in the following figure.

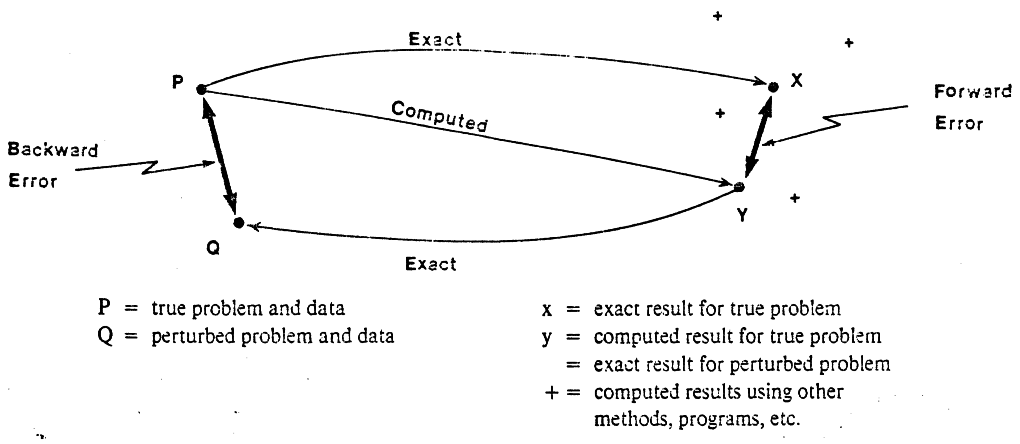

Figure 1.3.1.

1.4. Programming

There are several areas of knowledge about programming that are needed for scientific computation. These include knowledge about:

- The programming language (FORTRAN, Pascal, C, Java, Mathematica (MatCAD, **Matlab**).
- The computer system in which the language runs
- Program debugging and verifying the correctness of results
- Computation organization and expressing them clearly.

Debugging programs is an art as well as a science, and it must be learned through practice. There are several effective tactics to use, like:

- Intermediate output
- Consultations about program with experienced user
- Use compiler and debugging tools.

Some abilities of compilers:

- Cross-reference tables
- Tracing
- Subscript checking
- Language standards checking.

Some hints:

- Use lots of comments

- Use meaningful names for variables
- Make the types of variables obvious
- Use simple logical control structures
- Use program packages and systems (Mathematica, Matlab) wherever possible
- Use structured programming
- Use (if possible) OOP technics for technical problems.

1.5. Numerical software

There are several journals that publish individual computer programs:

- ACM Transactions on Mathematical Software (IMSL, International Mathematical Scientific Library)
- Applied Statistics
- BIT
- The Computer Journal
- Numerische Mathematik

The ACM Algorithms series contains more than thousand items and is available as the **Collected Algorithms of the Association for Computing Machinery**.

Three general libraries of programs for numerical computations are widely available:

IMSL	International Mathematical Scientific Library
NAG	Numerical Algorithms Group, Oxford University
SSP	Scientific Subroutine Package, IBM Corporation

There are a substantial number of important, specialized software packages. Most of the packages listed below are available from IMSL, Inc.

MP	Multiple Precision Arithmetic Package
BLAS	Basic Linear Algebra Subroutines
DEPACK	Differential Equation Package
DSS	Differential System Simulator
EISPACK	Matrix Eigensystems Routines
FISHPACK	Routines for the Helmholtz Problem in Two or Three Dimensions
FUNPACK	Special Function Subroutines
ITPACK	Iterative Methods
LINPACK	Linear Algebra Package
PPPACK	Piecewise Polynomial and Spline Routines
ROSEPACK	Robust Statistics Package
ELLPACK	Elliptic Partial Differential Equations
SPSS	Statistical Package for the Social Sciences.

User interface to the IMSL library:)

PROTRAN	John R. Rice, Purdue University

1.6. Case study: Errors, round-off, and stability

Example 1.6.1. Solve quadratic formula

$$ax^2 + bx + c = 0$$

with $5, 10, 15, \ldots 100$ decimal digits using `FORTRAN` and `Mathematica` code. Take $a = 1,\ c = 2, b = 5.2123(10)105.2123$. Use the following two codes:

```
X1=(-B+DIS)/(2*A)      IF(B.LT.0) THEN
X2=(-B-DIS)/(2*A)      X1=(-B+DIS)/(2*A)
                       ELSE
                       X1=(-B-DIS)/(2*A)
                       ENDIF
                       X2=C/X1
```

Compare the obtained results.

There are two important lessons to be learned from example 1.6.1.:

1. *Round-off error can completely ruin a short, simple computation.*
2. *A simple change in the method might eliminate adverse round-off effects.*

Example 1.6.2. Calculation of π.

Using five following algorithms, calculate π in order to illustrate the various effects of round-off on somewhat different computations.

Algorithm 1.6.2.1. Infinite alternate series

$$\pi = 4(1 - 1/3 + 1/5 - 1/7 + 1/9 - \cdots)$$

Algorithm 1.6.2.2. Taylor's series of $\arcsin(1/2) = \pi/6$

$$\pi = 6(0.5 + \frac{(0.5)^2}{2 \times 3} + \frac{1 \times 3(0.5)^4}{2 \times 4 \times 5} + \frac{1 \times 3 \times 5(0.5)^6}{2 \times 4 \times 6 \times 7} + \cdots)$$

Algorithm 1.6.2.3. Archimedes' method. Place $4, 8, 16, \ldots, 2^n$ triangles inside a circle. The area od each triangle is $1/2 \sin(\theta)$. The values of $\sin(\theta)$ are computed by the half angle formula

$$\sin(\theta) = \sqrt{[1 - \cos(2\theta)]/2}$$

and

$$\cos(\theta) = \sqrt{1 - \sin^2 \theta}.$$

The calculation is initialized by $\sin(\pi/4) = \cos(\pi/4) = 1/\sqrt{2}$. As the number of triangles grows, they fill up the circle and their total area approaches π. (Archimed carried a similar procedure by hand with 96 triangles and obtained

$$3.1409\ldots = 3\frac{1137}{8069} < \pi < 3\frac{1335}{9347} = 3.1428\ldots)$$

Algorithm 1.6.2.4. Instead of inscribing triangles in a circle, we inscribe trapezoids in a quarter circle. As a number of trapezoids increases, the sum of their areas approaches $\pi/4$.

Algorithm 1.6.2.5. Monte Carlo integration.

Monte Carlo integration for $\int_0^2 \frac{2}{1+x}\, dx$ is proceeded by choosing a pair (x, y) at random with x, y in $[0, 2]$, and comparing y with $2/(1 + x)$. If $y \leq 2/(1 + x)$ then the point (x, y) is under the curve $y = 2/(1 + x)$ and variable SUM is increased by 1. After `M` pairs, the integral is estimated by the fraction `SUM/M` of points that are under the curve.

Bibliography (Cited references and further reading)

[1] Milovanović, G.V., *Numerical Analysis I*, Naučna knjiga, Beograd, 1988 (Serbian).
[2] Milovanović, G.V. and Djordjević, Dj.R., *Programiranje numeričkih metoda na FORTRAN jeziku.* Institut za dokumentaciju zaštite na radu "Edvard Kardelj", Niš, 1981 (Serbian).

[3] Hoffman, J.D., *Numerical Methods for Engineers and Scientists.* Taylor & Francis, Boca Raton-London-New York-Singapore, 2001.
[4] Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., *Numerical Recepies - The Art of Scientific Computing.* Cambridge University Press, 1989.
[5] Rice, J.R., *Numerical Methods, Software, and Analysis.* McGraw-Hill, New York, 1983.
[6] Abramowitz, M., and Stegun, I.A., *Handbook of Mathematical Functions.* National Bureau of Standards, Applied Mathematics Series, Washington, 1964 (reprinted 1968 by Dover Publications, New York).
[7] Hildebrand, F.B., *Introduction to Numerical Analysis.* Mc.Graw-Hill, New York, 1974.
[8] Stoer, J., and Bulirsch, R., *Introduction to Numerical Analysis.* Springer-Verlag, New York, 1980.
[9] Kahaner, D., Moler, C., and Nash, S., *Numerical Methods and Software.*, Prentice Hall, Englewood Cliffs, 1989.
[10] Johnson, L.W., and Riess, R.D., *Numerical Analysis.* 2nd ed., Addison- Wesley, Reading, 1982.
[11] Wilkinson, J.H., *Rounding Errors in Algebraic Processes.* Prentice-Hall, Englewood Cliffs, 1964.
[12] Milovanović, G.V. and Kovačević, M.A., *Zbirka rešenih zadataka iz numeričke analize.* Naučna knjiga, Beograd, 1985. (Serbian).
[13] Forsythe, G.E., Malcolm, M.A., and Moler, C.B., *Computer Methods for Mathematical Computations.* Englewood Cliffs, Prentice-Hall, NJ, 1977.
[14] *IMSL Math/Library Users Manual* , IMSL Inc., 2500 City West Boulevard, Houston TX 77042.
[15] *NAG Fortran Library*, Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.