AN INTRODUCTION TO COMPUTERS AND NUMERICAL ANALYSIS

Oscar E. LANFORD III

Text for a series of lectures given at Les Houches Session XLIII (Critical Phenomena, Random Systems, Gauge Theories) August 1984.

Institut des Hautes Etudes Scientifiques
35, Route de Chartres
91440 Bures-sur-Yvette (France)

May 1985

IHES/P/85/29

1. Some preliminaries on computers.

In this section we introduce a number of fundamental ideas and pieces of terminology about computers. All these ideas will probably be at least somewhat familiar to everyone; this section is mostly intended to provide some common ground for what is to follow. The reader should be warned that what is said here will often be oversimplified; qualifications and details which obscure the main ideas are often omitted without warning.

In the most general terms, the function of a computer is to manipulate information. Internally, this information can be thought of as represented by sequences of 0's and 1's. (On a more concrete level, these 0's and 1's are actually represented by physical quantities—e.g., a voltage arranged to lie in one of two distinguishable ranges, or the presence or absence of an electrical pulse in a specified time interval.) A single 0 or 1 is referred to as a bit (abbreviation for *binary digit*). Usually, one is not directly interested in sequences of 0's and 1's but with other objects like numbers or characters, so it is necessary to choose a way of representing these objects as strings of 0's and 1's. There is evidently room for a good deal of arbitrariness in the choice of such representations, and a lack of standardization can be a serious practical problem. Here are some examples of representations of objects of interest by strings of bits:

Representation of integers. One very natural interpretation for a sequence of n bits is as the binary representation of an integer in the range $0, \ldots, 2^{n-1}$ (in the same way that a sequence of n decimal digits is interpreted as representing an integer in the range $0, \ldots, 10^{n-1}$.) This interpretation is so natural that one often does not distinguish between the bit string and the corresponding integer; it is called the *unsigned integer* interpretation of the bit string. For many purposes, of course, it is necessary to be able to represent negative as well as positive integers. It is not difficult to imagine a number of sensible ways of representing not-necessarily-positive integers; the most commonly used one is the following: A string of n bits is interpreted as representing the unique integer in the range $-2^{n-1}, \ldots, 2^{n-1} - 1$ which is congruent modulo 2^n to the unsigned integer corresponding to the bit string. This is referred to as the *signed integer* or *two's complement* interpretation of the bit string. Two standard choices for n (for working with integers of reasonable size) are $n = 16$ and $n = 32$; the range of representable signed integers is -32,768 through 32,767 in the first case and -2,147,483,648 through 2,147,463,647 in the second.

This system for representing signed and unsigned integers has the advantage that it is not necessary to have two different addition operations on sequences of bits depending on whether the sequences are to be interpreted as signed or as unsigned integers. The operation corresponding to addition modulo 2^n produces the correct integer sum with either interpretation, provided that the correct result does lie in the set of integers which can be represented. In fact, the operation of integer addition (and similarly subtraction) as performed by computers almost always means addition modulo 2^n; this has the advantage that the operation is never undefined. Note, by contrast, that the operation of *comparing* two bit patterns interpreted as integers (i.e., determining which represents the larger integer) does depend on which interpretation is being used. In the case of multiplication, one usually wants to produce a $2n$-bit product of n-bit integers, and this operation also depends on whether the bit-patterns are interpreted as signed or unsigned integers. Incidentally, one reason for choosing the range for signed integers to be $-2^{n-1}, \ldots, 2^{n-1}-1$ rather than, say, $-2^{n-1}+1, \ldots, 2^{n-1}$ is that the former choice makes it easy to tell whether a signed integer is positive or negative; it is negative if and only if its high order bit is 1.

Representation of characters. The representation of characters by sequences of bits is at least partially standardized (except that IBM uses a non-standard representation on its big computers). The standard is provided by what is known as the ASCII code, which uses seven bits to represent letters (both upper and lower case), digits, punctuation marks, special characters like @, and control characters like tab and carriage return. This code was constructed to represent texts in English; in countries where the language uses more than the twenty-six letters of the English alphabet, bit patterns used in standard ASCII to represent various special symbols are borrowed to represent the extra letters. Thus, although some parts of the ASCII code—e.g., that 65 corresponds to the letter A—are nearly universal, other parts are much less so. In fact, most computers today use eight bits to store each character so it is possible to double the size of the set of representable characters. There seems, unfortunately, to be almost no uniformity in the way the the extra bit patterns are used.

Representation of floating point numbers. This is a much more delicate matter than representing integers and characters; there are many choices to be made, and it is not easy to make the right ones. We will discuss this subject in detail in Sect. 10.

Computer organization. Schematically, a computer can be thought of as made up of three components:

1. Memory, in which information can be stored.
2. A central processing unit (abbreviated CPU) which performs sequences of operations under the control of a program stored in memory.
3. Input-output devices ("peripherals") for communicating with the outside world and for longer-term storage of information.

In the discussion which follows, we will, for simplicity, ignore input-output devices completely.

Although the memory is made up of storage units each of which holds one bit, these units are usually organized into groups of fixed size which we will refer to as *cells*. For example, for the 68000 microprocessor to be described in Sect.

3, memory is organized as a sequence of 8-bit cells. (An 8-tuple of bits is called a *byte*. By what was said above, a byte is the right size to represent a character, and it takes two or four bytes to represent integers of reasonable size.) Each cell in memory is identified by a unique integer, its *address*.

The CPU is the "active" element in the computer. Its function is to perform a certain number of elementary operations, such as copying a piece of information from one place to another, adding two integers, or testing some condition. More precisely: There is a list of such elementary operations which the CPU is capable of performing; this list is called the *instruction set* of the computer in question. Each instruction is represented as a sequence of bits; the coding is usually (but not always) chosen so that it takes some integer number of memory cells to hold the representation for an instruction.* What the CPU does is to read the appropriate number of bits from memory, interpret them as an instruction, and carry out that instruction. To do this, it needs to have available some places of its own for storing information; those places are generally referred to as *registers*. The computers used in practice differ very substantially in

- size of memory cell.
- number of memory cells which can be addressed.
- set of registers available.
- instruction set.

These properties, taken together, define what is called the *architecture* of the computer.

Control flow. Computers normally execute instructions from memory sequentially. However much computers may vary in number and kinds of registers available, every computer needs a register whose function it is to keep track of the address of the next instruction to be executed. This register is known as the *program counter*, often abbreviated as PC.

If we consider the simple situation in which each instruction occupies exactly one memory cell, then a normal instruction execution cycle goes as follows: The CPU interprets the contents of the program counter as the address of a memory cell, reads the contents of that cell, interprets them as an instruction, executes that instruction, and then adds one to the contents of the program counter (so that it contains the address of the memory cell immediately following the one which held the instruction it just executed.) This process is repeated over and over again. It would, however, be very limiting to have to proceed in sequence through a list of instructions, executing each one exactly once. One may want, for example, to execute some sequence of instructions repeatedly (looping), or to test some condition and, depending on the result, do one or another sequence

* The principal exceptions are computers designed primarily for numerical computation. These machines often have large memory cells, typically 64 bits, each of which can hold several instructions.

of instructions. Thus, there is a need for instructions, called *jump* or *branch* instructions, whose execution does not fall under the above scheme but whose function is to change the contents of the program counter, i.e., to alter the flow of control. The applications just mentioned all require conditional jump instructions in which the program counter is changed only if some condition is satisfied, but most computer instruction sets provide unconditional jump instructions as well.

Procedures. The sequence of elementary computer instructions necessary to perform a reasonable scientific computation is enormously long and complicated; the direct construction of a program to perform such a task would demand superhuman effort. The effort required can be made manageable by a process of successive abstraction. One isolates, within the main task to be accomplished, some number of subtasks which can be treated independently and, ideally, each of which needs to be performed repeatedly. Each of the subtasks which is too complicated to be attacked directly is then further subdivided, and so on. A low-level example of such a subtask is the printing of a single character on the terminal screen; a higher level example is finding the eigenvalues of a matrix. The programs for performing the subtasks are referred to as *procedures* or as *subroutines*.

In order to be able to exploit reusable procedures effectively, it is necessary to have available a pair of special jump instructions. The first of these, the *procedure call* instruction, is used to *invoke* a procedure, i.e., to transfer control to the beginning of the procedure while recording in some way where control is transferred from. The second, the *return* instruction, is used when the procedure has completed its task to to transfer control back to the first instruction following the call instruction by which the procedure was invoked.

2. A brief introduction to computer structure

Computers are very complicated devices, capable of accomplishing very complicated tasks. It is nevertheless possible to understand them rather completely by analyzing them on a number of different levels of abstraction, ranging from regarding a computer as a device which functions by exploiting the way electrons move inside semiconductors with inhomogeneous impurities to regarding the computer as a logical device which accepts sequences of instructions written in some high level language such as BASIC and carries them out.

Somewhere in the middle of this scale of increasing abstraction is what is called the *conventional machine level.* On this level, the computer has roughly the appearance described in the preceding section: There is a memory unit, organized as a linearly ordered array of cells; peripherals, or input-output devices, for communicating with the outside world and for relatively long-term storage of larger quantities of information than can be kept in memory; and there is the central processing unit, which reads instructions from memory and executes them.

The conventional machine level of abstraction represents how the computer looks on the lowest level at which the programmer normally has access to it—stripped of the "system software" which greatly facilitates the task of extracting useful work from it. Conventionally, the term *architecture* of a given computer is used to refer to the characteristics of that computer on this level of abstraction, i.e., to its memory organization, register structure, and instruction set.

In this section, I will try to provide a schematic view of what lies *below* the conventional machine level, i.e., of how large numbers of elementary electronic components are assembled to produce devices with the sort of behavior described above, and in the following section I will try to give some perspective on real computer architectures by describing a particular central processing unit, the Motorola 68000, which happens to fit onto a single chip.

The increasing pervasiveness of computers (and of other digital devices: watches, calculators, controllers, ...) derives in large measure from the fact that it is possible to manufacture relatively complicated digital circuits which are both remarkably cheap and remarkably small by what is essentially a multi-stage photographic printing process. Although this manufacturing process has been strikingly successful, it does have limitations. At any given time, the number of electronic devices which can (economically) be put onto a single chip is rather sharply limited by two opposing factors:

1. It is impossible to avoid completely the presence of defects of one sort or another in these chips. These defects may arise from defects in the underlying crystalline structure, from dirt falling onto the surface of the chip during the manufacturing process, ... In a crude approximation, the fraction of chips which are free of debilitating defects varies with the area A of the chip like e^{-A/A_0} for

some constant A_0. It is thus uneconomic to try to manufacture chips bigger than perhaps $5A_0$; the yields will simply be to low. It makes economic sense, and is in fact common practice, to manufacture (expensive) chips which are big enough so that only one in one hundred is good; doubling the area would mean that only one in ten thousand is good and would thus multiply the cost per good chip by one hundred. Currently, A_0 is something like 1/4 cm^2, and slowly increasing with time.

2. On the other hand, the minimum size of individual circuit elements which it is feasible to manufacture is limited by such factors as the difficulty of aligning features produced at different stages of the fabrication process and by the distortions caused by heating and cooling during fabrication. Current standard manufacturing methods permit a "minimum feature size" (not really a well-defined term!) of perhaps 2 microns. (a micron is 10^{-4} cm; for comparison, the wavelength of visible light is about 1/2 micron and the interatomic spacing in crystalline silicon about 10^{-4} microns.) A single transistor requires an area of perhaps 400 square microns, including space necessary to isolate it from adjacent components. In view of the rather sharp cut-off on feasible size, this imposes a similar cut-off on the complexity of what can be put on a single chip: The order of magnitude, currently, is perhaps 100,000 transistors.

Here are some concrete examples of what is feasible: The Motorola 68000 single-chip central processor has roughly 70,000 transistors (This is a large but relatively common chip; it sells, retail, for under \$100). Memory chips with a capacity of 64K bits (64K means $64 \times 2^{10} = 64 \times 1024$), and containing perhaps 100,000 transistors, are extremely common and very cheap (well under \$10). (It is easier to manufacture large chips with a regular structure like memory chips than less regular ones like processor chips.) Memory chips with a capacity of 256K bits are now commercially available but are still relatively expensive. Hewlett-Packard manufactures, for use in expensive desk-top computers, a central processor chip with some 450,000 transistors. By contrast, the Cray-1 central processor contains something like 300,000 chips, each of which contains something like a few hundred transistors, so the prospect of a single-chip processor of the complexity of the Cray-1 is some ways off.

It is a widely accepted rule of thumb ("Moore's Law") that the number of components on the most complex commercially available chips doubles roughly every two years. (From 1959 to 1974, it doubled *every year.*) Most of this increase comes from making individual features smaller, rather than from increasing the size of the chip. Microelectronic circuits are subject at least approximately to scaling laws which say that reducing linear dimensions by a factor of 2 not only increases the number of transistors which can be put on an individual chip by a factor of 4 but also increases the operating speed of these transistors by a factor of 2. These scaling laws break down at feature sizes of the order of 1/10 those currently feasible; one reason is that operating voltage scales linearly with dimension, is currently a few volts, and must be many times the thermal voltage for transistors to work.

The fundamental physical properties of semiconductors can be used in a number of different ways to produce amplifying and switching devices all referred to by the general name of transistor. The kinds of transistor widely used in computers can be grouped broadly into two classes: bipolar junction transistors and field effect transistors. In fact, the field effect transistors used are mostly of a particular kind, called MOS (Metal on Oxide on Semiconductor) transistors, referring to the physical structure of the way they were originally made. Bipolar circuitry is faster than MOS circuitry (by a factor of something like five) but larger (by a factor of the order of ten in area); it also consumes more power and generates more heat. Thus, bipolar circuitry is used in the time-critical parts of computers above the desk-top size, and MOS circuitry in places where low cost per circuit element is more important than speed. (The discussion in the preceding paragraphs of feasible complexity and scaling laws refers to MOS rather than bipolar circuitry.) There is a variant on MOS technology, called CMOS (Complementary MOS) which requires more elementary devices for a given function than standard MOS circuitry, and hence has somewhat lower feasible complexity, but which consumes much less power; CMOS circuitry is used in watches, calculators, and portable computers; its low power consumption and hence low heat generation make it attractive also for large computers.

There are a number of alternative technologies which offer potentially higher speed than bipolar. The two most widely discussed of these are:

1. Replacing silicon as the substrate by gallium arsenide, which has a significantly higher carrier mobility. Gallium arsenide is many times faster than silicon for some particular circuits, but the net gain for realistic circuits (using the same design principles used with silicon) is perhaps a factor of two.
2. Devices based on the Josephson effect can be fifty times faster than standard bipolar devices, but they must be operated a liquid helium temperatures.

From now on, we will restrict ourselves to the discussion of MOS circuitry. An individual MOS field-effect transistor is a device with three terminals called, respectively, *source*, *drain*, and *gate*, which functions schematically as a switch between source and drain which is either open or closed depending on the voltage applied to the gate. Figure 1 gives a rough sketch of what the device looks like physically and indicates the standard symbol used to represent it in circuit diagrams.

It works as follows: A junction between p-type and n-type silicon acts as a rectifier; current can flow from p to n but not the reverse. In the absence of a gate voltage, the resistance between source and drain is very large in both directions, since one or the other of the p-n junctions will prevent current flow. If, however, a sufficiently positive voltage is applied to the gate, it attracts electrons from

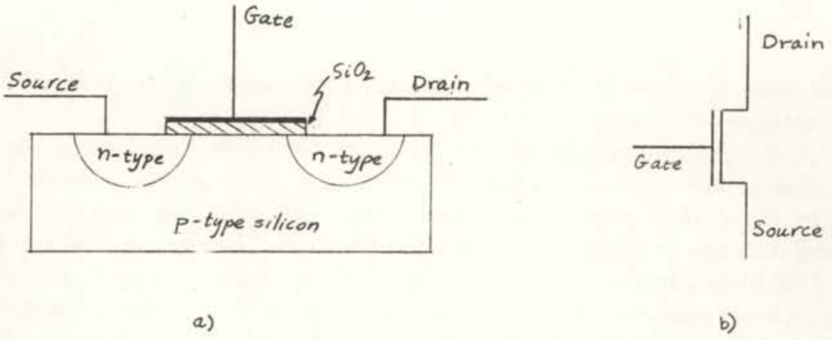

Figure 1.

the substrate, creating an effective n-type layer below the gate which makes a conducting path from source to drain. Thus, the switch between source and drain is open when the gate voltage is zero and closed when it is sufficiently positive. No current needs to flow in the gate circuit when the switch is either open or closed, although changing the state entails charging or discharging an effective capacitor and hence requires a transient current flow.

The simple picture developed in the preceding paragraph is all we will need about the physics of MOS devices; what we now want to do is to see how to organize a very large number—tens of thousands at least—of such switches to make a computer. One of the common ways of representing information in a digital circuit is as voltage levels. Typical values, for MOS circuits, are a range around +5 volts to represent 1 and a range around 0 volts to represent 0. For simplicity, we will refer to these voltage levels simply as 1 and 0 respectively. The first step in organizing transistors into digital devices is construction of some simple building blocks. One kind of building block is a circuit which "computes" a simple Boolean function of its inputs. Some important examples are:

NOT: one input; if input is 1, output is 0 and vice versa.

AND: n inputs; output is 1 if and only if all inputs are 1

OR: n inputs; output is 1 if and only if at least one input is 1

From elementary logic, there are relations among these, e.g.

$$X\, OR\, Y = NOT((NOT\, X) AND (NOT\, Y))$$

Schematically, a NOT circuit (inverter) can be constructed out of a voltage-controlled switch and a resistor as shown in Figure 2.

The input is connected to the gate of the switch. When the input is 1, the switch is closed, and the output is grounded through the switch; hence, is 0. On the other hand, when the input is 0, the switch is open, and the output is connected to the supply voltage through the resistor. Although we

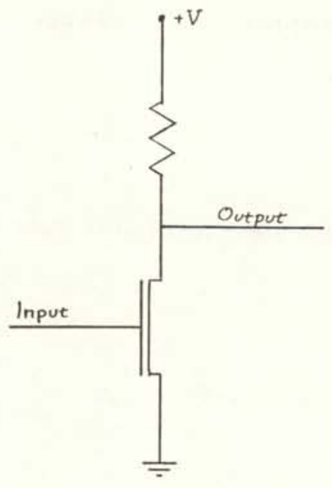

Figure 2.

have described the circuit as containing a resistor, in practice the resistor is emulated by another transistor.

AND and OR circuits are not quite so simple, but a NOT AND (NAND) circuit can be constructed as shown in Figure 3.

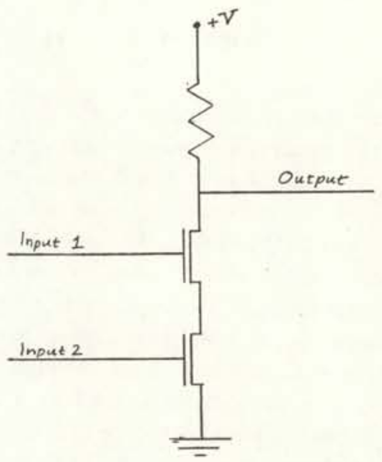

Figure 3.

The output will be connected to ground if and only if both inputs are 1. A NOR (NOT OR) circuit can be made in a similar way, with the transistors connected in parallel instead of in series.

Already at this point it is possible to see the fundamental mechanism which determines the operating speed of MOS circuitry. Consider a pair of NOT circuits, with the output of one connected to the input of the other as shown in Figure 4.

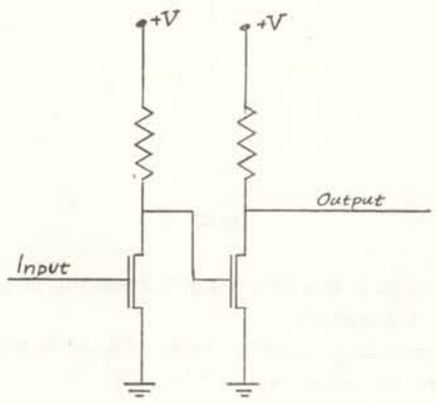

Figure 4.

Suppose that, at some instant, the input to the first inverter changes from, say, 0 to 1. Just before the change, the wire connecting the output of the first inverter to the input of the second is in the 1 state, which means that there is some charge on the gate capacitance of the second transistor. When the input to the first inverter has changed to 1, the gate of the second transistor is grounded through the first, and it will proceed to discharge. This takes some time because the resistance is finite. The output of the first inverter thus cannot change to reflect the changed input until this gate capacitance has sufficiently discharged, and there is hence a delay in passing the signal through the inverter. This delay will depend on how the output is loaded; if the output of the first inverter were instead connected in parallel to the inputs of three other inverters, the delay would be roughly three times as large. Thus, specifying the delay time without saying how the circuit is loaded doesn't mean very much, but the order of magnitude, for current standard MOS technology, is a few nanoseconds.

It is easy to see that, out of NOT circuits and any one of AND, OR, NAND, or NOR, one can construct a circuit to compute any given Boolean function (of any number of "variables", i.e., inputs). As an example of such a higher-order logic circuit, consider a *full adder*. A full adder is a circuit with three inputs and two outputs. Its function is as follows: Interpret each of the three inputs as integer (0 or 1). The sum of the inputs is thus a number in the range 0-3, so it can be represented in binary form with two digits. The outputs corresponding to a given set of inputs are to be exactly the two-digit representation of this sum.

Before describing how to construct such a circuit, we note that n of them can be connected together as shown in Figure 5 to make an n-bit parallel adder: The three inputs to the j-th full adder are the j-th bits of the two summand inputs and the high-order bit output of the $j-1$-st full adder (i.e., the carry). The circuit produces an $n+1$-bit sum.

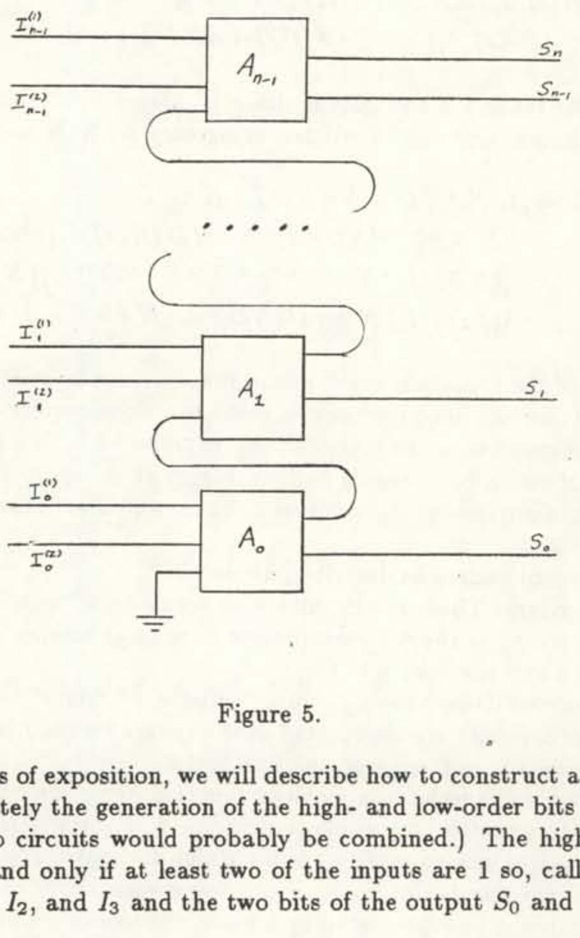

Figure 5.

For purposes of exposition, we will describe how to construct a full adder by analyzing separately the generation of the high- and low-order bits (although, in practice, the two circuits would probably be combined.) The high-order bit of the sum is 1 if and only if at least two of the inputs are 1 so, calling the three input signals I_1, I_2, and I_3 and the two bits of the output S_0 and S_1, we have

$$S_1 = (I_1\ AND\ I_2)\ OR\ (I_1\ AND\ I_3)\ OR\ (I_2\ AND\ I_3)$$

Since NAND circuits are easier to make than AND/OR circuits, it is useful to note that this can also be written as

$$S_1 = (I_1\ NAND\ I_2)\ NAND\ (I_1\ NAND\ I_3)\ NAND\ (I_2\ NAND\ I_3)$$

(Written in this way, the formula is ambiguous; what we mean by the two NANDs outside parentheses is a single three-input NAND.)

The low order bit is just slightly more complicated:

$$\begin{aligned}S_0 = {} & (I_1\ AND\ I_2\ AND\ I_3)\ OR\ (I_1\ AND\ (NOT I_2)\ AND\ (NOT\ I_3))\ OR \\ & ((NOT\ I_1)\ AND\ I_2\ AND\ (NOT\ I_3)))\ OR \\ & ((NOT\ I_1)\ AND\ (NOT\ I_2)\ AND\ I_3)\end{aligned}$$

(The low-order bit is 1 if and only if either all three inputs are 1 or exactly one of them is.) Again, this can be written using only NANDs and NOTs:

$$\begin{aligned}S_0 = {} & (I_1\ NAND\ I_2\ NAND\ I_3)\ NAND \\ & (I_1\ NAND(NOT\ I_2)\ NAND\ (NOT\ I_3))\ NAND \\ & ((NOT\ I_1)\ NAND\ I_2\ NAND\ (NOT\ I_3))\ NAND \\ & ((NOT\ I_1)\ NAND\ (NOT\ I_2)\ NAND\ I_3))\end{aligned}$$

In spite of the fact than any Boolean function can in principle be evaluated by circuits of the sort described above, some functions require circuitry which is complicated enough to be impractical. An example of such a function is the 2n bit product of two n-bit integers (with n bigger than something like 8). There are, however, many useful higher-level circuits which are feasible to construct. Among there are:

- n bit parallel adders as described above.
- n bit decoders: These are circuits with n inputs, 2^n outputs. The n inputs are interpreted as the binary representation of an integer j; the j-th output is set to 1 and the others to 0.
- multiplexers: These have n "control" inputs, 2^n "data" inputs, one output. The control inputs are interpreted as the binary representation of an integer j; the output is put equal to the j-th input.

In the above examples, the Boolean function has some regularity (as manifested in the fact that it is possible to describe it in a few words). There is also a perfectly general way to make a circuit which computes any Boolean function provided that the number of inputs is not too great. The idea is that a Boolean function can always be represented by a table and one can simply store the table in read-only memory. A read-only memory consists essentially of a multiplexer with each of its data inputs permanently set either to 0 or to 1. Such a circuit has the further advantage that the complicated part, the multiplexer, is independent of the function being realized, and it is therefore possible to manufacture general read-only memories which can then be programmed with the desired table.

The kinds of circuits we have been discussing all share the characteristic of being without memory; they are idealized as functioning instantaneously (although the delay between the time the inputs are changed and the time a corresponding change of outputs takes place is of great practical importance). Circuits of this type are called *combinational logic*. Memory circuits, i.e., those whose function is to store information, form a second essential class of digital circuits. There are a number of possible organizations of the elementary building blocks, depending on what the protocol is for changing the contents of the memory. A common example is a device called a *latch*, which functions as follows: It has two inputs, a data input and a control input, and one output. When the control input is 1 (say), the data output is equal to the data input, but when the control input is 0 the data output stays fixed. Thus, when the control input is 0, the latch remembers, and indicates on its output, the state of the data input at the time the control made its most recent 1-to-0 transition.

We turn now to a discussion of a higher level of organization of digital circuits. In thinking about this level, it is useful to introduce the notion of a *finite state machine*. Informally, a finite-state machine is an abstract object with a finite set of possible instantaneous states and a finite set of possible inputs. It undergoes a discrete deterministic time evolution; the state at time $n+1$ is a function of the state at time n and the inputs at time n. It also has a further piece of structure: a finite set of possible outputs, the output at time n being a function of the state and inputs at that time. (More formally, a finite-state machine can be defined as a quintuple: three finite sets and two functions such that ...) Since everything is finite, the next-state function and output function can be specified simply by giving tables. There are many practical devices whose behavior can be usefully modelled as that of a finite-state machine with a reasonable number of states; a traffic-light controller is a good example.

It is a simple matter, in principle at least, to build (out of the sort of building blocks we have been discussing) an electronic device which realizes any given finite state machine. The idea is as follows: Identify the set of states with $\{0,1\}^N$, so that states can be stored in an N-bit memory, and identify the possible inputs (respectively, outputs) with a subset of $\{0,1\}^J$ (respectively, $\{0,1\}^K$), so inputs (respectively, outputs) can be transmitted on J (respectively K) wires. Then make a circuit as shown in Fig. 6.

The box labelled "Memory" is an N-bit memory; it records the current state. The combinational logic circuit has as inputs the contents of the memory (i.e., the current state) and the inputs to the machine; it produces as outputs the next state and outputs corresponding to the current-state, current-inputs pair. In other words: it implements the next-state and output functions for the finite machine. The memory has the following behavior: It has a control

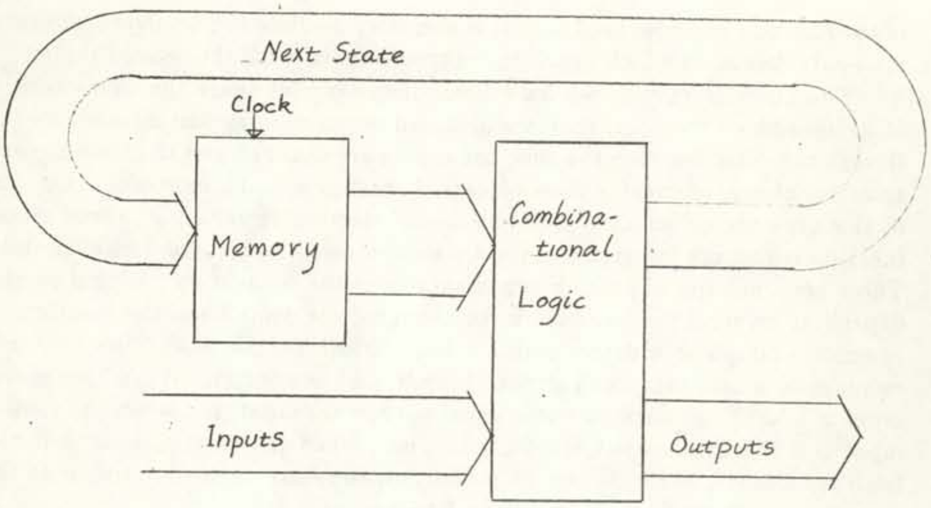

Figure 6.

input, the *clock* signal. It ignores its data inputs (i.e., the next-state output of the combinational logic) except when the clock signal makes a 0-to-1 transition (say). It records whatever its inputs are at the time of such a transition, but does not change its outputs to correspond until the subsequent 1-to-0 transition of the clock signal. The machine can then be "driven" simply by making the clock signal a step function taking alternately values of 0 and 1. Thus, suppose that, at the n-th 1-to-0 clock transition, the contents of the memory correspond to the n-th state. At this point, the memory outputs change so they also represent the n-th state. The input signal are also supposed to be synchronized so that, during the subsequent time interval when the clock signal is 0, the inputs are stable and correspond to the n-th input. During this 0 interval, the outputs of the combinational logic settle to the values corresponding to their current inputs, i.e., they settle to the n-th output and $n+$ 1-st state. In the subsequent 0-to-1 transition, the $n+$ 1-st state is read into the memory, and the n-th output is available. On the following 1-to-0 transition, the cycle starts over again, so the machine advances one time step per clock cycle.

The only practical problem to be solved, then, in implementing a given finite state machine, is designing the combinational circuit which computes the next-state and output function. For a general machine, doing this in a straightforward way starting from elementary logic circuits may be difficult. If the number of states and inputs is not too large, however, it can, as noted above, be constructed in a completely straightforward way by storing the next-state and output tables in read-only memory. While this approach is unlikely to give the fastest or smallest circuit accomplishing the desired task, its conceptual simplicity and the fact that it uses only standard integrated circuits often outweigh these disadvantages. (There is another approach, using a family of devices called programmable logic arrays, which is intermediate both in generality and efficiency between the elementary logic circuit approach and the read-only memory approach.)

Although a complete computer can, in principle, be regarded as a finite-state machine, it has an enormous number of possible states, making the above view ineffective. It is much better to view the central processing unit of a computer as a relatively passive set of data storage and manipulation circuitry "steered" by a comparatively simple finite-state machine. In this perspective, the CPU can be thought of as divided into three components: a set of *registers,* a circuit called a *data path/arithmetic-logical unit* (abbreviated ALU), and a *control unit.* We will discuss these three components in turn.

The term *register* is used here in the general sense of a local storage unit in the CPU. The registers include those available to the programmer, the program counter, the condition codes register (which records some information about the results of preceding operations; see Sect. 3), but also some special registers which are totally invisible to the programmer. Among these latter, there are generally two called respectively the memory address register (MAR) and memory data register (MDR) which play a special role in transmitting information between the CPU and memory. That process works as follows: There are control signals generated by the CPU requesting reads from or writes to memory. In the case of a memory read operation, the CPU puts the address of the memory cell to be read in the MAR and signals a memory read request. External memory control circuitry then takes over, reads the address from the MAR, fetches the contents of the desired memory cell, and copies it into the MDR, from which the CPU can retrieve it. Similarly, to store something in memory, the CPU puts the datum to be stored into the MDR, the address at which it is to be stored into the MAR, and signals a memory write request; the external memory control circuitry does the rest.

The data-path/arithmetic-logical unit is a complicated combinational circuit which takes input from the registers, manipulates it, and returns results to the registers. It is represented schematically in Figure 7.

The indicated control signals are all outputs of the control unit which will be discussed shortly. In the middle is the arithmetic-logical unit proper. It has two n-bit inputs on which it is capable of performing a number of simple operations, including, typically: forming the n-bit sum, the n-bit arithmetic difference, the bitwise logical OR and AND; shifting one of its inputs left or right one place; or just passing one of its inputs through unchanged. Which of these functions it performs is determined by its control signals. The input selector routes the

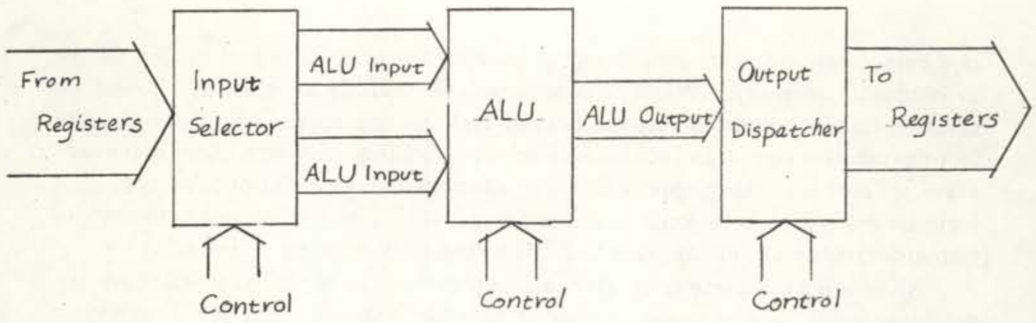

Figure 7.

contents of two of the registers—which pair is specified by the control signals—to the inputs of the ALU proper, and the output dispatcher directs the output of the ALU proper to one of the registers. The functioning of all this circuitry is controlled by a clock signal; the control signals are established near the beginning of a given clock cycle, and the result is returned to the desired register near its end.

We will now illustrate how, by issuing appropriate sequences of control signals, the control unit can make the ALU execute typical computer instructions. Let us first note one important and particularly simple ALU operation: copying the contents of one register into another. This is accomplished by setting the control signals to the ALU proper so that it simply passes one of its inputs through to its output, the control signals to the input selector so it passes the contents of the desired source register to the appropriate input of the ALU, and control signals to the output dispatcher so that the output of the ALU is transmitted to the desired destination register. We also note that execution of all instructions start in the same way, with the fetching of the instruction itself from memory. To do this, a first ALU cycle is used to copy the contents of the program counter (which, in our present perspective, is just one of the registers) to the MAR. A memory read request is signalled by the control unit to the memory controller at the end of this cycle. Normally, it takes some time for the result of the memory read to be available in the MDR. Rather than doing nothing while waiting for the requested information to be made available, the CPU does another ALU cycle to add 1 to the program counter so that it points to the next instruction. This is done by setting the input selector to pass to the ALU the contents of the program counter and the constant 1; the ALU to form the sum of its inputs; and the output dispatcher to direct the output of the ALU back to the program counter. If we suppose that one ALU cycle allows the memory circuitry enough time to put the desired datum into the MDR, the next step is for the control unit to interpret the contents of the MDR as an instruction.

What happens next depends on what the instruction is. We will describe two representative examples. Consider first an instruction to add the contents of one register to another. Executing this instruction is very simple, and takes only one ALU cycle: The input selector is set to select the source and destination registers as inputs to the ALU; the ALU is set to add its inputs; and the output dispatcher is set to send the output of the ALU to the destination register.

As a second slightly more complicated example, consider an instruction to copy a memory cell into a register, where the address of the cell to be copied is contained in the memory cell immediately following that which holds the instruction itself (or, equivalently, the complete instruction can be regarded as occupying two successive memory cells, the second being used to hold the address of the operand). In this case, the first few steps repeat the sequence of operations done in fetching the instruction: the program counter is copied to the MAR; a memory read request is signalled; and 1 is added to the program counter. When the memory read is completed, the next step is to copy the MDR (which now contains the *address* of the cell to be copied into a register) to the MAR, and again to request a memory read. When *this* memory read is completed, one more ALU cycle is needed to copy the MDR into the desired destination register. Notice that, although we have described this instruction as one to load an operand into a register, exactly the same sequence of steps could be used to load the contents of some memory cell into the program counter, i.e., to execute an indirect unconditional jump instruction.

These two examples serve to illustrate that an ALU with a reasonable set of capabilities can be made to execute almost any desired instruction. All that is necessary is to generate the appropriate control signals. As already noted, this is done by a part of the CPU called the control unit. The control unit can be a reasonably simple finite state machine, which takes as inputs the contents of the MDR (so that it can read instructions) and a few other signals from the data paths/ALU (for use in determining whether or not a conditional branch is to be taken). In real computers designed in this way, the number of states required is of the order of a few hundred to a few thousand; the number of control signals to be generated is of the order of one hundred. Thus, the control unit can quite reasonably be constructed by storing the next-state and output tables in read-only memory.

This brings us to the important notion of *microprogramming*. If the control unit is implemented in read-only memory, it is natural to think of it as a simple sub-computer which takes as input the computer instructions and generates, under the control of its program, stored in read-only memory, sequences of ALU operations to carry out those instruction. This view of computer organization leads to a number of conclusions of practical importance. One of these is that the problem of designing the hardware for a computer, including the control unit, can to a large extent be separated from the problem of designing and implementing its instruction set; the latter is a matter of programming the control unit sub-computer. A given instruction set can be implemented in hardware differing radically in speed (and hence in cost). Conversely, there is no reason why the memory holding the sub-computer program has to be read-only. If it is instead read-write memory, the instruction set executed by the computer can be changed completely, without any changes in the hardware, just by changing the contents of this memory. In this case, one speaks of a computer with *writable control store* and, in general, the program for the sub-computer is referred to as *microcode.*

Both of these ideas were used very effectively by IBM in the 360 series of computers, which were introduced in the late 1960's. On the one hand, although the computers in this series differed by a factor of more than twenty in speed and price, they all had exactly the same basic instruction set so a program working on one of them could be run without changes on any other and give identical results. On the other hand, some computers in this series had writable control store and microcode was provided which permitted them to execute the instruction sets of earlier IBM computers, making it possible to run old programs without modification.

In more recent times, most computers in a broad middle range of size and capability are designed using the microprogramming idea. Those which are not are either the very largest and fastest computers (because it is possible to obtain higher operating speeds with other organizations) or very small, such as the so-called "8 bit" single chip CPU's (because they can be constructed with fewer circuit elements using different design methods.)

3. Architecture of the 68000

As we have already said, the *architecture* of a computer refers to how the computer appears to the programmer at the lowest level, i.e., to its memory organization, register structure, and instruction set. To give some feel for this subject, we will describe, in reasonable detail but without always mentioning *all* the complexities, the architecture of the Motorola 68000 microprocessor. We first, however, make some remarks about instruction sets in general. Computer instructions can be grouped, loosely, into four categories:

1. Data movement instructions. The term "movement", although traditional, is a misnomer. What these instruction actually do is to *copy* information from one place to another; the source datum is left unchanged.
2. Data manipulation instructions. These include arithmetic operations, bitwise logical operations, and shifting.
3. Program control instructions. These include conditional and unconditional jump instructions; also procedure call and return instructions.
4. System control instructions. There are instructions which permit the computer to control peripheral devices, and also which permit programs to control other programs. They are essential for many things, and notably for constructing operating systems, but they are also usually the least comprehensible part of the instruction set. In this survey we will ignore them completely.

There is a tendency to think of data manipulation instructions as the essence of a computer's instruction set, but this is not where the main differences between computers lie. Except for instructions for floating point arithmetic, most modern computers provide very similar sets of data manipulation instructions. Differences appear more in such matters as how flexibly the operands for various operations can be specified; in what conditions can be tested by the conditional jump instructions; in how many registers are available and on what constraints there are on their use; and in how convenient a set of facilities there is for calling procedures and passing arguments to them.

We turn now to the 68000. This is a powerful single-chip CPU, used in a number of desktop computers and notably in the Apple Macintosh. Its architecture is broadly similar to that of the VAX* except that the 68000 lacks registers and instructions for operating on floating point numbers. (In fact, the architecture of the 68000 is designed so that floating point instructions can be added in a way which fits harmoniously with the rest of the instruction set when it becomes feasible to provide the capability to execute them.)

The 68000 operates on information in three different-sized pieces:

- byte: a group of 8 bits.

* A widely used "superminicomputer" manufactured by Digital Equipment Corporation.

- word: 2 bytes or 16 bits.
- long word: 4 bytes or 32 bits

(Warning: This terminology varies from manufacturer to manufacturer. The above terminology is that used by Motorola.) Most data movement and data manipulation instructions have three versions, one acting on bytes, one acting on words, and one acting on long words. The 68000's memory is treated as a linear array of individually addressable bytes; words and long words are represented as two and four successive bytes respectively. By convention, the address of a word or long word is the address of its first (lowest-numbered) byte, and, for reasons of efficiency, is required to be even. Addresses are 24 bits wide, providing the capacity to address a maximum memory of $2^{24} = 2^4 \times 2^{10} \times 2^{10} = 16 \times (1024)^2$ bytes (16 *megabytes*).

The 68000 has sixteen "general purpose" registers, in addition to its program counter. The general purpose registers are each 32 bits wide, and so can hold long words as well as bytes and words. (It is for this reason that the 68000 is often spoken of as a 32 bit computer.) The general purpose registers are divided into two groups of eight, called respectively *address* and *data* registers, and denoted A0–A7 and D0–D7. These names should not be interpreted as implying absolute restrictions on the use of these registers; either kind of register can be used to hold either addresses or data. However,

- the contents of an address register can be taken "automatically" in certain contexts as the address of an operand in memory, while data registers cannot be used in this way.
- a wide range of data manipulation instructions can be performed on the contents of data registers, while the operations which can act on the contents of address registers are limited—essentially—to addition and subtraction.

It is an important part of the design philosophy of the 68000 to treat the registers within each of these two sets in as symmetric a way as possible. The eight data registers are rigorously equivalent, as are the first seven address registers. The eighth address register has has all the capabilities of the first seven, but is distinguished from them by a special role in the procedure call and return instructions.

The 68000 has an unusually flexible set of data movement instructions. It is possible, with a single instruction, to move a byte, word, or long word from one register to another, from a register to a memory location, from a memory location to a register, or from one memory location to another. About the only limitation on these instructions is that single byte move operations cannot have an address register as source or destination.* As already noted, one very important element in a computer's power and convenience of use is

* This is another general principle of the 68000 architecture: "Address registers do not support byte operations".

the flexibility with which operands can be specified, i.e., the range of *addressing modes* available. The 68000 has a rich set of addressing modes, which can be used, in particular, to specify the source and destination for data movement instructions. It is conventional to include among addressing modes the case in which the operand is in a register; this is known as *register direct mode.* For all the other addressing modes, the operand is in memory; these addressing modes are thus all prescriptions for specifying a 24-bit quantity to be taken as the address of the operand. Although this has no fundamental significance, it may be useful for purposes of orientation to distinguish between a set of particularly simple and important basic addressing modes and the rest. The basic addressing modes on the 68000 are as follows:

- Register direct.
- *Absolute addressing.* An explicit address of the operand in memory is included in the instruction. On the 68000, there are two options: The address may be specified either as a long word, in which case the low-order 24 bits are taken as the actual address, or as a word, in which case the address is obtained by prepending eight 0's if the high-order bit is 0 or eight 1's if the high-order bit is 1, i.e., the address is taken to refer to either the lowest or highest 32K bytes of memory.
- *Address register indirect.* The operand is in a memory location whose address is specified as the contents of an address register.
- *Immediate.* The *operand* is included in the instruction. (Don't confuse this with absolute addressing, where the *address* of the operand is included in the instruction.)

Among the variants on these basic addressing modes provided on the 68000 are:

- *Address register indirect with displacement.* The instruction specifies both an address register and a 16-bit displacement. The address of the operand in memory is constructed by adding the 16 bit displacement, regarded as a signed integer, to the contents of the specified address register (and ignoring the high-order 8 bits, since this will produce a 32 bit quantity and addresses have only 24 bits.)
- *Address register indirect with postincrement.* As with address register indirect, the instruction specifies an address register; the contents of the address register are taken as the address of the operand in memory. After the address has been used, however, the contents of the specified address register is increased by 1, 2 or 4, depending on whether the operation in question was a byte, word, or long word operation. This addressing mode is useful for stepping through a list of objects stored in successive memory location.
- *Address register indirect with predecrement.* Like the above except that the contents of the address register is decreased (rather than increased) by 1, 2 or 4, and the change is made before (rather than after) the number in the address register is taken as the address of the desired operand.

- *Program counter relative with displacement.* This is like address register indirect with displacement, except that the contents of the program counter is used instead of the contents of one of the address registers.

The 68000 also provides another, more elaborate, addressing mode, where the instruction specifies not one but two registers (the first of which must be an address register but the second may be either an address or a data register) and an eight-bit "displacement"; the address of the operand in memory is generated by adding together the contents of the two specified registers and then adding on the displacement, regarded as an eight-bit signed integer. Finally, there is an analogous addressing mode in which the program counter is substituted for the first register in the above scheme.

It is almost, but not quite, true that the source and destination of a data movement instruction can be specified by an arbitrary pair of addressing modes. One restriction has already been mentioned: A single-byte move cannot have an address register as either source or destination. The only other restriction is that the destination cannot be specified by any of the program counter relative addressing modes. This latter restriction applies also to specifying the destination of data manipulation instructions; it is imposed in support of a general principle of sound program design which forbids modifying, during program execution, the regions of memory used to store the program (as opposed to the regions used to hold data).

We turn now to data manipulation instructions. These can be grouped into two-operand and one-operand instructions. The two-operand instructions are: addition, subtraction, multiplication, division, bitwise logical AND, bitwise logical OR, and bitwise exclusive OR. For each of these instructions, the result replaces one of the operands, so we can refer to the two operands as source and destination (in spite of the fact that the destination also serves as a source ...) The discussion for the remainder of this paragraph applies to all these two-operand instructions *except* multiplication and division, which don't fit into the general pattern. Each of these operations can be done on bytes, on words, or on long words. As already indicated in Sect. 1, the operations of addition and subtraction are done modulo 2^n, where n is 8 for byte operations, 16 for word operations, and 32 for long word operations. There are also variants on the addition and subtraction operations which take into account carries from prior operations; these are useful in performing multiple-precision arithmetic. The rules restricting the specifying of operands are as follows: It is required, first of all, that at least one of the operands be in a data register.* If the destination is a data register, the source may be either a data register or in memory; in the latter case, it can be specified using any of the memory addressing modes.

* Just to complicate things: For the exclusive OR operation (only), the source must be a data register

If the destination is not a data register, it must be in memory and can be specified using any of the memory addressing modes except the program counter relative ones. There are also special operations which add to or subtract from the contents of address registers; there are also some limited—but useful—addition and subtraction instructions with both operands in memory.

As indicated, the above discussion does not apply to the multiplication and division instructions. The multiplication instruction takes two 16-bit operands and produces a 32-bit product. (More precisely: there are two multiplication instructions; one computes the product interpreting the operands as unsigned integers; the other as signed integers. See Sect. 1.) The destination must be a data register; the source may be either a data register or in memory; in the latter case, its address can be specified using any memory addressing mode. The division instruction takes a 32-bit numerator and a 16-bit denominator and produces a 16-bit quotient and a 16-bit remainder. (Again, there are two versions, one for signed integers, the other for unsigned.) The destination (which also holds the numerator) must be a data register; the source (denominator) may be either a data register or in memory, with all memory addressing modes allowed in the latter case. In contrast to all the other instructions considered so far, the division instruction cannot always be carried out; this is the case, for example, when the denominator is zero. The question of what the 68000 does when an invalid division instruction is issued is an important and complicated one, which we will not follow up except to say that it initiates a change in the flow of control which can be used to invoke error-handling procedures.

The one-operand data-manipulation instructions provided by the 68000 are: clear(set to zero), negate (reverse sign), logical complement (reverse each bit), and a variety of shift and rotate instructions. Again, there are variants to act on bytes, words, and long words. The result of the operation replaces the operand, so only one location needs to be specified. This location may be either a data register or in memory; in the latter case, any memory addressing mode except the program counter relative ones can be used to specify it.

The 68000's unconditional jump instruction gives considerable flexibility in how the destination of the jump is specified; most of the ways in which the address in memory of a source operand for a data movement instruction are allowed. (The exceptions are that the autoincrementing and autodecrementing addressing modes, and immediate addressing, are forbidden.) The conditional jump instructions are less flexible; the destination address can be specified only by giving an 8-bit or 16-bit unsigned integer displacement from the location of the conditional jump instruction itself. (More precisely: from the memory location immediately following the conditional jump instruction.) Thus, a conditional jump instruction cannot change the program counter by more than 2^{15}, i.e., 32,768. This is enough for most purposes; in any event, a jump to an arbitrary memory location can be made in two steps by using a short conditional jump to get to an unconditional jump instruction with the desired destination.

The conditional jump instructions work as follows: There is a register in the CPU which we have not yet talked about, called the *condition codes* register, whose function is, roughly, to remember some information about the result of the most recent data movement or data manipulation instruction; the conditional jump instructions test the current state of the condition codes register to determine whether the jump is or is not to be taken. To be slightly more precise: The condition codes register can be thought of as holding five independent bits called N (negative), Z (zero), C (carry), V (overflow), and X (extend). For simplicity, we will ignore the last two. Each time an instruction is executed, some subset (depending on the instruction and possibly empty) of these bits is updated to reflect the results of that operation. For example: an integer addition instruction sets the Z bit to 1 or 0 according as the result of the addition is or is not zero (modulo 2^n); it sets the N bit to 1 or 0 according as the result, regarded as a signed integer, is or is not negative, (i.e., the N bit is made equal to the high-order bit of the result); and it sets the C bit to 1 or 0 according as, when the operation is regarded as an addition of unsigned integers, there is or is not a carry generated out of the high order bit (i.e., the returned result is the ordinary sum minus 2^n). A data movement instruction updates the Z and N bits according to value of the datum being moved; it always sets the C bit to 0. A jump instruction, conditional or unconditional, on the other hand, doesn't affect any of the condition codes. There is also a special comparison instruction, which takes two operands and which has the same effect on the condition codes as a subtraction instruction with the same operands, but which does not actually produce the result of the subtraction, leaving the operands unchanged. As indicated, the conditional jump instructions test the status of the condition codes to determine whether the jump is to be taken. There are quite a number of these instructions, including jump on minus (the jump is taken if and only if the N bit is 1), jump on plus (jump iff N bit is 0), jump on equal (jump iff Z bit is 1; presumably to be used after a subtraction or comparison instruction), jump on carry set (jump iff C bit is 1) ...

As we noted in Sect. 1, there is a need for special instructions for invoking procedures and returning on completion. In the 68000, the procedure call instruction works as follows:

1. four is subtracted from the contents of address register A7
2. the current contents of the program counter (at this point, the address of the first instruction following the procedure call instruction) is copied into the long word of memory whose address is the new contents of A7.
3. the address of the desired procedure (i.e., of its first instruction) is loaded into the program counter so that the next instruction to be executed will be the first instruction of the procedure.

Thus, on completion of the procedure call instruction, control has been transferred to the beginning of the procedure, the return address is recorded in memory, and the address of the place where it is recorded is in register A7. We say that A7 *points to* the long word in memory holding the return address.

The return instruction can now be very simple: When it is executed,

1. the long word pointed to by the register A7 is copied into the program counter.
2. four is added to the contents of A7.

This scheme for procedure calls and returns is an instance of what is called the *stack* organization of memory, with A7 playing the role of the *stack pointer.* This organization provides a simple and flexible way of keeping track of more or less arbitrary sequences of procedure calls and, more generally, sharing working space among procedures. To use it, it is arranged that, at the beginning of program execution, the stack pointer points to the top of some adequately large region of memory. Each procedure must then follow a few simple rules:

- It is free to use as working space the memory extending a reasonable distance below where the stack pointer was pointing when it began execution, but it must not modify the memory above that point.
- It must arrange that the stack pointer is restored to its initial value before a return instruction is issued.
- It must ensure that, whenever it in turn issues a procedure call, the value of the stack pointer is no larger than it was when it began execution.

As long as all procedures observe these rules, and as long as the amount of memory reserved as a stack is large enough, an arbitrary chain of procedure calls, including ones in which a procedure calls itself (recursion), will be managed, automatically, in an orderly way.

This completes our (incomplete) survey of the instruction set of the 68000. It may be informative at the point to look at a small sample of a 68000 program, expressed in assembly language. In assembly language, the programmer specifies exactly the sequence of computer instructions to be executed, but can specify them in the form of mnemonics rather than as an explicit sequence of 0's and 1's. The (slightly artificial) example we are going to discuss is that of computing

$$\sum_{i=0}^{n-1} a_i \times b_i,$$

where the a_i and b_i are one-word integers stored in successive memory locations beginning at 1000 and 2000 respectively; where the number n of terms to be summed is also a one-word integer, stored in memory locations 998-999; and where the calculation is to be done modulo 2^{16}. The program is as follows:

```
CLR        D2         1.
MOVE.W     998,D0     2.
```

```
       MOVE.L   #1000,A1    3.
       MOVE.L   #2000,A2    4.
LOOP:  MOVE.W   (A1)+,D1    5.
       MUL      (A2)+,D1    6.
       ADD.W    D1,D2       7.
       SUB.W    #1,D0       8.
       BNZ      LOOP        9.
```

The line numbers on the right are not part of the program; they are just there to identify the lines in the following commentary.

1. This sets register D2 (which is going to be used to accumulate the sum) to zero.
2. This moves the contents of memory locations 998 and 999 (i.e., the number of terms to add up) into the low-order half of register D0. The fact that it is a word, rather than a byte or long word, operation is indicated by the suffix .W on the instruction mnemonic.
3. This moves the number 1000, (i.e., the address of a_0) into register A1. Note the difference between the 998 in the preceding instruction (which was interpreted as the memory address of an operand) and the 1000 (preceded by #) which is itself an operand. Note also the suffix .L; we want to treat the address 1000 as a long word (32 bit) quantity.
5. The "LOOP:" on the left is a *label* to be used as the destination of a jump instruction. This is the first instruction of a sequence that will be executed n times. Each time this instruction is executed, two things happen: The word to which A2 points is copied into register D2, and A2 is increased to point to the subsequent word in memory (i.e., we are using here the addressing mode called address register indirect with postincrement).
6. The contents of D1 are multiplied by the word in memory pointed to by A2 and A2 is advanced to point to the subsequent word. This line has been simplified slightly; there is no instruction MUL but two different multiplication instructions called MULU (unsigned multiplication) and MULS (signed multiplication). We could use either here, since we are only interested in the low-order 16 bits of the product, and these don't depend on which kind of multiplication is done.
7. The low-order word of D1 is added to D2.
8. The counter in D0 is decremented by one.
9. This is a conditional jump instruction; the mnemonic is an acronym for Branch on Not Zero. A jump is taken if the Z bit in the condition codes register is not 1 i.e., if the counter in D0 has not yet reached zero; otherwise control passes to the subsequent instruction. Note how the destination of the jump is specified in assembly language via a label, although the destination of a conditional jump instruction on the 68000 must actually be specified by giving a displacement from the location of the instruction itself. The

program which translates the sequence of mnemonics into the corresponding sequence of computer instructions also computes the appropriate offset and inserts it into the generated instruction, thus sparing the programmer a tedious and error-prone calculation.

We will conclude our tour of the 68000 architecture with a discussion of the speed with which it executes instructions. One element which enters in a trivial way into the determination of the speed is the clock frequency. The timing of the 68000 chip is controlled by an external clock signal whose frequency is, within limits, arbitrary. Any particular chip has a maximum clock frequency at which it will function reliably; chips are sold with various rated maximum frequencies, the most common being 8 and 10 MHz. We will give execution times in clock cycles.

The second thing to understand about the rate of execution of instructions of the 68000 is that it is largely limited by the speed of memory access. Each memory operation transmits only one sixteen-bit word, and takes four clock cycles, assuming that the memory is fast enough. (If the memory cannot respond quickly enough, the CPU may have to wait some number of additional cycles, called "wait states", for the memory operation to be completed. The execution times we give will be those which apply in the absence of wait states.) Since each 68000 instruction is at least one word long, and since the instruction must be read from memory before it can be executed, no instruction can be executed in less than four clock cycles.

Here are the times required for a few representative instructions:

ADD.W	D1,D0	4 cycles
ADD.W	(A1),D0	8 cycles
ADD.L	(A1),D0	14 cycles
ADD.L	123456,D0	22 cycles
JMP	123456	12 cycles
MULU	D0,D1	70 cycles(maximum)

In the first two examples, the execution time is as small as possible consistent with the required memory accesses. The third example requires three memory accesses (one to read the instruction, two more to read the two words of the long word summand); an additional two cycles are apparently needed to complete the addition. The fourth example is similar, except that five memory accesses are required (three to read the instruction, which contains the 32-bit address 123456, then two more to read the summand). The jump instruction, like the first two examples, takes the minimum time consistent with the required memory accesses. The unsigned multiplication instruction, on the other hand, requires only one memory access (to read the instruction itself), but may take up to 70 cycles to execute. About the only instructions besides multiplication whose execution times are much larger than the times for the required memory accesses are division and multi-bit shift operation. (The latter require $2m$ clock cycles to shift m positions.)

To round out the picture, we mention one empirical timing for a complicated operation: A double precision floating point multiplication (about sixteen decimal digits of precision; see Sect. 10), done in software on a 10 MHz 68000 takes about .2 milliseconds.

4. Preliminary remarks on numerical analysis.

The purpose of this section is to make the following three elementary remarks about numerical computation and to illustrate them by examples:

1. The straightforward approach to a given problem is rarely the best; insight and cleverness in the choice of a numerical method often pay enormous dividends in efficiency of computation.
2. It is frequently possible to extrapolate from a (finite) sequence of successive approximations to the solution of some problem to get another approximation which is more accurate than any member of the sequence.
3. An explicit and mathematically exact solution to a problem may be useless numerically, while very simple and apparently crude approximations may give very satisfactory numerical results. A *mathematical* solution and a *computationally effective* solution to a given problem may be entirely different things.

1. Consider, for example, the problem of evaluating numerically the integral of a given function $f(x)$ over a finite interval $[a, b]$. Perhaps the most straightforward imaginable way to set about doing this is to put down a grid of uniformly spaced points

$$x_j = a + j \times h \text{ where } h = (b - a)/n.$$

and to approximate the integral by the area under the piecewise linear arc composed of the segments from $(x_j, f(x_j))$ to $(x_{j+1}, f(x_{j+1}))$, $j = 0, 1, \ldots, n-1$. This approach gives the approximation

$$\int_a^b f(x)dx \approx h[1/2f(x_0) + f(x_1) + \cdots + f(x_{n-1}) + 1/2f(x_n)] \equiv T_n,$$

known as the *trapezoid rule*. (The intervals $[x_j, x_{j+1}]$ into which $[a, b]$ is subdivided are referred to as *panels*.)

Although the trapezoid rule *does* work—i.e., $T_n \to \int_a^b f(x)dx$ as $n \to \infty$—it converges slowly. A more efficient method can be constructed as follows: Start, as before, by breaking the integration interval up into n panels of equal length; this time, take n to be even. Group these panels into successive pairs $[x_{2j}, x_{2j+1}]$, $[x_{2j+1}, x_{2j+2}]$. Approximate the integral of $f(x)$ from x_{2j} to x_{2j+2} by the integral, over the same interval, of the quadratic polynomial agreeing with $f(x)$ at $x_{2j}, x_{2j+1}, x_{2j+2}$. Sum the results over j to get an approximation to the whole integral. This gives the approximation

$$\int_a^b f(x)dx \approx \frac{h}{3}[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+\cdots+4f(x_{n-1})+f(x_n)] \equiv S_n,$$

known as *Simpson's rule*. To give some feeling for the relative effectiveness of the trapezoid rule and Simpson's rule, we apply them to the evaluation of $\int_1^2 dx/x$, for various choices of n. The results are shown in Table 1. (The last column in the table will be explained shortly.) Each table entry has been rounded to show roughly one more digit than is accurate. It is clear from the table that, in this example at least, Simpson's rule produces much more accurate results for a given amount of work; for example, Simpson's rule with 32 panels is more accurate than the trapezoid rule with 512.

n	T_n	S_n	$\hat{S}_n$
8	0.6941	0.693154	0.6931479
16	0.69339	0.6931476	0.69314719
32	0.69320	0.69314721	0.6931471807
64	0.693162	0.693147182	0.693147180563
128	0.693151	0.6931471807	0.69314718056000
256	0.6931481	0.693147180567	0.693147180559947
512	0.6931474	0.6931471805604	0.693147180559946

Table 1.

2. We next introduce a general technique, known as *Richardson extrapolation*, for extracting, from a sequence of successive approximations, a more accurate estimate of their limit. We will present this technique by applying it to accelerating the convergence of Simpson's rule. To get started, we need a theoretical error estimate for Simpson's rule which we state here without proof:

Proposition. *If f is six times continuously differentiable on $[a,b]$, then there is a constant c such that*

$$S_n = \int_a^b f(x)dx + cn^{-4} + \mathcal{O}(n^{-6}).$$

From this result and very simple algebra, we see that the principal error term cn^{-4} cancels out of

$$\hat{S}_n \equiv \frac{1}{15}[16S_n - S_{n/2}] = \int_a^b f(x)dx + \mathcal{O}(n^{-6}),$$

which we might therefore hope to be a better method for approximating definite integrals than Simpson's rule itself. Note that, since S_n is defined only for n even, $\hat{S}_n$ is defined only for n a multiple of 4. The last column of Table 4.1 shows that this method does indeed work better on the test problem considered above.

Both the improvement in going from the trapezoid rule to Simpson's rule, and that given by applying Richardson extrapolation to Simpson's rule, become more dramatic if higher accuracy is required. For example, to evaluate $\int_1^2 dx/x$ with an error of less than 10^{-10} requires

- about a quarter of a million panels with the trapezoid rule.
- between 128 and 256 panels with Simpson's rule.
- between 32 and 64 panels with Simpson's rule plus Richardson extrapolation.

There is a variant on Richardson extrapolation, known as *Aitken extrapolation*, which works as follows: Suppose we have a sequence x_n whose large-n behavior is

$$x_\infty + c\gamma^n + o(\gamma^n)$$

where c and γ are constants whose values are not known. We want to accelerate the convergence of x_n to x_∞ by subtracting an estimate for $c\gamma^n$. Since both c and γ have to be determined, we need to use x_{n-1} and x_{n-2}, as well as x_n, to compute the correction. To derive the correction formula, let us assume that the asymptotic form is exact, i.e., that

$$x_n = x_\infty + c\gamma^n.$$

Then

$$x_n - x_{n-1} = c\gamma^{n-1}(\gamma - 1)$$

$$\frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}} = \gamma$$

$$c\gamma^n = (x_n - x_{n-1})\frac{\gamma}{\gamma - 1} = \frac{(x_n - x_{n-1})^2}{(x_n - x_{n-1}) - (x_{n-1} - x_{n-2})}$$

This suggests taking as a corrected sequence

$$\hat{x}_n = x_n - \frac{(x_n - x_{n-1})^2}{(x_n - x_{n-1}) - (x_{n-1} - x_{n-2})},$$

and it is easy to verify that this correction does work in the sense that, if $x_n = x_\infty + c\gamma^n + o(\gamma^n)$, then $\hat{x}_n = x_\infty + o(\gamma^n)$. This method can be applied "experimentally", i.e., given a sequence of approximations to some quantity, one can construct a corrected sequence using the above formula and see whether it settles down more quickly than the original one. The correction can also be applied repeatedly to eliminate several error terms which vary exponentially with different rates.

Here is a practical example of the use of Aitken extrapolation. The problem is to determine "empirically" Feigenbaum's constant δ as follows: Let $f_\mu(x) = x^2 - \mu$. There is an increasing sequence of parameter values μ_n converging to a limit $\mu_\infty = 1.4011\cdots$ such that 0 is periodic with period 2^n for f_{μ_n}. Asymptotically, $\mu_n \approx \mu_\infty + c\delta^{-n}$, so if we put

$$r_n = \frac{(\mu_{n-1} - \mu_{n-2})}{(\mu_n - \mu_{n-1})},$$

the sequence r_n converges to δ. Table 2 shows the effect of applying Aitken extrapolation repeatedly to this sequence, computed with high accuracy up to $n = 20$. As an aid to the eye, we have rounded each column to one digit more than seems to have settled down. (More precisely: We round to n digits, with n such that the absolute value of the difference between the bottom two entries in the column lies between $.5 \times 10^{-n}$ and 5×10^{-n}. The rounding was done *after* the extrapolation. Also, we reproduce only the last five lines in the table, although earlier ones were used in the extrapolation.)

n	Unextrapolated	Extrapolated once	Extrapolated twice
16	4.66920 16084 8	4.66920 16091 02731	4.66920 16091 02991 0000
17	4.66920 16089 7	4.66920 16091 03023	4.66920 16091 02990 6959
18	4.66920 16090 7	4.66920 16091 02987	4.66920 16091 02990 6735
19	4.66920 16091 0	4.66920 16091 02991	4.66920 16091 02990 6720
20	4.66920 16091 0	4.66920 16091 02991	4.66920 16091 02990 6719

n	Extrapolated three times	Extrapolated four times
16	4.66920 16091 02990 63518	4.66920 16091 02990 67947 9
17	4.66920 16091 02990 67414	4.66920 16091 02990 67214 1
18	4.66920 16091 02990 67173	4.66920 16091 02990 67186 8
19	4.66920 16091 02990 67186	4.66920 16091 02990 67185 4
20	4.66920 16091 02990 67185	4.66920 16091 02990 67185 3

Table 2.

The table makes it clear that Aitken extrapolation is very effective in improving the convergence of this sequence. For example, the raw sequence r_n has settled down only to the eleventh decimal place but the four-times extrapolated sequence has settled down to the twenty-first place. (With further extrapolations, it is possible to get one more digit to converge.) One surprising consequence of this analysis is that the r_n with $n \leq 20$ contain much more information about the limit than is apparent to the naked eye (This information is *not* however still present in the unextrapolated values reproduced in the table; it is mostly contained in the digits which have been rounded off.)

3. One of the unexpected aspects of numerical analysis is the phenomenon— or group of loosely related phenomena—referred to as *numerical instability*. It

often turns out that methods which look very reasonable mathematically fail disastrously in practice. We will illustrate this in an simple and comprehensible example, borrowed from the textbook of Dahlquist and Bjorck[3]. The example is the computation of the numbers

$$y_n = \int_0^1 \frac{x^n}{x+5}dx,$$

for $n = 0, \ldots$. Evidently,

$$y_0 = \log(6/5),$$

and it is easy to derive a formula for y_n in terms of y_{n-1}:

$$\begin{aligned} y_n &= \int_0^1 \frac{(x^n + 5x^{n-1}) - 5x^{n-1}}{x+5}dx \\ &= \int_0^1 x^{n-1}dx - 5\int_0^1 \frac{x^{n-1}}{x+5}dx \\ &= \frac{1}{n} - 5y_{n-1}. \end{aligned}$$

This recursion relation together with the explicit value for y_0 constitutes, from the mathematical point of view, a complete solution to the problem of computing the y_n. Using this solution in the straightforward way to compute the y_n numerically leads, however, to nonsensical results. For example, computing with roughly 16 digits of precision, we get $y_{22} \simeq -.1925$ (which has the wrong sign) and $y_{27} \simeq 624.5$ (which is much too large).

In this case, the source of the trouble is easy to find. From the recurrence relation

$$y_n = \frac{1}{n} - 5y_{n-1}$$

any error in y_{n-1} gets multiplied by -5 in computing y_n. The numerical value obtained for y_0 is likely to be off by something of the order of 10^{-16}, and an error of ϵ in y_0 contributes $(-5)^n\epsilon$ to the error in y_n. (There will be additional error from round-off in the arithmetic operations along the way.) Thus, as soon as 5^n becomes comparable to to 10^{16}, we have to expect the computed value of y_n to be nonsense, and this is exactly what the computation shows.

Here is one way to fix the computation. We can rewrite the recursion relation (with n replaced by $n+1$) as

$$y_n = \frac{1}{5(n+1)} - \frac{y_{n+1}}{5}.$$

This downward recursion relation is stable; error ϵ in y_{n+1} contributes only error $-\epsilon/5$ to y_n. Mathematically, this form is not very useful, since we don't have a starting value, but numerically it is perfectly usable: It is evident from the definition that y_n is between 0 and 1/5, so if we choose some large N and approximate y_N by 0 we are making an error of no more than 1/5. If we now apply the downward recursion formula to compute approximations to the y_n for $n < N$, the error in the starting value contributes an error of no more than $5^{-(N+1-n)}$ to y_n. Thus, for example, as soon as $n \leq N - 22$, the contribution of the error in the starting value to the error in y_n has magnitude less than 10^{-16}.

5. Working with polynomials.

One of the standard techniques of numerical analysis is to approximate some operation on a general function by performing that operation on a polynomial approximating the function. Furthermore, the approximating polynomial is frequently constructed simply by forcing it to be *equal* to the function being approximated at a sufficient number of points, i.e., by interpolation. The derivation of Simpson's Rule alluded to in the preceding section is an example of this method. The fundamental existence and uniqueness theorem for interpolating polynomials is the following well-known result:

Theorem. *Given $n+1$ distinct points $x_0, \ldots, x_n$ and $n+1$ values $y_0, \ldots, y_n$, there is a unique polynomial $p(x)$ of degree not greater than n with*

$$p(x_i) = y_i \qquad \text{for } i = 0, \ldots, n.$$

If the y_i are function values $f(x_i)$, we speak of $p(x)$ as *interpolating* f at $x_0, \ldots, x_n$. We will discuss here some simple—but not entirely obvious—techniques for computing and working with interpolating polynomials.

The conventional representation for a polynomial

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

(known as its *power form*) is often not well adapted to numerical computations, and we will have occasion to use a more general representation known in the numerical analysis literature as the *Newton form*:

$$p(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \cdots + c_n(x - x_0)\cdots(x - x_{n-1}).$$

Here, $x_0, \ldots, x_{n-1}$ are n points, not necessarily distinct, called the *centers* of the representation. The power form is a special case with $x_0 = \cdots = x_{n-1} = 0$. It is easy to show that, for any given $x_0, \ldots, x_{n-1}$, every polynomial of degree n can be written uniquely in the above form.

Now let $p(x)$ denote the polynomial interpolating f at $n+1$ distinct points $x_0, \ldots, x_n$. We are going to present an algorithm for computing the coefficients $c_0, \ldots, c_n$ of the Newton form for $p(x)$ with centers equal to the interpolation points. Rather than derive this algorithm directly, we will approach it by indirection as follows: Given a function $f(x)$ and a point x_0, we define

$$f[x_0, x] = \frac{f(x) - f(x_0)}{x - x_0} \qquad \text{for } x \neq x_0.$$

We can regard $f[x_0, x]$ as either

- a function of the two variables x_0, x

• a function of one variable (x) depending on a parameter (x_0)

Evidently, if f is differentiable, we can write

$$f[x_0, x] = f'(\xi) \qquad \text{for some } \xi \text{ between } x_0 \text{ and } x.$$

$f[x_0, x]$ is called the *first order divided difference* of f.

We define higher-order divided differences recursively as follows: Given a function f and $n+1$ points $x_0, \dots, x_n$, we put

$$f[x_0, \dots, x_n, x] = q[x_n, x] \qquad \text{where } q(x) \equiv f[x_0, \dots, x_{n-1}, x],$$

with $x_0, \dots, x_{n-1}$ held fixed. Thus: $n+1$st order divided differences are defined by differencing nth order divided differences in their last variable. Since $f[x_0, x]$ is clearly some kind of discrete approximation to a derivative, $f[x_0, \dots, x_{n-1}, x]$ is a discrete approximation to an nth derivative. (Actually, as we shall see, it is a discrete approximation to $f^{(n)}(x)/n!$ rather than $f^{(n)}(x)$.) For simplicity, we take $f[x_0, \dots, x_{n-1}, x]$ to be defined only for *distinct* values of its arguments. Extending the formalism to allow for repeated arguments (provided f is sufficiently differentiable) is not difficult. Finally, anticipating a result to be proved shortly, we remark that, although $f[x_0, \dots, x_{n-1}, x]$ is defined in a way which treats its arguments nonsymmetrically, it is in fact a symmetric function.

The immediate interest of divided differences is:

Proposition. *If $p(x)$ is the polynomial interpolating f at $x_0, \dots, x_n$, then*

$$\begin{aligned} p(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) \\ + \cdots + f[x_0, \dots, x_n](x - x_0) \cdots (x - x_{n-1}). \end{aligned}$$

Corollary. *$f[x_0, \dots, x_n]$ is a symmetric function of $x_0, \dots, x_n$.*

The corollary follows easily from the proposition since the latter says that $f[x_0, \dots, x_n]$ is the leading coefficient of the polynomial interpolating $f(x)$ at $x_0, \dots, x_n$, and this polynomial is unchanged if $x_0, \dots, x_n$ are permuted.

Proof (of the proposition). We first remark that it is enough to prove that, for any $n, x_0, \dots, x_n, f[x_0, \dots, x_n]$ is the leading coefficient of the polynomial interpolating $f(x)$ at $x_0, \dots, x_n$. To see that this suffices, we observe that if

$$c_0 + c_1(x - x_0) + \cdots + c_n(x - x_0) \cdots (x - x_{n-1})$$

interpolates $f(x)$ at $x_0, \dots, x_n$, then, for $j \leq n$, the polynomial interpolating $f(x)$ at $x_0, \dots, x_j$ is just

$$c_0 + c_1(x - x_0) + \cdots + c_j(x - x_0) \cdots (x - x_j - 1).$$

Applying the leading coefficient result with n replaced by j gives

$$c_j = f[x_0, \ldots, x_j]$$

To prove that the leading coefficient of the interpolating polynomial is given by the divided difference, we make a number of general remarks about divided differences:

1. If $p(x) = x^n, \quad (n = 1, 2, \ldots)$ then

$$p[x_0, x] = \frac{x^n - x_0^n}{x - x_0} = x^{n-1} + x^{n-2}x_0 + \cdots + x_0^{n-1}$$

   is a polynomial in x of degree $n-1$ with leading coefficient 1.
2. If $p(x) = a_0 + a_1x + \cdots + a_nx^n$, then $p[x_0, x]$ (as a function of x at fixed x_0) is a polynomial of degree not greater than $n-1$ and the coefficient of x^{n-1} is again a_n.
3. If $p(x)$ is as in 2., $p[x_0, \ldots, x_{n-1}, x] \equiv a_n$.

Applying this to the polynomial $p(x)$ interpolating $f(x)$ at $x_0, \ldots, x_n$ shows that the coefficient of x^n is $p[x_0, \ldots, x_n]$. But since $p(x_i) = f(x_i)$ for $i = 0, \ldots, n$, $p[x_0, \ldots, x_n] = f[x_0, \ldots, x_n]$, completing the proof.

Before taking up the question of explicit computation of divided differences, we digress briefly to note some alternative expressions for them which are useful for theoretical purposes. We consider $n+1$ distinct points $x_0, \ldots, x_n$ which, for definiteness, we assume to be in increasing order: $x_0 < x_1 < x_2 < \cdots < x_n$. Let $p(x)$ be the polynomial interpolating $f(x)$ at these points. Since $f(x) - p(x)$ vanishes at the $n+1$ points $x_0, \ldots, x_n$, Rolle's Theorem says that there exist points $x_0', \ldots, x_{n-1}'$, with

$$x_0 < x_0' < x_1 < x_1' < \cdots < x_{n-1}' < x_n$$

such that $f'(x_i') - p'(x_i') = 0$ for $i = 0, 1, 2, \ldots, n-1$. Repeating this argument n times, we see that there is a ξ, with $x_0 < \xi < x_n$, such that

$$f^{(n)}(\xi) - p^{(n)}(\xi) = 0$$

But, since $p(x)$ is a polynomial of degree not greater than n with leading coefficient $f[x_0, \ldots, x_n]$,

$$p^{(n)}(x) \equiv n! f[x_0, \ldots, x_n]$$

so

$$f[x_0, \ldots, x_n] = \frac{1}{n!} f^{(n)}(\xi)$$

Summing up: If f is n times differentiable on (x_0, x_n), there exists a ξ in (x_0, x_n) such that

$$f[x_0, \ldots, x_n] = \frac{1}{n!} f^{(n)}(\xi).$$

Alternatively, it is not difficult to show that if $f(x)$ is analytic on a connected domain Ω containing $x_0, \ldots, x_n$, then

$$f[x_0, \ldots, x_n] = \frac{1}{2\pi i} \oint_\gamma \frac{f(\varsigma) d\varsigma}{(\varsigma - x_0) \cdots (\varsigma - x_n)}$$

where γ is any contour in Ω winding once around each of $x_0, \ldots, x_n$.

Given the above formulas for divided differences, it is easy to estimate the error in approximation by interpolation. Let $f(x)$ be a sufficiently smooth function; $x_0, \ldots, x_n$ $n+1$ distinct points; $p(x)$ the polynomial interpolating $f(x)$ at $x_0, \ldots, x_n$; $\bar{x}$ another point. We want to estimate $f(\bar{x}) - p(\bar{x})$. There is evidently no need to consider the case where $\bar{x}$ is one of the x_i's. Otherwise, we let $\bar{p}(x)$ denote the polynomial interpolating $f(x)$ at $x_0, \ldots, x_n, \bar{x}$. Then, on the one hand,

$$\bar{p}(\bar{x}) = f(\bar{x})$$

and, on the other,

$$\bar{p}(x) = p(x) + f[x_0, \ldots, x_n, \bar{x}](x - x_0) \cdots (x - x_n)$$

so

$$\begin{aligned} f(\bar{x}) - p(\bar{x}) &= f[x_0, \ldots, x_n, \bar{x}](\bar{x} - x_0) \cdots (\bar{x} - x_n) \\ &= \frac{1}{(n+1)!} f^{(n+1)}(\xi)(\bar{x} - x_0) \cdots (\bar{x} - x_n) \end{aligned}$$

for some ξ in the smallest interval containing $x_0, \ldots, x_n, \bar{x}$. Again, there is also a formula in terms of a contour integral.

We now take up the question of computing divided differences. From symmetry (and the definition) we get

$$f[x_j, x_{j+1}, \ldots, x_k] = \frac{f[x_j, \ldots, x_{k-1}] - f[x_{j+1}, \ldots, x_k]}{x_j - x_k}.$$

Thus, to compute the coefficients $f(x_0), f[x_0, x_1], \ldots f[x_0, \ldots, x_n]$ of the interpolating polynomial, one can proceed by computing first $f[x_i, x_{i+1}]$ for $i = 0, 1, \ldots, n-1$, then $f[x_i, x_{i+1}, x_{i+2}]$ for $i = 0, 1, \ldots, n-2$, $\ldots$, and finally $f[x_0, \ldots, x_n]$.

This leads to a very simple algorithm for computing the coefficients: Suppose we have arrays containing $x_0, \ldots, x_n$ and $f_0, \ldots, f_n$ $(\equiv f(x_0), \ldots, f(x_n))$. The following replaces $f_0, \ldots, f_n$ by $f[x_0], f[x_0, x_1], \ldots, f[x_0, \ldots, x_n]$:

```
for $i := 1$ to n
    for $j := n$ downto i
        $f_j := (f_j - f_{j-1})/(x_j - x_{j-i})$
```

Similarly, if we are given $x_0, \dots, x_n$, a point $\bar{x}$, and an array of coefficients $c_0, \dots, c_n$, then we can compute

$$p(\bar{x}) = c_0 + c_1(\bar{x} - x_0) + c_2(\bar{x} - x_0)(\bar{x} - x_1) + \cdots + c_n(\bar{x} - x_0)\cdots(\bar{x} - x_{n-1})$$

by

```
$y := c_n$
for $j := n - 1$ downto 0
    $y := y \times (\bar{x} - x_j) + c_j$.
```

On exiting the loop, $y = p(\bar{x})$.

It is sometimes useful to be able to change the set of centers with respect to which a polynomial is represented—for example, to pass from Newton form to power form. There is of course a straightforward, brute-force solution to this problem, but there is also a simpler non-obvious solution. We will approach this solution indirectly by asking what looks at first like an unrelated question. The above program-fragment for evaluating the polynomial can be re-expressed as follows: Put $c'_n = c_n$; then construct successively $c'_{n-1}, c'_{n-2}, \dots, c'_0$ by

$$c'_j = c'_{j+1}(\bar{x} - x_j) + c_j$$

(where $\bar{x}$ is the point at which the polynomial is to be evaluated). As noted above, $c'_0 = p(\bar{x})$. The question we want to ask is: What is the significance of the other c'_js? The following proposition gives the answer:

Proposition. *With the above notation,*

$$p(x) = c'_0 + c'_1(x - \bar{x}) + c'_2(x - \bar{x})(x - x_0) + \cdots + c'_n(x - \bar{x})(x - x_0)\cdots(x - x_{n-2})$$

i.e., $c'_0, \dots, c'_n$ are the coefficients of $p(x)$ in the representation with centers $\bar{x}, x_0, \dots, x_{n-1}$.

Remark. If we want to pass from the representation with centers $x_0, \dots, x_n$ to that with centers $\bar{x}_0, \dots, \bar{x}_n$, we can do it by a sequence of $n + 1$ steps: First pass to the representation with centers $\bar{x}_n, x_0, \dots, x_{n-1}$, then to centers $\bar{x}_{n-1}, \bar{x}_n, x_0, \dots, x_{n-2}$, and so on. The proposition gives an easy way to perform each of these steps.

Proof. Although the proposition does not require that $\bar{x}, x_0, \dots, x_n$ be distinct, it suffices to prove it under that assumption since the general case follows by continuity. We start with

$$c'_n = c_n = p[x_0, \dots, x_n] = p[\bar{x}, x_0, \dots, x_{n-1}]$$

(since $p[x_0, \ldots, x_n]$ is independent of $x_0, \ldots, x_n$.) We will show by downward induction on j that $c'_j = p[\bar{x}, x_0, \ldots, x_{j-1}]$. Thus, assume that this is true for $j+1$ and recall that

$$p[\bar{x}, x_0, \ldots, x_j] = \{p[\bar{x}, x_0, \ldots, x_{j-1}] - p[x_0, \ldots, x_j]\}/(\bar{x} - x_j)$$

Thus:

$$\begin{aligned} p[\bar{x}, x_0, \ldots, x_{j-1}] &= p[x_0, \ldots, x_j] + (\bar{x} - x_j)p[\bar{x}, x_0, \ldots, x_j] \\ &= c_j + (\bar{x} - x_j)c'_{j+1} \qquad \text{by the induction hypothesis.} \\ &= c'_j \qquad \text{by the definition of } c'_j, \end{aligned}$$

as desired.

The proposition also provides a method for evaluating the derivative of a polynomial written in Newton form; one simply reworks the polynomial into a form with the first *two* centers at the place where the derivative is to be evaluated. In the above notation, this can be done as follows: Put

$$c'_n = c''_n = c_n$$

and work downward constructing

$$\begin{aligned} c'_j &= c'_{j+1} \times (\bar{x} - x_j) + c_j \\ c''_j &= c''_{j+1} \times (\bar{x} - x_{j-1}) + c'_j \end{aligned}$$

for $j = n-1, \ldots, 1$; then $c''_1 = p'(\bar{x})$. In this way, both $p(\bar{x})$ and $p'(\bar{x})$ can be evaluated with a total of $2n-1$ multiplications and $2n-1$ additions.

6. Best uniform approximation by polynomials.

To motivate the next topic to be taken up, let us consider briefly how one might go about computing the exponential function on the computer. An obvious first step is to reduce to the problem to one of computing e^x for $-1/2 \leq x \leq 1/2$ by using $e^{n+x} = e^n e^x$. For x between $-1/2$ and $1/2$, the first thing which comes to mind is to use the Taylor series. A little reflection suggests, however, that it ought to be possible to do better: A given finite sum of terms in the Taylor series will give much more accurate results for x near 0 than for x near $\pm 1/2$. If what we want is to make the error *uniformly* as small as possible with a given amount of work, it seems likely that it will be be possible to trade part of the accuracy near the center of the interval for a little more accuracy near the ends. We are thus led to consider the problem from a higher level and to ask what polynomial $p_n^*(x)$ of degree no greater than n makes

$$\sup_{-1/2 \leq x \leq 1/2} |e^x - p_n^*(x)|$$

as small as possible. This is a special case of the general problem of finding the *best uniform approximation* to a given (continuous) function on some interval by a polynomial of a given degree. It is this general problem that we will consider in this section.

Having given the above motivation, we now need to note that it is in fact considerably oversimplified.

1. In selecting a method for approximating the exponential function on the computer, accuracy at the general point is not the only consideration. For example, it is very desirable to preserve exactly the identity $e^0 = 1$. This can be accomplished by, for example, approximating $(e^x - 1)/x$ by a polynomial $p(x)$ and using $1 + xp(x)$ as the desired approximation to e^x.

2. For a fixed amount of computational work, one can usually get a better approximation using a rational function than using a polynomial. There is a systematic theory of best uniform approximation by rational functions (with numerator and denominator of fixed degrees) but this theory is distinctly more complicated than the theory for approximation by polynomials, and so, in the interest of simplicity, we will not discuss it.

We first need a simple result which works in the following general context: X will denote a compact Hausdorff space, Π a finite dimensional vector space of real-valued continuous functions on X (shortly to be specialized to X a closed bounded interval and Π the space of all polynomials of degree no greater than some fixed n). For h a continuous real-valued function on X, $\|h\|$ will denote the supremum norm of h:

$$\|h\| = \sup_{x \in X} |h(x)|$$

The question we want to discuss is: Given a continuous function f on X, how does one characterize the p's in Π which minimize $\|f-p\|$? A useful preliminary remark is that the minimization problem certainly does have at least one solution since $p \mapsto \|f-p\|$ is a continuous function on the finite dimensional space Π which goes to infinity at infinity.

Proposition. *For $p^* \in \Pi$, the following are equivalent:*
1. $\|f-p^*\| = \inf\{\|f-p\| : p \in \Pi\}$
2. Writing Y for $\{y \in X : |f(y)-p^*(y)| = \|f-p^*\|\}$, there is **no** *$r(x) \in \Pi$ which vanishes nowhere on Y and which has the same sign as $f(y)-p^*(y)$ everywhere on Y.*

Proof. 2.$\Rightarrow$ 1. We assume that 1. does not hold. Thus, there exists a $\tilde{p} \in \Pi$ such that $\|f-\tilde{p}\| < \|f-p^*\|$. We write $r = \tilde{p}-p^*$; we will argue that, everywhere on Y, r is non-zero and has the same sign as $f-p^*$, and thus that 2. also fails. The argument is almost immediate: Writing

$$f-\tilde{p} = f-p^* - (\tilde{p}-p^*) = f-p^*-r,$$

we see that, for $y \in Y$,

$$|f(y)-p^*(y)-r(y)| = |f(y)-\tilde{p}(y)| \leq \|f-\tilde{p}\| < \|f-p^*\| = |f(y)-p^*(y)|$$

(only the last equality uses $y \in Y$). This inequality implies that $r(y)$ cannot vanish and must have the same sign as $f(y)-p^*(y)$, as desired.

1. $\Rightarrow$ 2. We suppose that 2. does not hold, i.e., that there is an $r \in \Pi$ which does not vanish anywhere on Y and which has the same sign as $f-p^*$ everywhere on Y. The idea will be to show that moving p^* a little bit in the direction r gives a strictly better approximation to f, i.e., that

$$\|f-p^*-\lambda r\| < \|f-p^*\|$$

for sufficiently small positive λ. It is clear that

$$|f(y)-p^*(y)-\lambda r(y)| < |f(y)-p^*(y)|$$

on Y. What we have to do is to make sure that the improvement is not spoiled by what happens off Y. This is a simple continuity argument: We first note that there exist a neighborhood U of Y and a strictly positive ϵ such that

1. If $x \in U$, then $|f(x)-p^*(x)| \geq \|f-p^*\|/2$
2. If $x \in U$, then $|r(x)| > \epsilon$ and $r(x)$ has the same sign as $f(x)-p^*(x)$.
3. If x is not in U, then $|f(x)-p^*(x)| < \|f-p^*\| - \epsilon$

We write M for $\|r\|$. If, now, $0 < \lambda < \|f-p^*\|/2M$

$$|f(x)-p^*(x)-\lambda r(x)| \leq |f(x)-p^*(x)| - \lambda\epsilon \leq \|f-p^*\| - \lambda\epsilon$$

for $x \in U$, and, on the other hand, if $0 < \lambda < \epsilon/2M$, then

$$|f(x) - p^*(x) - \lambda r(x)| \leq |f(x) - p^*(x)| + \epsilon/2 \leq \|f - p^*\| - \epsilon/2$$

for $x \in X \setminus U$. Thus, for sufficiently small positive λ,

$$|f(x) - p^*(x) - \lambda r(x)| \leq \|f - p^*\| - \lambda\epsilon$$

for all x, as desired.

We now specialize as indicated above to X a compact interval $[a, b]$ and $\Pi = \Pi_n$, the space of polynomials of degree not greater than n. In this, more concrete, situation, we have a more specific result:

Theorem (Chebyshev). *Let f be a continuous real-valued function on $[a, b]$, and let p^* be a polynomial of degree not greater than n. Then the following are equivalent:*
1. $\|f - p^*\| = \inf\{\|f - p\| : p \in \Pi_n\}$
2. There is a sequence $x_0 < x_1 < \cdots < x_{n+1}$ of points of $[a, b]$ such that

$$|f(x_i) - p^*(x_i)| = \|f - p^*\| \quad \text{for} \quad i = 0, 1, \ldots, n + 1$$

and such that the sequence $f(x_i) - p^*(x_i)$ alternates in sign.

Condition 2. is often expressed by saying that $f - p^*$ has the *equi-oscillation property*. Note that the number of half-oscillations (i.e., runs from $\|f - p^*\|$ to $-\|f - p^*\|$ or the reverse) is $n + 1$.
Proof. 2. $\Rightarrow$ 1. (This is the more useful half.) Suppose $f - p^*$ has the equi-oscillation property but that there exists $\tilde{p} \in \Pi_n$ with $\|f - \tilde{p}\| < \|f - p^*\|$. Then, with the x_i's as in 2.,

$$\tilde{p}(x_i) - p^*(x_i) = [f(x_i) - p^*(x_i)] - [f(x_i) - \tilde{p}(x_i)]$$

has the same sign as $f(x_i) - p^*(x_i)$ and hence alternates in sign as i runs from 0 to $n + 1$. This implies that $\tilde{p}(x) - p^*(x)$ vanishes at least $n + 1$ times, and this is impossible for a polynomial of degree not greater than n which does not vanish identically.
1. $\Rightarrow$ 2. We will assume that $f - p^*$ does not have the equi-oscillation property and show that 1. cannot hold. By the preceding proposition, it suffices to produce a polynomial $r(x)$ of degree not greater than n which does not vanish anywhere on

$$Y \equiv \{\, y : |f(y) - p^*(y)| = \|f - p^*\| \,\}$$

and which has the same sign as $f(y) - p^*(y)$ everywhere on Y. To illustrate how to construct $r(x)$, we consider first the simple case where $Y = \{y_0 < y_1 < \cdots < y_m\}$

with $m \leq n$ and with the $f(y_i) - p^*(y_i)$ alternating in sign. In this case, we choose for each $i = 1, \ldots, m$ a z_i in (y_{i-1}, y_i), and we put

$$r(x) = \pm(x - z_1) \cdots (x - z_m),$$

where the sign is + or − according as $f(y_m) - p^*(y_m)$ is positive or negative.

To deal with the general case, we note that Y is closed and hence that $[a, b] \setminus Y$ is a union of relatively open intervals. We will call a connected component of $[a, b] \setminus Y$ a *sign-change interval* if both its end points are in Y and if $f(x) - p^*(x)$ has opposite signs at the two end points. A little thought shows that the negation of the equi-oscillation property is simply the statement that there are no more than n sign-change intervals. Label these intervals with $i = 1, \ldots, m \leq n$; choose for each i a z_i in the ith sign-change interval, and put

$$r(x) = \pm(x - z_1) \cdots (x - z_m)$$

with the overall sign chosen as in the simple case discussed above. The degenerate case $m = 0$ (no sign-change intervals) is possible; in this case, we take $r = \pm 1$.

Exercise. Show that, for any continuous f and any n, there is **only one** $p^* \in \Pi_n$ such that

$$\|f - p^*\| = \inf\{\, \|f - p\| : p \in \Pi_n \,\}$$

i.e., prove the uniqueness of the polynomial of best uniform approximation. (I don't know how to do this without using the equi-oscillation characterization of a polynomial of best uniform approximation.)

Given an interval $[a, b]$, a continuous function f on $[a, b]$, and a degree n, the polynomial of degree no greater than n which best uniformly approximates f on $[a, b]$ can usually be found only by numerical computation. (There are good algorithms for solving this problem, but they are relatively complicated.) There is, however, one entertaining exception to this general rule: The best uniform approximation to x^{n+1} by a polynomial of degree no greater than n can be found explicitly. It is clear that, if we can solve this approximation problem on just one interval, we can solve it on any other by a linear change of variables. We will work on the interval $[-1, 1]$.

Once again we will "guess" a solution and show that it works. The first step is to introduce the *Chebyshev polynomials*. The Chebyshev polynomial $T_n(x)$ of order n may be defined by the identity

$$\cos(n\theta) = T_n(\cos(\theta));$$

to justify this definition we need to show that $\cos(n\theta)$ can be written as a polynomial of degree n in $\cos(\theta)$. This is clear for $n = 0, 1$. For general n, we argue by induction. From the elementary trigonometric identities

$$\cos((n \pm 1)\theta) = \cos(n\theta)\cos(\theta) \mp \sin(n\theta)\sin(\theta)$$

we get

$$\cos((n+1)\theta) = 2\cos(n\theta)\cos(\theta) - \cos((n-1)\theta)$$

Hence, if we define a sequence of polynomials $T_n(x)$ by

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

and

$$T_0(x) = 1; \qquad T_1(x) = x,$$

it follows by induction that

$$\cos(n\theta) = T_n(\cos(\theta))$$

as desired. It is easy to see from the recursion relation that

$$T_n(x) = 2^{n-1}x^n + \text{ lower order terms.}$$

We can thus define for each n a polynomial $q_n(x)$ of degree not greater than n by

$$T_{n+1}(x) = 2^n(x^{n+1} - q_n(x)),$$

We claim that $q_n(x)$ is the desired best uniform approximant to x^{n+1}. The proof of this claim follows almost immediately by combining the definition of the Chebyshev polynomials with Chebyshev's characterization of the polynomial of best approximation: As θ runs from 0 to 1, $\cos(\theta)$ runs monotonically from 1 to -1, and $\cos((n+1)\theta)$ runs from 1 to -1, then back to 1,..., for a total of $n+1$ half-oscillations. Hence, as x runs from -1 to 1, $T_{n+1}(x)$ makes $n+1$ half-oscillations between 1 and -1. But this means exactly that the error $2^{-n}T_{n+1}(x)$ in approximating x^{n+1} by $q_n(x)$ has the equi-oscillation property of order n and hence that $q_n(x)$ is, as claimed, the best uniform approximation to x^{n+1}. It is worth noting that $q_n(x)$ is actually a polynomial of degree $n-1$ (rather than n); that it is even or odd according as $n+1$ is; and that the maximum error is 2^{-n}. It is also suggestive to observe that q_n can be obtained by interpolating the function to be approximated (x^{n+1}) at the *Chebyshev abscissae* of order n, i.e., the roots of $T_{n+1}(x)$, i.e., the numbers

$$\cos(\frac{(j+1/2)\pi}{n+1}), \quad j = 0, 1, \ldots, n.$$

Given a function f and an interval (which we suppress from the notation), we write

$$e_n(f) = \inf\{\, \|f - p\| : p \in \Pi_n \,\}.$$

The Weierstrass Approximation Theorem says that $e_n(f)$ goes to zero as n goes to infinity. It is of interest to have some more detailed information about its rate of decrease. There are a number of results bearing on this question, e.g.,

- for any strictly positive real number r and any norm $\|.\|_r$ defining the C^r topology, there is a constant c_r such that

$$e_n(f) \leq c_r n^{-r} \|f\|_r$$

  for any $f \in C^r$.
- if Ω is a complex domain containing $[a, b]$, there are constants $c_\Omega, \gamma > 0$ such that

$$e_n(f) \leq c_\Omega e^{-\gamma n} \sup_{z \in \Omega} |f(z)|$$

  for any function $f(z)$ bounded and analytic on Ω.

These estimates vary in a trivial way with the interval $[a, b]$ (by linear change of variable). They are proved for $[a, b] = [-1, 1]$ by considering

$$\phi(\theta) = f(\cos\theta)$$

Approximating $f(x)$ by a polynomial of degree no greater than n is equivalent to approximating $\phi(\theta)$ by an even, real-valued trigonometric polynomial of degree not greater than n. Classical harmonic analysis provides a powerful set of tools which can be brought to bear on this latter approximation problem.

As noted above, finding the polynomial of *best* approximation is a relatively complicated problem. If we relax the problem to one of trying to find a *good* approximation, it is both natural and computationally straightforward to try to approximate $f(x)$ by interpolating at $n+1$ points $x_0, \ldots, x_n$ distributed in some reasonable way over $[a, b]$. Surprisingly, this approach often fails catastrophically for large n, and especially if $x_0, \ldots, x_n$ are taken to be evenly spaced along $[a, b]$, i.e., $x_j = a + j \times h$, with $h = (b - a)/n$. For example: If $[a, b] = [-1, 1]$ and $f(x) = 1/(1 + 25x^2)$ *(Runge's example)* and if $p_n(x)$ is the polynomial interpolating $f(x)$ at $n+1$ evenly spaced points from -1 to 1 then not only does $\|f - p_n\|$ not go to zero as n goes to infinity, but in fact p_n blows up: $\|p_n\| \to \infty$. This statement is not difficult to prove analytically (using the contour-integral formula for the error in interpolation). What happens is that the approximation is excellent in the center of the interval, but at the ends of the interval $p_n(x)$ seems to rebel against being clamped down at the points x_j by overshooting drastically between them.

The bad behavior of interpolation at many evenly spaced points—known loosely as Runge's phenomenon—has widespread consequences for numerical analysis. It implies, for example, that it is a *very* bad idea to try to compute numerically the time derivative of some physical quantity by differentiating a polynomial of high degree interpolating experimentally measured values of that quantity at a large number of uniformly spaced times.

Interpolation at the Chebyshev abscissae works much better than interpolation at evenly spaced points for constructing good uniform approximations. For example, if f is Holder continuous on $[-1,1]$ and if $p_n(x)$ is the polynomial interpolating f at the nth order Chebyshev abscissae, then it can be shown that $\|f-p_n\|$ goes to zero as n goes to infinity. In fact, there is a result due to Powell which says that, whatever f is, $p_n(x)$ is almost as good an approximation as possible, in the following sense: There is a sequence of constants c_n such that

$$\|f-p_n\| \leq c_n e_n(f)$$

for all $f(x)$ continuous on [-1,1], and

$$c_n \leq \begin{cases} 3, & \text{for } n \leq 20; \\ 4, & \text{for } n \leq 100. \end{cases}$$

For large n, the c_n grow like $\log(n)$, but this is not of much practical importance since it would be very unusual to use an approximating polynomial of degree greater than 100. Thus, except for a few cases of particularly great practical importance (like the exponential function), it is generally good enough to generate polynomial approximations by interpolation at the Chebyshev abscissae rather than to undertake the more difficult task of finding the polynomial of best uniform approximation.

7. Numerical evaluation of integrals.

Newton-Cotes methods. The derivation of Simpson's rule sketched in Sect. 4 is a prototype for a general technique for constructing methods to evaluate integrals numerically. The idea is as follows: Pick a positive integer j ($j = 2$ for Simpson's rule); divide the interval of integration into a number n of subintervals (panels) of equal length, where n is a multiple of j; and organize the panels into groups of j, i.e., so that the first group of j covers the interval from x_0 to x_j, the second from x_j to x_{2j}, and so on. Then approximate $\int_{x_{kj}}^{x_{(k+1)j}} f(x)dx$ by the integral over the same interval of the polynomial (of degree not greater than j) interpolating f at $x_{kj}, \ldots, x_{(k+1)j}$. This gives an approximation of the form

$$\int_{x_{kj}}^{x_{(k+1)j}} f(x)dx \approx h[c_0^{(j)} f(x_{kj}) + c_1^{(j)} f(x_{kj+1}) + \cdots + c_j^{(j)} f(x_{(k+1)j})].$$

where $h \equiv (b-a)/n$ is the width of an individual panel and $c_0^{(j)}, \ldots, c_j^{(j)}$ are constants which do not depend on f or h, i.e., which are "universal". It is immediate from the derivation that this approximate formula is actually exact if f is a polynomial of degree not greater than j. What we have so far is an approximation to the integral over a group of j successive panels; to get an approximation to the total integral we have only to add up the contributions from the various groups of panels. The methods obtained in this way are called *Newton-Cotes* methods.

It can be shown fairly easily that the j-panel Newton-Cotes method has error asymptotically $\mathcal{O}(n^{-(j+1)})$ for odd j and $\mathcal{O}(n^{-(j+2)})$ for even j, provided that the function f being integrated is smooth enough. Thus, by taking j large, we can get a rapidly convergent method. The instability of evenly spaced interpolation, unfortunately, turns up here: The coefficients $c_k^{(j)}$, $k = 0, \ldots, j$ are all non-negative only for $j = 1, 2, \ldots, 7$, and 9, and non-positivity of some of the coefficients implies that the method has less than ideal error propagation properties, for the following reason: Suppose we know the $f(x_i)$'s only with uncertainty ϵ, i.e., we know only that each computed $f(x_i)$'s differs from the corresponding exact value by an error which is no larger in magnitude than ϵ. The uncertainty in the function values propagates to an uncertainty of

$$|b-a|\epsilon \times \left(|c_0^{(j)}| + |c_1^{(j)}| + \cdots + |c_j^{(j)}|\right)/j$$

in the approximation to the integral. Because the methods we are considering are exact for constant functions, we have

$$c_0^{(j)} + c_1^{(j)} + \cdots + c_j^{(j)} = j;$$

hence, the factor

$$\left(|c_0^{(j)}| + |c_1^{(j)}| + \cdots + |c_j^{(j)}|\right)/j$$

in the above error estimate is one if the $c_k^{(j)}$ are all non-negative and is strictly greater than one otherwise.

Romberg's method. Repeated Richardson extrapolation applied to the trapezoid approximation gives a very effective technique for evaluating integrals numerically known as *Romberg's method.* Suppose we want to evaluate $\int_a^b f(x)dx$ for a smooth function f. We write T_n for the n-panel trapezoid approximation to the integral, i.e.,

$$T_n = \frac{b-a}{n}[1/2f(x_0) + f(x_1) + \cdots + f(x_{n-1}) + 1/2f(x_n)].$$

Romberg's method is based on the following result, which gives an asymptotic expansion for T_n for large n.

Proposition. *If $f \in C^{2r+2}$, there exist constants $c_1, \ldots, c_r$ (depending on f) such that*

$$T_n = \int_a^b f(x)dx + c_1 n^{-2} + c_2 n^{-4} + \cdots + c_r n^{-2r} + O(n^{-(2r+2)}).$$

We omit the proof of this proposition; it is an immediate consequence of the Euler-Maclaurin summation formula. It is perhaps not surprising that T_n admits an asymptotic expansion in powers of n^{-1}, but it does not seem obvious that, as the proposition indicates, the odd powers of n^{-1} do not appear in this expansion. While the proposition does not guarantee that all even powers appear, i.e., that the c_i are all different from zero, they are in fact generically non-zero.*

The basic idea of Richardson extrapolation is the combine two approximations to some quantity one wants to compute in order to cancel the dominant term in the error. We start by doing this with T_n and $T_{n/2}$. A simple calculation shows that if we define

$$T_n^{(1)} = \frac{1}{3}\left(4T_n - T_{n/2}\right) = T_n + \frac{1}{3}\left(T_n - T_{n/2}\right)$$

(for even n), then

$$T_n^{(1)} = \int_a^b f(x)dx - 4c_2 n^{-4} - 20c_3 n^{-6} + \cdots.$$

* There is, however, an important non-generic case: The c_i all vanish if f is periodic with period $b-a$. This does not imply that the error in the trapezoid rule is zero, but does imply that it decreases faster than any inverse power of n if f is infinitely differentiable. In this case, applying Richardson extrapolation is almost certain to make the convergence worse.

Thus, as we intended, the term in n^{-2} has disappeared from the asymptotic expansion for $T_n^{(1)}$. It is worth noting, however, that the other terms now have bigger coefficients. Thus, if the n^{-2} term happens to be absent, Richardson extrapolation will actually make the convergence worse. For this reason, it is important to know exactly which error terms are likely to be present before applying Richardson extrapolation.

Similarly, we define

$$T_n^{(2)} = \frac{1}{15}\left(16T_n^{(1)} - T_{n/2}^{(1)}\right) = T_n^{(1)} + \frac{1}{15}\left(T_n^{(1)} - T_{n/2}^{(1)}\right)$$

(for n a multiple of 4), then $T_n^{(2)}$ has an asymptotic expansion starting with n^{-6}. The process can be repeated; the next two extrapolation coefficients are 1/63 and 1/255, and the general coefficient is $1/(4^j - 1)$.

This procedure makes it very straightforward to construct rapidly converging sequences of approximations to the desired integral. A particularly effective way to use it is by computing a triangular array of approximations to the integral as follows: First compute T_1; then T_2 and $T_2^{(1)}$; then T_4, $T_4^{(1)}$, and $T_4^{(2)}$; and so on. After this process has gone on for a while, computation of a new T_{2^n} will be relatively time-consuming (although the computation can be speeded up by noting that half the work has already been done in evaluating $T_{2^{n-1}}$). Once T_{2^n} is available, however, the $T_{2^n}^{(j)}$ for j from 1 to n are cheap to compute. By comparing each of the $T_{2^n}^{(j)}$'s with the corresponding $T_{2^{n-1}}^{(j)}$, it is possible to estimate the error in that approximation to the integral and thus to decide whether one of the available approximations provides sufficient accuracy or whether it is necessary to go on to the next power of 2.

Table 3 illustrates how this process works in practice. It shows the result of using the method just described to evaluate

$$\int_1^{20} \frac{dx}{x} = \log(20) = 2.99573227\cdots$$

This integral is mildly difficult to do because quite a lot of the mass comes from a small part of the interval of integration. To make the table more readable, we have taken advantage of knowing the correct answer to round each approximation roughly one digit beyond the last correct one. It is worth noting that the difference between a given approximation and the table entry immediately above it gives a very conservative estimate of its error.

Gaussian integration. We come now to one of the most elegant ideas in all of numerical analysis. We start from the fact, easy to prove, that given any $n+1$ distinct points $x_0, \ldots, x_n$, there is a unique set of coefficients $A_0, \ldots, A_n$ such

n	T_n	$T_n^{(1)}$	$T_n^{(2)}$	$T_n^{(3)}$	$T_n^{(4)}$	$T_n^{(5)}$
1	10.0					
2	5.9	4.5				
4	4.1	3.5	3.4			
8	3.4	3.1	3.1	3.08		
16	3.1	3.02	3.01	3.01	3.01	
32	3.03	2.998	2.997	2.997	2.997	2.997
64	3.01	2.9960	2.9958	2.99578	2.99577	2.99577
128	2.998	2.99575	2.995734	2.995733	2.995733	2.995733

Table 3.

that

$$\int_{-1}^{1} p(x)dx = \sum_{i=0}^{n} A_i p(x_i)$$

for all polynomials $p(x)$ of degree not greater than n. (The coefficients A_i are generally referred to as *weights*, although they need not, in general, all be positive.) It is easy to find choices of $x_0, \ldots, x_n$ which make this formula exact for all polynomials of degree up to some $\bar{n} > n$. For example, if n is even and if the set $\{x_0, \ldots, x_n\}$ is symmetric about 0, then the A_i's, since they are uniquely determined, are also symmetric (in an obvious sense), and therefore $\int_{-1}^{1} x^{n+1}dx$ equals $\sum_{i=0}^{n} A_i(x_i)^{n+1}$ since both are zero. Hence, the formula is exact for all polynomials of degree up to at least $n+1$.

The question we want to study is: How should the $x_0, \ldots, x_n$ be chosen in order to get the formula to be exact for polynomial up to as high a degree as possible? Simple variable counting suggests that, since there are $2n+2$ parameters available ($n+1$ x_i's and $n+1$ A_i's), it should be possible to satisfy $2n+2$ conditions and hence to make the formula exact for all polynomials of degree not greater than $2n+1$. A direct attack on this question is not very attractive, since it involves trying to solve a set of $2n+2$ strongly non-linear simultaneous equations.

There is, however, an indirect approach, discovered by Gauss, which leads quickly to a completely satisfactory solution. To describe it, we need first to review briefly the theory of orthogonal polynomials. The Gram-Schmidt procedure, applied to the sequence of functions $1, x, x^2, \ldots$ with respect to the scalar product $(f,g) = \int_{-1}^{1} f(x)g(x)dx$, gives a mutually orthogonal sequence of polynomials $P_0, P_1(x), P_2(x), \ldots$. Appropriately normalized, these are just the Legendre polynomials, but what we need to know about them can most easily be derived directly from their orthogonality. From the fact that each $P_n(x)$ is a polynomial of degree exactly n, it follows easily that any polynomial of degree not greater than n can be expressed as a linear combination of $P_0, \ldots, P_n(x)$ and hence is orthogonal to $P_{n+1}(x)$. This, in turn, implies that $P_{n+1}(x)$ has $n+1$ distinct zeroes in $(-1,1)$. (Proof: If not, let $z_1 < \cdots < z_m$, where $m \leq n$, be all the points in $(-1,1)$ where $P_{n+1}(x)$ changes sign, and let $r(x) = (x-z_1)\cdots(x-z_m)$. Then $r(x)P_{n+1}(x)$ has constant sign on $(-1,1)$ so its integral cannot vanish, and this contradicts the fact that $r(x)$, as a polynomial of degree not greater than n, is orthogonal to $P_{n+1}(x)$.)

We are now ready to prove:

Theorem. *Let $x_0, \ldots, x_n$ be the zeros of $P_{n+1}(x)$, and let $A_0, \ldots, A_n$ be such that*

$$\int_{-1}^{1} p(x)dx = \sum_{i=0}^{n} A_i p(x_i) \tag{$*$}$$

for all polynomials $p(x)$ of degree not greater than n. Then $(*)$ holds in fact for all polynomials of degree not greater than $2n+1$.

Proof. Let $p(x)$ be a polynomial of degree not greater than $2n+1$. Using the division algorithm we can write

$$p(x) = q(x)P_{n+1}(x) + r(x)$$

where $q(x)$ and $r(x)$ both have degree not greater than n. By the choice of the A_i's, $(*)$ holds for $r(x)$, so it suffices to prove

$$\int_{-1}^{1} q(x)P_{n+1}(x)dx = \sum_{i=0}^{n} A_i q(x_i)P_{n+1}(x_i).$$

This equality holds because both sides are zero: The left-hand side because $P_{n+1}(x)$ is orthogonal to all polynomials of degree not greater than n; the right-hand side because all the $P_{n+1}(x_i)$'s vanish.

Remark. No matter how the $x_0, \ldots, x_n$ are chosen, the above integration formula cannot be exact for all polynomials of degree $2n+2$; the polynomial $p(x) = [(x-x_0)\cdots(x-x_n)]^2$ is always a counterexample. (Since $p(x)$ vanishes at $x_0, \ldots, x_n$, $\sum_{i=0}^{n} A_i p(x_i)$ vanishes, while $\int_{-1}^{1} p(x)dx$ is strictly positive.) Thus, the $2n+1$ of the theorem is best possible. It is not difficult to see, furthermore, that the only choice of $x_0, \ldots, x_n$ which gives a formula exact for all polynomials of degree not greater than $2n+1$ is the one given.

Not only is the integration formula based on the zeros of $P_{n+1}(x)$ exact for polynomials of as high a degree as possible, it also has the desirable feature that the weights A_i are all positive. To see this, let

$$l_i(x) = (x-x_0)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n), \qquad i = 0, \ldots, n.$$

Then $l_i(x_j) = 0$ for $j \neq i$ but $l_i(x_i) \neq 0$. Since $[l_i(x)]^2$ has degree $2n$,

$$\int_{-1}^{1} [l_i(x)]^2 dx = \sum_{j=0}^{n} A_j [l_i(x_j)]^2 = A_i [l_i(x_i)]^2,$$

so

$$A_i = \frac{1}{[l_i(x_i)]^2} \int_{-1}^{1} [l_i(x)]^2 dx > 0.$$

One useful consequence of the positivity of the weights is the error estimate

$$\left| \int_{-1}^{1} f(x)dx - \sum_{i=0}^{n} A_i f(x_i) \right| \leq 4e_{2n+1}(f).$$

(The quantities $e_m(f)$ were defined in Sect. 6 as

$$e_m(f) = \inf\{\|f - p\| : p \text{ a polynomial of degree not greater than } m\}.)$$

To prove the above estimate, we note that, if $p(x)$ is any polynomial of degree not greater than $2n + 1$,

$$\begin{aligned} \left| \int_{-1}^{1} f(x)dx - \sum_{i=0}^{n} A_i f(x_i) \right| &= \left| \int_{-1}^{1} [f(x) - p(x)]dx - \sum_{i=0}^{n} A_i [f(x_i) - p(x_i)] \right| \\ &\leq 2\|f - p\| + 2\|f - p\| \end{aligned}$$

(where we have used the exactness of the integration formula for $p(x)$ and the identity $\sum_{i=1}^{n} |A_i| = \sum_{i=1}^{n} A_i = 2$.) Taking the infimum over p gives the desired estimate. Since, as we have seen, the $e_m(f)$ converge rapidly to zero as m goes to infinity (provided that f is smooth), this result assures us that the Gaussian integration formulas give results which converge rapidly to the exact integral as the number of points increases. There is also a formula for the error of the form $c_n f^{(2n+2)}(\xi)$, but this formula involves such a high-order derivative of f that it is not very useful in practice.

For comparison with the example of a concrete application of Romberg's method discussed above, we report that using the ten-point Gaussian integration formula (with an obvious linear change of variables) to compute $\int_1^{20} dx/x$ gives a result which is off by about 5.3×10^{-4} (i.e., roughly the accuracy of Romberg's method with 32 panels), and the twenty-point formula makes an error of about 7.7×10^{-8} (substantially better than Romberg's method with 128 panels). This is not an atypical outcome. It is generally the case that a Gaussian integration formula gives better accuracy than a more straightforward method like Romberg's with the same number of function evaluations. On the other hand, Romberg's method, unlike Gaussian methods, provides not only approximations to the integral but also a lot of information which can be used to estimate the accuracy of these approximations. Furthermore, the number of panels used in Romberg's method can be increased mechanically until the desired accuracy has been attained (or at least appears to have been attained), whereas increasing the accuracy of a Gaussian method requires passing to a higher-order formula with a new (and independent) set of x_i's and A_i's. For most purposes, the fact that Romberg's method gives control over the error and the Gaussian methods do not makes Romberg's method more useful in spite of its usually producing less accuracy for a given amount of work.

8. Numerical methods for ordinary differential equations

We will consider here only *initial value problems* for ordinary differential equations. This is by no means the whole subject, but quite a different set of techniques is required to treat boundary value problems. We will generally write the equation we are considering as

$$\frac{dy}{dx} = f(x, y) \quad \text{with initial condition } y_0 \text{ at } x = a.$$

The exact solution to this equation will be denoted by $y(x)$. In nearly everything we say, it will make little or no difference whether y is a number or a vector, i.e., the theory we are going to develop applies to systems of differential equations as well as to single equations.

We begin with the simplest possible method, known as *Euler's method.* The prescription is as follows: Choose a step size h and write $x_i = a + ih$. Then generate a sequence y_i, $i = 1, 2, \ldots$ of approximations to the successive $y(x_i)$'s by

$$y_{i+1} = y_i + hf_i; \quad f_i \equiv f(x_i, y_i).$$

We can make a preliminary heuristic error estimate for this method as follows: If $y_i = y(x_i)$ (which is at least true for $i = 0$ and which we hope will be approximately true for general i)

$$\begin{aligned} y(x_{i+1}) &= y(x_i) + hy'(x_i) + (h^2/2)y''(\xi_i) \quad \text{for some } \xi_i \text{ between } x_i \text{ and } x_{i+1} \\ &= y_{i+1} + (h^2/2)y''(\xi_i) \end{aligned}$$

Thus, the new error in step i is $\mathcal{O}(h^2)$. To get from the initial point a to some given final point b takes $\mathcal{O}(h^{-1})$ steps so we would expect the total error in the computed $y(b)$ to be $\mathcal{O}(h^{-1}) \times \mathcal{O}(h^2) = \mathcal{O}(h) = \mathcal{O}(n^{-1})$ where n is the number of steps used to get from a to b. We will prove the correctness of this heuristic power-counting shortly. Since the global error is of the order of the first power of n^{-1}, we say that Euler's method is of *first* order.

The preceding analysis shows that Euler's method amounts to using the first two terms in the Taylor series for $y(x)$ about x_i to approximate $y(x_{i+1})$. This suggests trying to improve on Euler's method by using more terms in the Taylor series. In principle, this can be done in a straightforward way; repeatedly differentiation of the differential equation gives formulas for the successive $y^{(k)}(x_i)$ in terms of x_i, $y(x_i)$, and the partial derivatives of $f(x, y)$. These formulas, unfortunately, rapidly become extremely complicated—especially for systems—so this method is not very practical except for very simple $f(x, y)$'s.

There is a less obvious approach to getting higher order accuracy which avoids most of the disadvantages of the Taylor series method. Methods using this alternative approach are known collectively as *Runge-Kutta* methods. We will illustrate the idea by deriving the simplest of these methods, known as the *corrected Euler's method* or *Heun's method.* As usual, we will use $y(x)$ to denote the desired exact solution, and we will suppose that $y(x_i) = y_i$. What we want is an approximate formula for $y(x_{i+1})$. We argue as follows:

$$\begin{aligned} y(x_{i+1}) - y(x_i) &= h \cdot y'(x_{i+1/2}) + \mathcal{O}(h^3) \\ &= h \cdot 1/2 \cdot [y'(x_{i+1}) + y'(x_i)] + \mathcal{O}(h^3) \\ &= h \cdot 1/2 \cdot [f(x_{i+1}, y(x_{i+1})) + f(x_i, y_i)] + \mathcal{O}(h^3) \\ &= h \cdot 1/2 \cdot [f(x_{i+1}, y_i + hf_i) + f_i] + \mathcal{O}(h^3) \end{aligned}$$

(We have written f_i as an abbreviation for $f(x_i, y_i)$.) The first two equalities above use the fact that, in each case, the lowest power of h cancels by symmetry; the third is just the differential equation; the fourth uses $y(x_{i+1}) = y_i + hf_i + \mathcal{O}(h^2)$ and the fact that there is another factor of h outside the brackets.

This sequence of observations leads us to the following method: Put

$$\begin{aligned} k_1 &= hf(x_i, y_i) \\ k_2 &= hf(x_{i+1}, y_i + k_1) \\ y_{i+1} &= y_i + 1/2(k_1 + k_2) \end{aligned}$$

By what we have shown, if $y_i = y(x_i)$, then $y_{i+1} = y(x_{i+1}) + \mathcal{O}(h^3)$. By the same kind of power counting we used for Euler's method, we expect that the total error made in propagating from a to b with n steps of this method will be $\mathcal{O}(n^{-2})$, i.e., we expect this method to be of second order.

The $\mathcal{O}(h^3)$ term in the error can be written explicitly, but is quite complicated. Notice that the improved rate of convergence of this method, as compared to Euler's method, is purchased at a cost of having to evaluate $f(x, y)$ *twice* per time step, rather than just once.

This idea can be pursued further (although the algebra involved in deriving the formulas quickly gets very complicated). There is a widely used fourth-order method of this type, which we will refer to as the standard fourth-order Runge-Kutta method, given by the following rule:

$$\begin{aligned} k_1 &= hf(x_i, y_i) \\ k_2 &= hf(x_{i+1/2}, y_i + 1/2k_1) \\ k_3 &= hf(x_{i+1/2}, y_i + 1/2k_2) \\ k_4 &= hf(x_{i+1}, y_i + k_3) \\ y_{i+1} &= y_i + \frac{1}{6}[k_1 + 2k_2 + 2k_3 + k_4]. \end{aligned}$$

It can be shown that, if $y_i \doteq y(x_i)$, then $y_{i+1} = y(x_{i+1}) + \mathcal{O}(h^5)$, and so the global error should be $\mathcal{O}(n^{-4})$. So far as I know, the only way to derive this method or to prove that the local error is $\mathcal{O}(h^5)$ is by a long and untransparent computation. The method can be partially motivated, however, by observing that, in the trivial case where $f(x, y)$ depends only on x (so solving the differential equation amounts simply to evaluating an integral), it reduces to Simpson's rule. It is possible to write a formula for the h^5 term in the local error, but this formula is so complicated as to be essentially useless. The method given above is not the only fourth-order method of this type requiring only four evaluations of $f(x, y)$ per time step; in fact, there is a continuous family of such methods.

We have thus seen three methods of Runge-Kutta type:

- Euler's method which is first order and requires one evaluation of $f(x, y)$ per time step.
- Heun's method which is second order and requires two evaluations of $f(x, y)$ per time step.
- The standard fourth-order Runge-Kutta method, which requires four evaluations of $f(x, y)$ per time step.

It is not hard to find a third-order method requiring three evaluations of $f(x, y)$ per time step. Contrary to what this trend might lead one to expect, there is no fifth-order method requiring five evaluations of $f(x, y)$ per step, but there are fifth-order methods requiring six evaluations per step, and sixth-order methods requiring seven evaluations per step.

We will next give a simple but powerful argument for estimating the global error for a large class of methods which justifies the intuitive estimate used above that one power of n^{-1} is lost in going from local to global error estimates. The argument works for all *single step* methods, i.e., methods in which only y_i (and not $y_{i-1} \ldots$) is used to compute y_{i+1}. We will need the following notation: $y(x|\bar{x}, \bar{y})$ will denote the value at x of the solution of the differential equation with value $\bar{y}$ at $x = \bar{x}$, and we will write $y_i(x)$ for $y(x|x_i, y_i)$ The local discretization (truncation) error in the i-th step is defined to be

$$\tau_i = y_{i-1}(x_i) - y_i.$$

i.e., the difference between what the method gives for propagating from y_{i-1} at x_{i-1} to x_i and what would be obtained by propagating with the differential equation. We suppose we are given an initial point a, a terminal point b, and a number of steps n; we put $h = (b-a)/n$; $x_i = a + ih$. What we want to estimate is $y(x_n) - y_n \equiv y_0(b) - y_n(b)$. We start by writing this as

$$y_0(b) - y_1(b) + y_1(b) - y_2(b) + \cdots + y_{n-1}(b) - y_n(b).$$

Next:

$$y_{i-1}(b) = y(b|x_{i-1}, y_{i-1}) = y(b|x_i, y_i + \tau_i)$$

To see this: By definition $y_{i-1}(x)$ is the solution of the differential equation with the value y_{i-1} at $x = x_{i-1}$. But, as a solution of the differential equation, it can equally well be determined by giving its value at any other x; in particular at x_i; and its value at x_i is exactly $y_i + \tau_i$ by the definition of τ_i. Thus,

$$y_{i-1}(b) - y_i(b) = y(b|x_i, y_i + \tau_i) - y(b|x_i, y_i) = u_i \tau_i$$

where

$$u_i \equiv \frac{\partial y}{\partial \bar{y}}(b|x_i, \bar{y}) \quad \text{evaluated at } \bar{y} = \eta_i \text{ for some } \eta_i \text{ between } y_i \text{ and } y_i + \tau_i.$$

Inserting this into a previous formula gives

$$\text{global error } \equiv y_0(b) - y_n = \sum_{i=1}^{n} u_i \tau_i.$$

This formula can be given a very natural interpretation: The total error is the sum over the steps i of the new error in the ith step multiplied by the factor by which the differential equation itself multiplies small changes in initial condition in propagating from x_i to b. It also separates cleanly the effects of the numerical method from those intrinsic to the differential equation: the local discretization error τ_i is primarily determined by the method being used, while u_i is primarily a property of the differential equation. It is worth noting that this formula is extremely general: It does not require, for example, that the step size be constant, nor even that the same method is used for every step. Indeed, it actually assumes nothing at all about how the y_i are produced, and hence about the origin of the local error τ_i; in particular, the round-off error can be included τ_i and we then get an estimate on the total error including the effect of round-off.

There are a number of ways to use this formula to get simpler but less detailed error formulas. For example, if we just want to estimate the asymptotic behavior of the error for large n, and if we have an asymptotic formula

$$\tau_i \sim T(x_i, y_i) h^{p+1}$$

then

$$y(b) - y_n \sim h^p \times h \sum_{i=1}^{n} \frac{\partial y}{\partial \bar{y}}(x_i, y_i) T(x_i, y_i).$$

$$\sim (\frac{b-a}{n})^p \times \int_a^b \frac{\partial y}{\partial \bar{y}}(x, y(x)) T(x, y(x)) dx$$

Thus, as expected, a method with local discretization error of order h^{p+1} will have global error of order n^{-p}.

The methods discussed so far have all been single step methods, i.e., the determination of y_{i+1} involves only y_i and not $y_{i-1}\ldots$. We will now look at some multi-step methods. One of the attractions of multi-step methods is that it is easy to construct such methods of arbitrarily high order which require only one evaluation of $f(x,y)$ per time step. On the other hand, they are more complicated to use than single-step methods. On the theoretical side, multi-step methods, unlike single-step methods, have error-propagation rates which are different from those of the differential equation being solved. A careful theoretical analysis is needed to ensure that they do not give rise to a catastrophically rapid growth of error.

There are a great many multi-step methods. We will concentrate mostly on a simple but useful family of methods, known as *Adams-Bashforth* methods. Conceptually, the m-step Adams-Bashforth method (where $m = 2,3,\ldots$) is as follows: We suppose that we have computed $y_i, y_{i-1},\ldots$. We write f_j for $f(x_j, y_j)$, $j \le i$, and we let $p(x)$ be the polynomial (of degree not greater than $m-1$) interpolating

$$(x_i, f_i), (x_{i-1}, f_{i-1}), \ldots, (x_{i-m+1}, y_{i-m+1}).$$

Thus, if the y_j approximate the values of some solution $y(x)$ of the differential equation, $p(x)$ approximates $y'(x)$. We then take

$$y_{i+1} = y_i + \int_{x_i}^{x_{i+1}} p(x)dx.$$

To apply this method, it is convenient to note that it can be rewritten in the form

$$y_{i+1} = y_i + h[C_0^{(m)} f_i + C_1^{(m)} f_{i-1} + \cdots + C_{m-1}^{(m)} f_{i-m+1}]$$

where the coefficients $C_j^{(m)}$ are universal and can be computed in a straightforward way. Thus, for example, for $m = 4$,

$$y_{i+1} = y_i + \frac{h}{24}[55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}]$$

The first step in making an error analysis for a multi-step method is to decide what is the appropriate way to measure the local discretization error. The definition we are going to give may look unnatural at first, but it is justified by the analysis which can be based on it. The first surprise is that the local discretization error is a quantity associated with an *exact* solution of the differential equation, and not with a computed approximate solution. Thus, let $y(x)$ be any solution; write y_j for $y(x_j)$ and f_j for $f(x_j, y_j)$; and define the local discretization error in the $i+1$st step to be

$$\tau_{i+1} = y_{i+1} - \{y_i + h[C_0^{(m)} f_i + \cdots + C_{m-1}^{(m)} f_{i-m+1}]\}.$$

We have given a definition adapted to the Adams-Bashforth methods we are discussing, but it should be clear how to generalize it to other multi-step methods. The point of the definition is as follows: A multi-step method replaces solving the differential equation by solving a difference equation which approximates it; the local discretization error is the amount by which a given exact solution of the differential equation fails to solve the difference equation, or, better, the term which must be added to the right hand side of the difference equation to make the solution of the differential equation solve it also.

For the m-step Adams-Bashforth method, it is easy to derive a formula for the local discretization error by using the error formula for interpolation; the error can be written as

$$\tau_i = \gamma_m h^{m+1} y^{(m+1)}(\eta_i)$$

for some η_i between x_{i-m} and x_i, with

$$\gamma_2 = \frac{5}{12}; \quad \gamma_3 = \frac{3}{8}; \quad \gamma_4 = \frac{251}{720}; \quad \cdots$$

Thus, the m-step method has local error of order h^{m+1}. An analysis which is not difficult, but which uses some subtle properties of the method, shows that the global error for the m-step method is of order n^{-m}, provided that the starting values $y_1, \ldots, y_{m-1}$ are accurate enough. Thus, the m-step method, which requires only one evaluation of $f(x, y)$ per step, has order m, and m can be made as large as desired.

In practice, to solve a well-behaved differential equations with a desired accuracy of the order of one part in 10^8, methods of fourth or fifth order generally turn out to be optimal among Adams-Bashforth methods, and the fourth-order Runge-Kutta method described above typically requires roughly the same number of evaluations of $f(x, y)$ for a given accuracy as the fourth or fifth order Adams-Bashforth method. Since the Runge-Kutta method is easier to program—it requires no special provision for obtaining the starting values, and the step size can be adjusted easily as the solution proceeds—it is generally the method of choice. Adams-Bashforth methods of higher order can be very effective in computations aiming at higher accuracy. Finally, if the amount of computation to be done justifies a serious effort to maximize the efficiency of computation, it is likely to be advantageous to use one of the subroutines for solving differential equations available in standard subroutine libraries.

We alluded above to the fact that multi-step methods can have bad error-propagation properties. It will turn out that the error propagation properties of Adams-Bashforth methods are *not* bad. Nevertheless, we will introduce the analysis leading to this conclusion by looking at some methods which superficially look better than Adams-Bashforth methods but which it is unwise to use because of their error propagation properties.

We look first at a *very* bad method:

$$y_{i+1} = -4y_i + 5y_{i-1} + h[4f_i + 2f_{i-1}].$$

It is easily verified that this two-step method has local discretization error of order h^5(!). Trying it out on the simple problem of solving $y' = y$ with initial condition $y_0 = 1$ and, for definiteness, the exact value e^h for y_1, we get

with $h = .2$, $y_5 = 2.734$ which is correct within 1%

with $h = .1$, $y_{10} = -.127$, which has the wrong sign

with $h = .05$, $y_{20} = -1.6 \times 10^6$

This behavior is not difficult to understand. Applied to $y' = y$, the method leads to the linear difference equation

$$y_{i+1} = (-4 + 4h)y_i + (5 + 2h)y_{i-1}.$$

The general solution to this difference equation can be written as

$$y_i = c_1\beta_1^i + c_2\beta_2^i$$

with c_1 and c_2 arbitrary constants and β_1, β_2 the roots of

$$\beta^2 - (-4 + 4h)\beta - (5 + 2h) = 0,$$

provided that the roots of this equation are distinct. For small h, the equation is approximately

$$0 = \beta^2 + 4\beta - 5 = (\beta - 1)(\beta + 5)$$

so there is one root (β_1) near 1 and a second (β_2) near -5. The initial conditions force $c_1 \approx 1$ and $c_2 \approx 0$, but do not make c_2 *exactly* zero. Thus, $c_1\beta_1^i$ is a reasonable approximation to the desired solution of the differential equation, while $c_2\beta_2^i \approx c_2(-5)^i$ grows explosively; for small h and large n it completely dominates the computed solution, leading to the behavior reported above.

It is customary to refer to β_2^i as a *parasitic* solution of the difference equation, since its behavior does not at all correspond, for small h, to that of a solution of the differential equation which it is intended to solve. Multi-step methods inevitably give rise to parasitic solutions, since they approximate first order differential equations by higher order difference equations. The problem is to determine under what circumstances the parasitic solutions cause trouble.

The kind of strong instability displayed by the above example is not really dangerous since it is virtually guaranteed to be noticed. There are, however, less dramatic—therefore more pernicious—versions of instability. We illustrate on a simple and almost-practical method, called the *midpoint* method, and defined by:

$$y_{i+1} = y_{i-1} + 2hf_i.$$

Since, for any smooth function $y(x)$,

$$y(x_{i+1}) = y(x_{i-1}) + 2hy'(x_i) + \mathcal{O}(h^3),$$

this method has local discretization error $\mathcal{O}(h^3)$ and thus we would expect it to have global error $\mathcal{O}(n^{-2})$. This expectation is correct but, as we shall see, is not the whole story. We first look at two concrete examples, comparing the effectiveness of the midpoint method with that of Euler's method. The first example is to solve $y' = y$, with initial condition $y(0) = 1$, with step size .05; for the midpoint method, we take $y_1 = e^h$. The results are as follows: For $x = 1$, Euler's method is off by 2.4% and the midpoint method by .04%, while for $x = 10$ Euler's method is off by 22% and the midpoint method by .4%. Thus, so far, the midpoint method appears to be a clear improvement over Euler's method. We now redo the computation, except that we solve $y' = -y$ (and take $y_1 = e^{-h}$). For $x = 1$, the situation is more or less as before: Euler's method makes an error of 2.5% and the midpoint method an error of .05%. For $x = 10$, however, the situation has changed: Euler's method makes an error of about 23%, but the midpoint method give a result which is completely wrong—it is off by 477,000%!

The complete explanation for this behavior will require some analysis, but we will anticipate part of the result here: For the midpoint method, applied to solve the second example above up to x, we are going to derive the following asymptotic error estimate:

$$\text{global error } \approx -\frac{xh^2}{6}e^{-x} - (-1)^n \frac{h^3}{12}e^x.$$

We will refer to the first term on the right as the normal error term and the second as the anomalous error term. Since the anomalous term involves a higher power of h than the normal term, it will be negligible for small enough h. It may however be necessary to take h *ridiculously* small to get into this asymptotic regime if x is moderately large; for example, for $x = 10$, the two terms do not become comparable in size until $n \approx 2.4 \times 10^8$. For small but reasonable values of h, the fact that the anomalous error term contains a factor of e^x makes it much more important than might otherwise be anticipated.

We now proceed with the analysis. Applied to the differential equation

$$y' = \lambda y$$

with λ a constant, the midpoint method leads to the difference equation

$$y_{i+1} = y_{i-1} + 2\lambda h y_i$$

whose general solution is

$$y_i = c_1\beta_1^i + c_2\beta_2^i$$

with β_1, β_2 the roots of

$$\beta^2 - 2\lambda h\beta - 1 = 0,$$

(provided, as usual, that these roots are distinct). It follows immediately from the quadratic equation that

$$\beta_2 = \frac{-1}{\beta_1}.$$

Writing α for λh, we get, for the solutions of this equation,

$$\beta = \frac{2\alpha \pm \sqrt{4\alpha^2+4}}{2} = \pm\sqrt{1+\alpha^2} + \alpha.$$

Taking for β_1 the root with the $+$ sign, we get

$$\beta_1 = 1 + \alpha + \frac{\alpha^2}{2} + O(\alpha^4) = e^\alpha - \frac{\alpha^3}{6} + O(\alpha^4).$$

Thus,

$$\beta_1^i \approx e^{\lambda x_i}[1 - x_i\frac{\lambda^2 h^2}{6}]$$

so β_1^i is a good approximation to the correct solution to the differential equation. On the other hand

$$\beta_2^i = (-1)^i \beta_1^{-i} \approx (-1)^i e^{-\lambda x_i}$$

bears no resemblance at all to this solution. The constants c_1 and c_2 in the general solution of the difference equation are determined by the two initial conditions

$$c_1 + c_2 = 1, \qquad c_1\beta_1 + c_2\beta_2 = y_1 = e^{-\lambda h}.$$

Solving gives $c_1 = 1 - c_2$ and

$$c_2 = \frac{\beta_1 - e^{\lambda h}}{\beta_1 - \beta_2}$$

which reduces approximately to $-\lambda^3 h^3/12$ since $\beta_1 - \beta_2 \approx 2$ and $\beta_1 - e^{\lambda h} \approx -\lambda^3 h^3/6$. For $\lambda = -1$, combining the preceding formula for β_1^i with that for c_2 and dropping all but the largest terms gives the asymptotic error formula cited at the beginning of this analysis.

The result of the above analysis for general λ can be summarized as follows: The general solution to the difference equation obtained by applying the midpoint method to the problem $y' = \lambda y$ can be written as a linear combination of two particular solutions, one of which, β_1^i, resembles a solution of the differential equation and the other of which, the *parasitic* solution, does not. The initial conditions fix the coefficient of the good solution to be approximately one and that of the parasitic solution to be small but non-zero. The consequences of this situation depend on whether λ is positive or negative. For λ positive, the solution of the differential equation and the good solution of the difference equation grow whereas the parasitic solution decays, so the parasitic solution doesn't have much effect. This explains why the midpoint method worked well for $y' = y$. For λ negative, on the other hand, the good solution decays and the parasitic solution grows. Thus, in spite of its small coefficient, the parasitic solution can make an unexpectedly large contribution to the error at large x. Again, this is exactly what we saw for $y' = -y$.

Although the above analysis applies directly only to a trivial test problem, we can guess what will happen if the midpoint method is applied to compute a trajectory of some more general nonlinear differential equation. In regions where the solution being computed is unstable, i.e., where nearby orbits separate, accumulated error will grow because of the action of the differential equation itself. On the other hand, where the solution is stable, the properties of the numerical method will make the accumulated error grow although the differential equation itself tends to damp it out. Thus, unless the trajectory being computed is unstable everywhere, the net effect is likely to be that the overall error for a long computed trajectory is substantially larger than it would be for a single-step method with comparable local discretization error. Hence: Unless a great deal is known about the properties of the equation being solved, it is probably not a good idea to use the midpoint method.

The midpoint method, since it is only of second order anyway, would not be a very strong candidate for serious use even if it didn't have the problems just discussed. The same phenomenon occurs, however, for a method which used to be widely used, called Milne's method:

$$y_{i+1} = y_{i-1} + \frac{h}{3}[f_{i+1} + 4f_i + f_{i-1}]$$

This method can be derived by applying Simpson's rule to integrate $y'(x)$ from x_{i-1} to x_{i+1}.* It is a fourth-order method with a substantially smaller constant in its error estimate than the corresponding Adams method. Applied to $y' = \lambda y$ it gives the difference equation

$$(1 - \frac{\alpha}{3})y_{i+1} - \frac{4\alpha}{3}y_i - (1 + \frac{\alpha}{3})y_{i-1} = 0$$

* We remark that, unlike other methods discussed so far, Milne's method is *implicit*, i.e., y_{i+1} appears on the right as well as on the left. The subject of implicit methods is an important chapter in the theory of numerical methods for solving ordinary differential equations which we have had to omit for lack of time.

where we have written α for $h\lambda$. Thus, the general solution of the difference equation will be $c_1\beta_1^i + c_2\beta_2^i$, where β_1 and β_2 are the roots of

$$(1-\frac{\alpha}{3})\beta^2 - \frac{4\alpha}{3}\beta - (1+\frac{\alpha}{3}) = 0.$$

Notice that the product of the roots of this equation is $-(1+\alpha/3)/(1-\alpha/3)$. If we admit (as can easily be checked) that one of the roots is approximately e^{α}, then the other is approximately $-e^{-\alpha/3}$. Thus, exactly as for the midpoint method, the parasitic solution for Milne's method will cause no trouble for $\lambda > 0$, but it will grow, while the good solution decays, for $\lambda < 0$.

Let us now look at the fourth-order Adams-Bashforth method from this point of view. Applied to the standard test problem, $y' = \lambda y$, this method gives the difference equation

$$y_{i+1} = y_i + \frac{\alpha}{24}[55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}]$$

where, again $\alpha = \lambda h$. To find the general solution of this difference equation, we should look at the roots of

$$\beta^4 - \beta^3 - \frac{\alpha}{24}[55\beta^3 - 59\beta^2 + 37\beta - 9] = 0.$$

For small α, this equation is approximately $\beta^4 - \beta^3 = 0$, so there will be three roots near zero and only one near 1. In the language of the preceding paragraph, the root near 1 has to give rise to the good solution of the difference equation, so the parasitic solutions are produced by the small roots and hence damp out quickly and cause no trouble. (In this case, it is not apparent that the roots are all distinct for small α, so a little theory of linear difference equations with constant coefficients is needed to complete the argument.) Although we have for definiteness considered the fourth-order method, the same analysis applies to Adams-Bashforth methods of all orders.

To conclude this brief introduction to numerical methods for ordinary differential equations, we will say just a few words about a principal topic of current research interest in the area, the investigation of methods for what are called stiff differential equations. The term *stiff* is not precisely defined, but the general idea is that a stiff differential equation is one which has decaying transients which damp out on a time scale which is very short compared with the time scale over which the interesting behavior occurs. Examples arise frequently in applications to such subjects as the dynamics of chemical reactions (where the various reactions which need to be considered occur at very different rates) and of electronic circuits. Ratios of time scales as large as 10^6 are common in practice. The sorts of methods described above won't work at all on such problems unless the time step is taken to be of the order of the decay time for the fast transients; otherwise, the calculation explodes. What is needed for these problems is a method which gives at least a reasonable approximation to the correct solution when applied with a time step which is reasonably small compared to the slow time scales but may be very large compared to the fast time scales. A good test to see whether a method is going to do this is to apply it to the test problem $y' = \lambda y$ with $\alpha = h\lambda$ large and negative (or more generally with a large negative real part) and to see whether *all* solutions of the resulting difference equation decay (perhaps at a rate quite different from the rate of decay of the correct solution of the differential equation.)

An example of a method which has this property is the *trapezoid* method, defined by

$$y_{i+1} = y_i + \frac{h}{2}[f_{i+1} + f_i]$$

This method is implicit, so it is necessary to solve a (generally nonlinear) equation at each step. For the test problem, however, the equation is linear and easy to solve:

$$y_{i+1} = y_i + \frac{\alpha}{2}[y_{i+1} + y_i]$$

$$(1-\alpha/2)y_{i+1} = (1+\alpha/2)y_i$$

$$y_{i+1} = \left(\frac{1+\alpha/2}{1-\alpha/2}\right)y_i$$

so, finally,

$$y_i = \left(\frac{1+\alpha/2}{1-\alpha/2}\right)^i y_0.$$

Thus,

$$y_i \to 0 \text{ if and only if } \left|\left(\frac{1+\alpha/2}{1-\alpha/2}\right)\right| < 1$$

i.e, if and only if $\Re\{\alpha\} < 0$, which is exactly what one wants.

9. Fast Fourier Transforms

Perhaps the most important single advance in practical numerical computation in recent years has been the observation—fundamentally extremely simple—that that the calculation of Fourier transforms can be speeded up a great deal by organizing the computation in a clever way. Many other operations of practical importance—such as convolution (filtering) and computing time-averaged correlations—can be expressed in terms of Fourier transforms, so speeding up Fourier transformation has made available more efficient methods for these operations as well. We will sketch in this section one version of the reorganization of computation of Fourier transforms.

The mathematical operation we want to look at is the *discrete Fourier transform.* This means the following: We are given an integer N and N numbers $f_0, \ldots, f_{N-1}$, and want to compute the numbers

$$\hat{f}_k = \sum_{j=0}^{N-1} f_j e^{2\pi i j \cdot k/N}$$

for integer k. If we like, we can regard the formula as defining $\hat{f}_k$ for all k, but the function defined in this way is periodic with period N, so a complete determination of the Fourier transform requires the computation of N numbers, say $\hat{f}_0, \ldots, \hat{f}_{N-1}$. (The discrete Fourier transform is a particular case of the general Fourier transform for locally compact abelian groups, the group in this case being the additive group $\mathbf{Z}/N\mathbf{Z}$; we think of f, extended periodically, as a function on $\mathbf{Z}/N\mathbf{Z}$, and $\hat{f}$ as a function on the dual group.)

We will write

$$\omega_N = e^{2\pi i/N},$$

so

$$\hat{f}_k = \sum_{j=0}^{N-1} f_j (\omega_N)^{jk}.$$

If the powers of ω_N have been computed and tabulated beforehand, the computation of a single $\hat{f}_k$ (with $k \neq 0$) requires $N-1$ complex multiplications ($N-1$ rather than N because it is not necessary to multiply f_0 by ω_N^0) and $N-1$ complex additions; computing *all* the $\hat{f}_k$ in the straightforward way thus requires $N(N-1)$ complex additions and $(N-1)^2$ complex multiplications. In short: the number of operations required grows with N roughly like N^2.

Suppose, now, that we want to compute the discrete Fourier transform for a function defined at an even number of points. We will accordingly write the number of points as $2N$ rather than N. Splitting the sum defining the discrete

Fourier transform into even and odd terms, we get

$$\begin{aligned}\hat{f}_k = \sum_{j=0}^{2N-1} \omega_{2N}^{jk} f_j &= \sum_{j=0}^{N-1} \omega_{2N}^{(2j)k} f_{2j} + \sum_{j=0}^{N-1} \omega_{2N}^{(2j+1)k} f_{2j+1} \\ &= \sum_{j=0}^{N-1} \omega_N^{jk} f_{2j} + \omega_{2N}^{k} \sum_{j=0}^{N-1} \omega_N^{jk} f_{2j+1} \\ &\equiv \hat{f}_k^{(e)} + \omega_{2N}^{k} \hat{f}_k^{(o)},\end{aligned}$$

where we have used

$$(\omega_{2N})^2 = \omega_N.$$

What this formula shows is that a $2N$-point discrete Fourier transform can be calculated by calculating two N-point Fourier transforms (of $f_0, f_2, f_4, \dots$ and $f_1, f_3, f_5, \dots$) and combining the results in a simple way. Using this method, the number of arithmetic operations required to do a $2N$-point Fourier transform is only a little more than *twice* the number required for an N-point transform, rather than about *four times* the number as would be the case for the obvious method. This gain of a factor of about 2 is not, however, the really significant improvement; that is obtained when the above reorganization can be done repeatedly, and most notably when N is a power of 2. Let us work out the counting carefully in that case. We write $A(N)$ (respectively $M(N)$) for the number of complex additions or subtractions (respectively, multiplications) required to evaluate an N-point Fourier transform by this method; N is supposed to be a power of 2. Then

$$\begin{aligned}A(2N) &= 2A(N) + 2N \\ M(2N) &= 2M(N) + (N-1).\end{aligned}$$

(In counting multiplications, we have used the following remark: The term

$$\omega_{2N}^{k} \hat{f}_k^{(o)}$$

changes only by an overall factor of -1 when k increases by N, so it is only necessary to multiply $\hat{f}_k^{(o)}$ by ω_{2N}^k for $k = 1, \dots, N-1$; the result is then subtracted rather than added for $k > N$.) Solving these recursion relations with the initial conditions

$$A(1) = M(1) = 0$$

gives

$$A(2^n) = n2^n$$

and

$$M(2^n) = (n-2)2^{n-1} + 1.$$

i.e., both $A(N)$ and $M(N)$ are roughly proportional to $N \log(N)$ for large N.

The improvement in computational speed which is achieved by this reorganization is striking. To take a realistic example: Carrying out 2048-point discrete Fourier transform by the straightforward method requires about 4.2×10^6 each of additions and multiplications. By the above method, this is reduced to about 2.25×10^4 additions and 9.2×10^3 multiplications, so the total number of operations is reduced by about a factor of 263!

Although we have described the method only for N a power of 2, there is a straightforward generalization which works for N's which are products of small prime factors. There is also a different approach (see [11]) which works for N prime.

10. Floating point numbers*

It is a great convenience, in writing programs for numerical computations, not to have to be concerned in constructing the program about the size—within reasonable limits—of the numbers to be encountered. For this reason, nearly all programs for numerical computations manipulate numbers in *floating point* form, i.e., represented as the product of a number of order of magnitude unity by an integer power of some base (usually 2,10, or 16). While the general idea of a floating point representation is straightforward, there are many difficult choices to be made in deciding how many bits to use to represent a number, how bit strings of a given length are to correspond to numbers, and how the exact operations of arithmetic are to be approximated by operations on the finite set of numbers which can be so represented.

While it seems to be inevitable that any set of choices will produce at least some undesirable effects, some choices are definitely better than others. Up to now, different computer manufacturers have made different sets of choices and have thus constructed floating point number systems with different sets of anomalies. Independent of which of these systems is best, the lack of standardization is already undesirable since it has as a consequence that a program, written in a high-level language like FORTRAN and working correctly on one computer, may fail completely on another because of differences between the two floating point environments. The portability problem is a particularly serious obstacle to the construction of general subroutines for such tasks as finding the eigenvalues of matrices. Such subroutines need to be as robust as possible and so must anticipate and deal with the effects of anomalies in the floating point number system. In the absence of a standardized floating point environment, it has been necessary to resort to the wasteful expedient of developing different versions of these subroutines adapted to the particularities of the various computers on which they are to run.

Over the past half-dozen or so years, a working group of the IEEE (Institute of Electrical and Electronics Engineers) has developed a detailed specification for a binary floating point programming environment for microcomputers known as IEEE-P754 (the 'P' meaning that it is at this time a proposed standard which has not yet received final approval). This specification, if widely adopted, promises to alleviate many of the unfortunate features of the historical situation. In this section, I will discuss a few selected aspects of floating point number systems with particular emphasis on the choices made in the proposed IEEE standard. It is worth noting that the IEEE standard has gained considerable practical acceptance even before being formally approved; two commercially important product which conform quite closely to the standard are the Intel 8087 numerical

*It is a pleasure to record here my indebtedness to W. Kahan for many informative discussions on floating point number systems and the IEEE standard.

processor (a coprocessor for the 8086 and 8088 CPU chips)* and the floating point arithmetic provided in software for the Apple Macintosh.

To understand how floating point numbers are represented inside the computer, it is helpful to consider first the representation of decimal numbers in a calculator. A typical calculator manipulates numbers of the form

$$\pm d_0.d_1...d_9 \times 10^{\pm e_0 e_1} \text{ with } d_0, \ldots, d_9, e_0, e_1 \in \{0, \ldots, 9\} \text{ and } d_0 \neq 0.$$

Thus, the memory required to store a number is 12 digits plus two bits to represent the overall sign and the sign of the exponent. The requirement that d_0 be non-zero, i.e., that the number be *normalized* , fixes the position of the decimal point and thus eliminates a non-uniqueness in the representation of numbers with fewer than ten digits. The number zero, of course, does not have a normalized representation and so has to be treated separately.

The proposed IEEE standard specifies formats for representing numbers in three different precisions, called single, double, and extended precision. In a first approximation, to be corrected later on, the single precision format is as follows: Four bytes (i.e., 32 bits) are used to represent a number. The non-zero numbers which can be represented are those having a binary representation of the form†

$$\pm d_0.d_1, \ldots, d_{23} \times 2^e \quad \text{with } d_0, \ldots, d_{23} \in \{0,1\},\ d_0 \neq 0, \text{ and } -126 \leq e \leq 127.$$

We will refer to numbers which can be written in this form (together with 0 and a few others to be specified later) as the *representable* numbers in the IEEE single precision format. Thus, the largest representable number is

$$1.1 \cdots 1 \times 2^{127} = (2 - 2^{-23}) \times 2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}$$

and the smallest strictly positive representable number is

$$1 \times 2^{-126} \approx 1.18 \times 10^{-38}$$

If we define the *precision* of the format as the maximum relative spacing between two successive non-zero representable numbers, then the precision is $2^{-23} \approx 1.19 \times 10^{-7}$, i.e., roughly seven decimal digits. This is not enough for most kinds of scientific and engineering computations, but single precision numbers

* The proposed IEEE standard has gone through a number of drafts; the current one (which appears to have a good chance of being the last) is numbered 10.0. The Intel 8087 actually conforms very closely to draft 8.0 which differs in a number of details from draft 10.0.

† In this context, $d_0.d_1 \cdots d_{23}$ denotes $d_0 + d_1 2^{-1} + d_2 2^{-2} + \cdots + d_{23} 2^{-23}$; the d_i are each 0 or 1.

are nevertheless quite useful for applications where speed and compactness are important, and which need only limited accuracy, such as signal processing and some kinds of data analysis and Monte Carlo computations.

The IEEE standard specifies not only what numbers are representable in the single precision format but also exactly how they are to be represented as sequences of 0's and 1's. The string of 32 bits available to represent the number is split into three substrings (fields):

1. A single-bit sign field: The bit is 0 for positive numbers and 1 for negative ones.
2. An eight-bit exponent field: To understand how the exponent is represented, it is convenient to start by regarding the contents of this field as representing an unsigned integer, which then lies between 0 and 255 (inclusive). This unsigned integer is referred to as the *biased* exponent. The values 0 and 255 for the biased exponent are reserved for special purposes (including representing zero); the other values, from 1 through 254, are mapped to ordinary exponents by subtracting an *exponent bias* of 127:

   $$e = \text{biased exponent } - 127.$$

3. The remaining field of 23 bits, referred to in the IEEE standard as the *significand* field, which holds $d_1, \ldots, d_{23}$. Since we are working with normalized numbers, d_0 is always one and so it does not need to be recorded. This representation is thus said to have a *hidden bit.*

As already indicated, the bit patterns with all zeros or all ones in the exponent field are not used in the ordinary way to represent numbers. One of the reserved bit patterns is needed to represent zero, and the IEEE standard specifies that the pattern with all 32 bits zero is used for that purpose.

Using the terminology introduced to describe the single-precision format, we can give an economical description of the IEEE double-precision format: The normalized representable numbers are those of the form

$$\pm d_0.d_1, \ldots, d_{52} \times 2^e \quad \text{with } d_0 = 1 \text{ and } -1022 \leq e \leq 1023.$$

As with single precision numbers, a single bit suffices to represent the sign. Eleven bits specify the exponent; these bits, interpreted as an unsigned integer, determine a biased exponent with bias 1023. Allowed values of the biased exponent are in the range from 1 to 2046; again, the smallest and largest values 0 and 2047 are reserved for special purposes. The remaining 52 bits simply hold $d_1, \ldots, d_{52}$; the bit d_0 is, as in the single precision format, implicitly understood to be 1. The largest number representable in double precision format is approximately $2 \times 2^{1023} \approx 1.80 \times 10^{308}$; the smallest positive number is $1 \times 2^{-1022} \approx 2.23 \times 10^{-308}$; and the precision (as defined above) is $2^{-52} \approx 2.22 \times 10^{-16}$.

The IEEE standard also provides for a third format, called *extended*, offering higher precision and a greater range of magnitudes than does the double precision format. Unlike the single and double precision formats, the extended format is not completely specified; instead, it is required to satisfy a number of conditions. Rather than to summarize that list of conditions, we will describe a format which meets the conditions and which is used by the Intel 8087 numerical processor; it is referred to in Intel documentation as the *temporary real* format. In this format, 10 bytes (i.e., 80 bits) are used to represent a number. As usual, one bit represents the sign. Fifteen bits represent the exponent in the same sort of biased representation as in single and double precision. The exponent bias is $16383 = 2^{14} - 1$, providing a range of exponents from $-(2^{14} - 2)$ to $2^{14} - 1$; the smallest and largest biased exponents are again reserved. The remaining 64 bits hold the fraction; this time, there is no hidden bit, i.e., the leading 1 bit is represented explicitly. In this format, the largest representable number is approximately $2^{16384} \approx 1.19 \times 10^{4932}$; the smallest is $2^{-16382} \approx 3.36 \times 10^{-4932}$ and the precision is $2^{-63} \approx 1.08 \times 10^{-19}$. Having available such an extended format simplifies many things. Here are two simple examples:

1. Computing elementary transcendental functions via, e.g. rational approximations, with nearly full double-precision accuracy; say with error less than one unit in the last place. Finding a sufficiently accurate approximation is only part of the problem; it is also necessary to devise a way of evaluating this approximation with sufficiently small round-off error. With only double-precision arithmetic, this may be extremely difficult or impossible, but it becomes easy if the calculations can be done in extended precision and the result then rounded back to double precision.
2. It is surprisingly difficult to write a program for evaluating Euclidean norms of vectors which is reasonably efficient and which works in all cases where the vector elements are all representable numbers and the Euclidean norm itself does not overflow. The problem is that the *squares* of the vector elements may be either too big or too small to bè representable numbers. This difficulty disappears if the intermediate calculations can be done in extended precision with its wider exponent range.

The great complication in designing a floating point computational environment is that the finite set of representable numbers in any particular format is not closed under the elementary arithmetic operations.* The exact result

* The IEEE standard, incidentally, deviates from standard practice in including the extraction of square roots in the list of elementary arithmetic operations along with addition, subtraction, multiplication, and division. This choice reflects a judgment that it is advantageous to include square roots among the operations performed in hardware rather than leaving them to be performed in software. On the one hand, in a number of important kinds of computation,

of an arithmetic operation performed on representable operands may fail to be representable for a variety of reasons:

1. It may have too many digits.
2. It may be larger in magnitude than the largest representable number (overflow).
3. It may be smaller in magnitude than the smallest positive representable number (underflow).
4. The requested arithmetic operation may itself be invalid (e.g., division by zero, square root of a negative number).

Although these occurrences are all referred to as *exceptions*, they have quite different characters. The first, in contrast to the others, should be regarded as normal in floating point computation; it will happen in most operations, and the user generally does not want to be informed about it. The IEEE standard specifies a default response: The result returned is to be the representable number nearest to the mathematically exact result, with the further stipulation that, when the exact result is mid-way between two adjacent representable numbers, the result returned is the one with low-order bit zero. This unambiguously specifies the result of any valid requested operation which does not produce underflow or overflow. It is not obvious that this specification can be met efficiently, but the fact is that it can. Note that this specification implies that if the exact result of a requested operation is representable, then the returned result will be the exact one, and that the operations of addition and multiplication are strictly commutative. (They are not, however, associative. For example, if ϵ is a sufficiently small positive number, then the result of adding ϵ to 1 will be exactly 1, so

$$(-1) + (1+\epsilon) = (-1) + 1 = 0 \neq \epsilon = ((-1)+1) + \epsilon.)$$

In the absence of underflow and overflow, the approximate result obtained for any valid individual arithmetic operation performed as specified by the IEEE standard can be written as the mathematically exact result multiplied by a correction factor $1+\eta$ where η depends in a complicated way on the operation and the operands but is always small:

$$|\eta| \leq u$$

where

$$u = \begin{cases} 2^{-24} \approx 5.96 \times 10^{-8} & \text{in IEEE single precision,} \\ 2^{-53} \approx 1.11 \times 10^{-16} & \text{in IEEE double precision,} \\ 2^{-64} \approx 5.42 \times 10^{-20} & \text{in the Intel temporary real format.} \end{cases}$$

square root extractions are about as frequent as divisions. On the other, circuitry for doing division is usually easy to adapt to do square roots as well, and the time required for a square root in hardware is about the same as for a division, whereas square-roots done in software are much slower.

By repeated use of these estimates, it is in principle possible to estimate rigorously the accuracy of any given calculation. In practice, these estimates prove to be too complicated to be of any use except in simple situations. There is a variant on the brute-force approach to estimating accumulated round-off error, known as *backwards error analysis*, in which the above elementary error estimates are used repeatedly to show that the final result of a given calculation is the exact answer to a perturbed problem, or, more precisely, to estimate *how much* the original problem must be perturbed to give one which the computed result solves. This technique works very well, for example, in estimating the effects of round-off error in the numerical solution of systems of linear equations.

By contrast to the routine occurrence of excess digits, the occurrence of an overflow, underflow, or invalid operation is a unusual and catastrophic event which requires special action. One acceptable response is simply to stop the computation with an error message giving as much information as possible about the circumstances which led to the exception. A better alternative is to give the user at least the option of specifying an error-handling procedure to be invoked whenever an exception occurs. Sometimes, however, it may be more useful to provide some sort of result and proceed with the computation. The philosophy for dealing with exceptions adopted by the IEEE standard is to give the user the option of specifying an error handling routine for each kind of exception but to provide a default action allowing the computation to continue if no error handling procedure has been specified. The default actions are all completely specified in the standard.

Let us look first at the case of overflow. For definiteness, we will discuss the double precision format. Recall that in this format the exponent field can represent biased exponents from 0 to 2047, but that only values from 1 through 2046 are used to represent ordinary numbers. The bit patterns with biased exponent 2047 and significand field 0 are used to represent $\pm\infty$. The default action on encountering an overflow is to return ∞ with the appropriate sign; the same action is taken when a non-zero number is divided by zero. It is natural to extend certain arithmetic operations to allow infinite operands, as, for example, $x + \infty = \infty$ and $x/\infty = 0$ for x any finite number. The number system is enlarged to allow these operations. Doing this, however, introduces a number of new invalid operations such as $\infty - \infty$ and ∞/∞. This fix-up strategy has some odd side effects. For example, if x is a normalized number large enough so that $2 \times x$ overflows, then $x/(2 \times x)$ will be computed to be zero.

The treatment of underflow is more complicated. The traditional default action in response to underflow is to return zero. This is often a good thing to do, but not always. For example, if x and y are two distinct small representable numbers such that $x - y$ underflows, then $x - y$ will be computed to be zero even though x is different from y. This can cause trouble in a program which tests whether $x \neq y$ before dividing something by $x - y$. This trouble can be fixed by defining the test for $x = y$ to mean that evaluating $x - y$ produces zero, but there will then be triples x, y, z such that $x = y$ and $y = z$ but $x \neq z \ldots$

Perhaps the most controversial aspect of the IEEE standard is that it specifies another, and more complicated, default response to underflow. First of all, some new (very small) representable numbers are introduced, represented by bits patterns with biased exponent 0. In the double precision format, the bit pattern with biased exponent zero and significand field $d_1, \ldots, d_{52}$ is taken to represent the number

$$\pm 0.d_1 \ldots d_{52} \times 2^{-1022}$$

(whereas the same significand field, with biased exponent one, represents

$$\pm 1.d_1 \ldots d_{52} \times 2^{-1022}.)$$

These new representable numbers are called *denormalized* numbers (but they are in fact as good as any others except that their relative spacing is less fine than for normalized numbers.) In particular, the bit patterns with biased exponent zero and $d_1, \ldots, d_{52} = 0$ represent zero. Either value of the sign bit is allowed, so there are (contrary to what was said above) two representation for zero, one with a plus sign and one with minus. These two representations are equivalent for most purposes, and the test for equality does not distinguish between them, but a strictly positive number gives $+\infty$ when divided by $+0$ and $-\infty$ when divided by -0.

With these new representable numbers available, the IEEE standard specifies that any valid arithmetic operation whose exact result does not overflow is to return the representable number nearest the exact result, with the usual "round to even" prescription in the event of a tie. Then comes a small but useful twist: In traditional terminology, one would say that the result of an arithmetic operation underflows if the result produced by the above prescription is denormalized, i.e., if the exact result is not zero but is smaller in modulus than $(1 - 2^{-53}) \times 2^{-1022}$. A little reflection shows, however, that the only reason to call attention to a denormalized result is that the process of rounding an exact result to the nearest representable number can cause a larger relative error if the result is denormalized than is possible if it is normalized. There is therefore no need to signal an underflow when the exact result is a representable number, even if that number is denormalized. Accordingly, the IEEE standard redefines underflow: The result of an arithmetic operation is said to underflow only if both

a. The exact result of the operation is not representable.

b. The representable number obtained by rounding the exact result is denormalized.

By this definition, subtraction cannot produce an underflow.

While the denormalized-numbers scheme deals in a very smooth way with the anomalies arising from underflows, it does not eliminate all of them; for example, there are non-zero numbers x for which $x/2$ is computed to be zero.

In closing this section, I want to stress that the discussion given here of the IEEE floating point arithmetic standard is far from complete and also, on some topics, simplified to the point of inaccuracy. I highly recommend a careful study of the official text of the proposed standard to the reader who wants to understand in detail the subtleties of floating point computation.

11. Suggestions for further reading.

Computers. Reference [9] is a general textbook on computer structure and organization which I have found particularly useful. The September 1977 issue of *Scientific American*, which is devoted to microelectronics, contains a number of excellent articles. For more details on computer hardware, see [6]. Reference [8] gives an up-to-date survey of the current status and and probable future evolution of microelectronic technology. The definitive reference on the architecture of the 68000 is [7].

Numerical analysis. There are a great many beginning textbooks. Four which I have found useful are listed as references [2], [3], [4], and [10]. For the numerical solution of ordinary differential equations, see [5].

Floating point arithmetic. At the time these notes were written, Draft 10.0 of the proposed IEEE binary floating point standard had not been published. The now-obsolete draft 8.0 is published in [12], and a draft of a more general standard (IEEE P854) in [1].

12. Bibliography.

1. W. J. Cody, J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson, A proposed radix- and word-length-independent standard for floating point arithmetic, *IEEE Micro* **4** (1984)#4, 86–100.
2. S. D. Conte and C. de Boor, *Elementary Numerical Analysis, an Algorithmic Approach*, 2nd ed. (McGraw-Hill, New York, 1980).
3. G. Dahlquist and A. Bjork, *Numerical Methods* (Prentice-Hall, Englewood Cliffs, 1974).
4. G. E. Forsythe, M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations* (Prentice-Hall, Englewood Cliffs, 1974).
5. C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations* (Prentice-Hall, Englewood Cliffs, 1971).
6. C. Mead and L. Conway, *Introduction to VLSI Systems* (Addison-Wesley, Reading, 1980).
7. Motorola Inc., *M68000 16/32-Bit Microprocessor Programmer's Reference Manual*, 4th ed. (Prentice-Hall, Englewood Cliffs, 1984).
8. C. L. Seitz and J. Matisoo, Engineering limits on computer performance, *Physics Today* **37** (1984)#5, 38–45.
9. A. S. Tanenbaum, *Structured Computer Organization*, (Prentice-Hall, Englewood Cliffs, 1976).
10. B. Wendroff, *Theoretical Numerical Analysis* (Academic Press, New York, 1966).
11. S. Winograd, On computing the discrete Fourier transform, *Math. Comp.*, **32** (1978), 175-199.
12. (no author listed) A proposed standard for binary floating-point arithmetic, *Computer* **14** (1981)#3, 51-62.