NUMERICAL METHODS, OPTIMIZATION TECHNIQUES AND COMPUTER PROGRAMMING

(DIB02)

(PG DIPLOMA)

ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

Lesson 2.1.1

PARALLEL COMPUTERS

Contents

2.1.1.1 Introduction
2.1.1.2 Obejctive
2.1.1.3 Parallelism
2.1.1.4 First Generation
2.1.1.5 Trnaspeuter
2.1.1.6 Multitasking
2.1.1.7 Stack-Based Operation
2.1.1.8 Issues
2.1.1.9 Application
2.1.1.10 Self Assessment
2.1.1.11 Summary
2.1.1.12 References

2.1.1.1 Introduction

2.1.1.1.1 Pre-History

In 1821, Charles Babbage proposed and developed a prototype of his Difference Engine designed to automatically generate successive values of a polynomial. He went on the design the Analytical Engine which was to be a generall purpose computer. Each of these designs were examples of parallel computer architecture, with results for successive computations being constructed whilst the preceeding results were delivered. When, in 1943, the code-breaking ENIAC was introduced, it also showed some aspects of parallel architecture in its design. However, for both Babbage and the designers of ENIAC, the technology was not adequate for the day-to-day realisation of these designs. EDVAC and
successive digital computers followed the sequential architecture laid down by Von Neumann.

2.1.1.1.2 History

Parallel Computers actually have a long history, although it was not until the late 1960's that practical digital parallel computers were introduced, and until the mid 1980's that they gained wider penetration. To a large extent their lack of penetration can be put down to cost, although some blame must be attributed to the difficulty of developing code for these machines. Each architecture was so different from the next that it was difficult to exploit the potential power of the machines without re-writing the code to use the architecture adequately. This, in turn, made programs non-portable, and even now the problem of writing programs for parallel computers has not gone away.

Although the price of parallel hardware has fallen, and degrees of parallelism can be found in most modern desktop PC's, there are still diverse forms of parallel hardware implementation. The next chapter examines the various hardware forms in more detail, but in this section we will provide a quick history of

developments to give a flavour of what is available. The Virtual Museum of Computers provides a more in-depth look at the development of computers in general, and provides links to sites which provide a look at the history of Parallel Computers.

2.1.1.2 Objective

- To know what parallelism of computers would be
- To understand the phenomenon of multitasking and stack based operation.
- To be aware of the issues of parallellism and its applications

2.1.1.3 What is Parallelism ?

As humans we tend to work in three modes :

Sequence

As individuals, we perform one job after the completion of the previous job. For example, in manual labour, we will dig a hole before attempting to pour the concrete which will be used to create the foundations for a house.

Concurrent

We must, at times, be able to keep a number of jobs going at once. For example, we would have to put the aggregates into the cement mixer part-way through digging the hole for the foundations, and turn the mixer on. We would then periodically need to re-visit the cement mixer to check on the mixing process, while still in the process of digging the hole. To a casual observer it would look as though both jobs were being done at the same time, but in fact one person is alternating between the jobs.

Parallel

In building a house, there are some jobs which are completed signifigantly faster if more than one person can be involved. We may, if finances were available, allocate a number of workers to all dig the foundations at once.

It would be helpful to have equivalent modes of operating within computing. In this section we will examine this possibility.

Parallel

Parallel execution is the controlled, cooperative execution of an identifiable subset of the components of a program (statements, subroutines, objects) across a number of distinct processing elements which are interconnected to provide control and communication, in order to solve a distinct computationally time-complex problem in a timely manner.

This definition must, however, be qualified, since it permits a wider area of study than we are interested in here. Almasi and Gottlieb (1994) mark out a useful perimeter for us :

We are interested in computers that use a high degree of parallelism to speed the computation required to solve a single large problem. This leaves out much of the world of business programs, where I/O rather than computing power is typically the bottleneck. It leaves out most commercial multi-processors, whose added processing units are used to increase the number of jobs that can be handled at a time, rather than to speed up a single job (in other words, to improve throughput rather than turnaround time). Strictly speaking, it also leaves

out distributed systems such as networks of personal workstations, because, although the number of processing units can be quite large, the communication in such systems is currently so slow that it prevents large speedups in one job.

2.1.1.4 The First Generation

In 1969 the ILLIAC IV was introduced, followed soon after by the CDC7600. The ILLIAC IV was an Array Processor - it fetched an array of data, performed one operation on all elements of the array (up to 63 elements at a time), and then stored the result back again. The CDC7600 was, in contrast, a Pipelined Processor - it fetched a vector of data, one element at a time, and put the element into a pipeline of processors, each of which performed an operation on the data item in succession. Speed-up was achieved by filling the pipeline with many items of data moving through the pipeline at the same time, each item at a different stage in the pipeline. NASA's ILLIAC IV was retired in 1976 when the Cray-1 - the first supercomputer - was introduced. This computer was also a pipelined processor, but with many pipelines and an architecture which used every available advance in technology to ensure that the pipelines and data delivery to and from the pipelines were all as fast as possible. Successful though the Cray-1 was, none of these machines could be regarded as truely general purpose - they were all specialised back-end processors to a front-end sequential machine. This was not to hinder the development of other specialised architectures, however, such as the ICL DAP (1979) which was a 1024 1 bit processor array, the Goodyear MPP (1983) - a 132 x 128 1 bit array processor, or the Connection Machine (CM-1 and CM-2 in 1986/7) providing 65535 1 bit processors with FPUs.

However, the search was also on for a way to create a parallel computer from standard processing units. The C.mmp was an early attempt (1977) which used 16 PDP-11 processors, whilst the BBN Butterfly sought to connect 256 M68000 cpus using a multi-stage network. The speed and cost of the network, and the computational effort needed to maintain the communications were all limiting factors which resulted in some novel architectures in an attempt to bypass the problem. The cm* was a tree structured architecture that limited communication requirements to four connected processors, but allowed expansion by adding nodes to the tree.

2.1.1.5 From the Transputer

The Inmos transputer (1985) changed things dramatically by placing communication processors on the chip with the main processor so that the processor was not hindered by having to maintain communications whilst performing other operations. Furthermore, it used cheap serial lines driven at high speed to reduce the cost of interconnection. The development of the higher specification T-800 in 1988 opened the way to the development of powerful multiple processor parallel machines, such as the Meiko Computing Surface. The Parsys Supernode attempted to tackle the problem of having to use a fixed architecture in earlier machines by providing a degree of reconfiguration.

In the meantime, other manufacturers were adding similar capabilities to their more powerful CPUs, allowing (for example) the development of the Alliant FX/2800 (1990) which connected 28 Intel i860 processors, and the CM-5 (1992) which uses 1024 SPARC processors. The latest Cray T3D (1993) follows the same line, using 2048 DEC Alpha processors.

In 1994 Inmos announced the development of the T9000 transputer which helps to remove the problem of routing and the requirement to rewrite programs to a particular architecture by providing a virtual routing capability. Although this processor has yet to gain acceptance, this technology represents another step to the

creation of a practical parallel architecture. In the meantime, the advent of fast networking technology and switches has allowed the development of 'virtual' parallel machines from networks of workstations using software technologies such as MPI, PVM, or Corba. Whilst still requiring large granularity processes with limited communication in order to achieve reasonable speed-up, it has opened up access to parallel programming to the wider programming community.

The Processor

The design of the processor part of the transputer is very interesting. Although you may be tempted to skip directly to the programming section, hold on for awhile. Some of the questions that will arise later when we start Faswriting parallel tasks will be readily answered if we know a few things about the processor hardware. Also, it is important to know where the performance of the processor comes from. We will see that several key design choices contribute to the transputer's overall performance:

No dedicated data registers: The transputer does not have dedicated registers, but a stack of registers, which allows for an implicit selection of the registers. The net result is a smaller instruction format.

Reduced Instruction Set design: The transputer adopts the RISC philosophy and supports a small set of instructions executed in a few cycles each.

Multitasking supported in microcode: The actions necessary for the transputer to swap from one task to another are executed at the hardware level, freeing the system programmer of this task, and resulting in fast swap operations.

2.1.1.6 Multitasking

The process by which a processor divides its time among several programs (seemingly) running at the same time is called multitasking. Multitasking is an important way to improve the performance of a processor by allowing it to start running another program if the one that it is currently executing cannot continue for awhile. A typical reason for this to happen is that an Input/Output operation is necessary but the peripheral, or I/O controller is not available. By allowing the processor to put the first program aside and switch to another one the total execution time of both is reduced.

Multitasking is the first form of parallelism supported by the transputer, and is accomplished by having the processor maintain a list of tasks that must be executed. Each task executes for a small amount of time, called a quantum, before it is stopped, and swapped for another task (if one awaits). For the transputer, a quantum of time is typically on the order of two milliseconds (2 10-3 seconds)[1]. Task are executed in a round-robin fashion. When a task eventually terminates it is removed from the list. At any time, new tasks may be created and added to this list.

At any time a transputer task may be in either one of two states:

Active

This state refers to a task that is being executed, or in the list of tasks waiting to be executed.

Inactive
This state refers to a task that is not in the list of active tasks, because either one of three conditions is preventing it from continuing execution:

The task is waiting for an input from one of the I/O ports
the task is waiting to output to one of the I/O ports, or
the task has been asked to stay idle until a specified time in the future.

Active tasks
The transputer maintains the active tasks chained in a linked list, and two of its internal registers are used to point to the front and rear of the list. The actual list is stored in memory, and the registers contain the memory address of the cells defining the tasks. To further increase the flexibility and power of the multitasking environment, the transputer implements two levels of priority for the tasks:

High priority tasks (level 0):
These tasks indeed have a high priority: Once they gain control of the processor, they continue executing until they complete, or until they need to transfer information over a serial link. In this case, the task is said to be blocked waiting for a link. These tasks thus enjoy an unlimited quantum, and, as a result, only tasks that are expected to run for short periods of time are given high priority.

Low priority tasks (level 1):
These tasks run whenever there are no high priority tasks active, and run for up to one quantum of time, switching in a round robin fashion. Most user-defined tasks will be low-priority.

Hence the transputer needs to maintain not one, but two linked-lists, one for low-priority and one for high-priority tasks, and uses a total of four registers to point to the front and rear of both lists.

Fast task-switching
The switching between tasks belonging to the same priority-list or to different priority-lists is handled directly by the processor, and all register updates are controlled internally, through the microcode. Removing this action from a software kernel renders this operation extremely fast: less than 1 us typically [INMO88b].

Inactive tasks
Sometimes, active tasks encounter a situation where they cannot continue running until some external event occurs. In such cases the processor deschedules the , which becomes inactive. The external event is typically the expiration of a quantum, a timer event, or an event related to information transfer over a link.

When a task changes from active to inactive, it is removed from its associated linked-list and placed in what the Inmos documentation refers to as the workspace, which is a reserved area of memory. Because inactive tasks are removed from the link-lists, the processor does not suffer any overhead from them when it scans the lists to find the next task to run.

Multitasking Rule The instantaneous performance of a transputer is not affected by the number of inactive tasks that may exist at any given time

We will refer to the inactive tasks that are waiting for communication as blocked tasks. We will explore this concept when we look at the communication process in Section 2-4. The inactive tasks that await a specified time, though, merit some attention right now.

Timers and inactive tasks

The T800 contains two 32-bit timers. The timers are separate entities outside the processors, as shown in Figure 2-2. Each timer is associated with a priority. One timer, available to high-priority tasks, is incremented every microseconds, and cycles through all its 32-bit states in 4295 seconds (232 s), or 71 minutes and 35 seconds. The other timer is associated with low-priority tasks and is incremented every 64 microsecond, resulting in a 76-hour long cycle.

Each timer can be viewed as two registers. One, called Timer, incremented at every tick of a clock, the other, TNextReg, defining a time in the future when some event must occur.

The following scenario explains how these counters come into play. Assume that a task is designed to check the status of an I/O controller at regular intervals, say 20 times a second. Upon checking the controller and having performed its function, the task can deschedule itself, that is voluntarily change its status from active to inactive, while simultaneously specifying a time in the future when it should be rescheduled. It does so by reading the contents of Timer and by adding a time delay to it. This new value represents the time in the future when the tasks wants to be awakened, and is stored in the TNextReg register by the processor. When the contents of Timer reaches the count in TNextReg, an event occurs (interrupt) and the processor puts the task back in the list of active tasks.

More precisely, when a task performs such a descheduling action, it is removed from the list of active tasks and added to a timer queue, located in memory. If that task requested a "wake-up" time earlier than any of the times associated with the tasks in that queue, then that time is stored in the TNextReg register. When the task becomes active again, it is placed at the end of the list of active tasks, and because this list may not be empty, the task may not start right away.

The capability for tasks to determine the exact time at which they are activated is a useful feature in context where synchronization of tasks is important, for debugging purposes, or when reliability is an issue, and watch-dog timers and time-out detection must be implemented.

2.1.1.7 Stack-based operation

The processor contains six registers. Three (the A, B, and C registers) are used as data registers and implement a stack. If you ever used a Hewlett-Packard calculator, you are already familiar with the reverse polish notation, and the simplicity and efficiency associated with its use. Here is how the processor evaluates the following high-level expression:

x = a+b+c;

where x, a, b, and c represent integer variables. With an evaluation stack, there is no need to specify which processor registers receive the variables. Here, the processor is simply told to load each variable, one after the other and to add them. As variables are loaded they are pushed into the stack. Every time an operation is

performed, the two values at the top of the stack are first popped, then combined, and the result of the operation is pushed back onto the stack:

```
            ;stack contents (=Undefined)
            ;[              ]
load a      ;[a             ]
load b      ;[b     a       ]
load c      ;[c     b      a]
add         ;[c+b   a       ]
add         ;[c+b+a ]
store x     ;[c+b+a ]
```

The above code shows a fragment of a program assigning the sum of the three variables to x, written in a pseudo assembly language. We could have added the first two variables before loading the third one to get the same result. The advantage of operating with a data stack lies is that it removes the need to add extra bits to the instruction to specify which register is accessed. As a result, instructions can be packed in smaller words, the net result of which is a tighter fit in memory, and less time spent fetching the instructions from memory.

2.1.1.8 Issues

The definition of Parallel raises a number of further issues which must be considered :

"Components of a Program"

There is signifigant debate about what should be parallelised. This debate is intrinsically linked to the kinds of execution architectures that are available, and the structure of the problem itself (see Carriero and Gelernter, 1989). Various languages occupy the various niches in the landscape created by these variables. We will seek to examine this debate in further detail in the chapters on Architectures, Design, and Languages.

"Controlled, Cooperative Execution"

If two people are both seeking to pass through a doorway at one time, contention is bound to arise. In concurrent programming we find out ways to deal with this contention, and in distributed systems we can identify ways of implementing these control primitives without the need for global shared resources. However, parallelism introduces further issues in cooperation, such as routing and job allocation, which must be resolved.

"A number of processing elements"

Even if we can resolve the debate on what kind of processing abilities each processor should have, a debate will still rage over how many processors are needed, and how they are interconnected. Furthermore, a policy on and software provision for failure, reconfiguration, and recovery must be considered.

"Inter- connected to provide Communication"

The programmer must also begin to consider whether the interconnectivity will provide sufficient data rates to support a suitable increase in speed. Many parallel programs can actually work out to be ineffective, not because they do not work, but because they do not provide sufficient speed up.

2.1.1.9 Applications

Although many will be aware of the use of Cray supercomputers by varous weather centers, few are aware of other areas where parallel computation is applied. In fact, until the end of the 1970's there were relatively few applications which were using parallel architectures, mainly due to the cost of the machines that were available. During the 1980's, however, there was an explosion of interest and enthusiasm in parallel computing, and this has led to a widespread application of parallel computing. A few applications are highlighted below :

Scientific and Engineering Applications

Most applications of parallel processing in the Science and Engineering research communities have been focused upon numerical simulations where vast quantities of data must be processed in order to create or test a model. Example applications include:

- Global atmospheric circulation,
- Blood flow circulation in the heart,
- The Evolution of Galaxies,
- Simulations of artificial ecosystems,
- Airflow circulation over aircraft components,
- Atomic particle movement from collidatron experiments,
- Optimisation of mechanical components.
- Airflow circulation is a particularly important application. It is suggested that a large aircraft design company might perform up to five or six full body simulations per working day. It is important that the processing is timely in order to meet the work schedule, and therefore parallel processing is important.

Database Systems

Opportunties for speed-up through parallelising a Database Management System abound. However, the actual application of parallelism required depends very much on the application area that the DBMS is used within. For example, in the financial sector the DBMS would suffer a lot of short simple transactions, but with a high number of transactions per second, whilst in a CAD situation (eg. VLSI design) the transactions would be long and with low traffic rates. In a Text query system, the database would undergo few updates, but would be required to do complex pattern matching queries over a large number of entries. An example of a computer designed to be used to speed up database queries is the Teradata computer, which employs parallelism in processing complex queries.

AI Systems

Search is a vital component of many AI systems, and the search operations will be performed over large quantities of complex structured data using unstructured inputs. Applications of parallelis include :

- Search through the rules of a production system,
- Using fine-grain parallelism to search the semantic networks created by NETL,

- Implementation of Genetic Algorithms,
- Neural Network processors,
- Preprocessing inputs from complex enviroments, such as visual stimulii.

Image / Graphics Processing

The production of realistic moving images for television and the film industry is big business. Whilst much of the work can be done on high specification workstations, large computer animation input will often involve the application of parallel processing. Even at the cheap end of the image production spectrum, affordable systems for small production companies have been formed by connecting cheap PC technology using a small LAN to farm off processing work on each image to be produced.

2.1.1.10 Model Questions

1. What is meant by parallelism?
2. What are the various issues on the raised on parallelism?
3. How is parallelism applied ?

2.1.1.11 Summary

Although the enthusiasm of the 1980's has waned during the 1990's, the application of parallel processing to new problems continues to expand. With the continued introduction of multiple processor management capabilities into modern operating systems, and the spread of high speed networking, there would appear to be every reason to expect this trend to continue.

2.1.1.12 References

- Computers And Commonsense – Roger Hunt and John Shelly
- Operating systems Concepts – Silberschatz Galvin
- Operating Systems (3rd Edition) -- Harvey M. Deitel, et al

Lesson 2.1.2

INHERENT PARALLELISM IN PHYSICAL PHENOMENA AND ITS MODELS

Objective
2.1.2.1 Introduction
2.1.2.2 Defining Parallelism
2.1.2.3 Emerging Parallelism
2.1.2.4 Inherent Parallelism in Scientific and Engg. Applications
2.1.2.5 Inherent Parallelism in General Physical Models
Summary
Model Questions
References

Objective

To know the importance of parallelism in this modern computing world and inherent parallelism of different physical models.

2.1.2.1 Introduction

Very different computational requirements of user problems given rise to a broad spectrum of computers ranging from general purpose via specialized multipurpose to special architectures. For the choice of the appropriate computer system, the user has to take into account how the computational demands of the target application and involved economic considerations. After a short characterization of scientific and engineering problems, main hardware and software means for specialization art described. Because of the versatility of multiprocessors, such systems are especially well suited for adapting the architecture to the user problem. This issue is discussed for architectural concepts such as hierarchical bus systems and shared memory systems.

The "'T**" von Neumann computer is known as a universal computer. Since then, the computational performance has been increased by several orders of magnitude due to technological advance in hardware components, specialization and parallel processing.

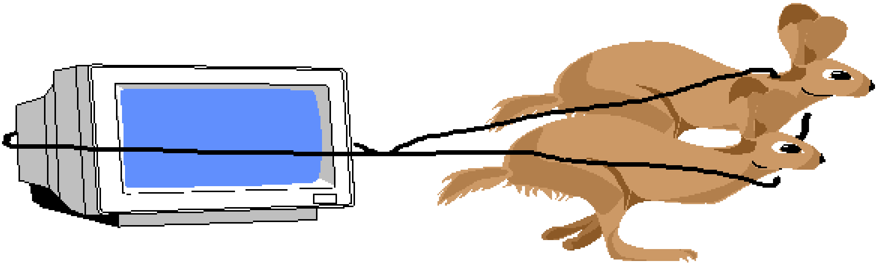
There has been an enormous gain of speed by very large scale integration technology, by fast special components and units by hierarchically organized memories and so on. Nevertheless, limits of technically possible performance of the uniprocessor are showing up. Users demand more and more computing power and storage capacity. On these grounds, the interest in parallel processing systems has strongly been growing within the last three or five years. With parallel systems also the structure of the interconnection system can be adapted to multipurpose or special purpose use. This possibility does not exist with uniprocessor systems.
Whether to use a general purpose or a special purpose machine is primarily an economic problem. A user needs a general purpose machine if it is aimed to a broad spectrum of application problems. On the contrary, be wants a highly efficient special

purpose machine if the application is well defined and featured by a small variation of its computational parameters. Anyway, the user will be interested in a system which provides for the needed computational power and storage capacity at a minimum price/performance ratio.

2.1.2.2 Defining Parallelism

Not all models are created equal. Even if two models are ``equivalent'', such that they may be used interchangeably to investigate a given system, the models may differ in many respects: (1) they may reflect different perspectives, (2) one may be more maintainable than another in a given setting, (3) one may facilitate analysis in a superior fashion, or (4) one may be more suitable for parallel execution than another. We refer to the fourth quality as the *inherent parallelism* of the model. According to the philosophy described below, inherent parallelism should be regarded as a function of the model representation, not the physical system. This important distinction is clarified in the subsequent discussion.

Instruction-Level
Parallelism



1

PDES (Partial Differential Equations) protocols, in their usage and evaluation, describe a notion of inherent parallelism. However, the concept is typically implicit, and vague in its characterization. Explicit formulations of inherent parallelism have appeared in as a basis for *critical path* analysis.

"Inherent parallelism refers to the intrinsic structure of a system which allows it to be split into a number of elements which can run independently of each other, without requiring information to be transferred between them."

2.1.2.3 Emerging parallelism

Parallel computational practices separate tasks, data, instructions and memory and distribute them -- in various different ways -- between separate but interconnected elements, which perform their operations simultaneously.

The idea is natural, obvious and immediate; which is to be expected if parallel computing is the inscription within the computational act of certain familiar (natural, obvious, immediate) forms of collective cognition. This was recognized by computer engineers some time ago. As B. Chandrasekaran observes in Natural and Social System Metaphors for Distributed Problem Solving: "It is clear that distribution of processing or computation is an intrinsic characteristic of most natural phenomena ... . Social organizations from honeybee colonies to a modern corporation, from bureaucracies to medical communities, from committees to representative democracies are living examples of distributed information processing embodying a variety of strategies of decomposition and coordination. Computation in biological brains, especially in their sensory processors such as vision systems, displays a high degree of distribution. There is substantial evidence that higher cortical functions are also computed (and controlled) in the brain in an essentially distributed mode ... ." .

Observe that, unlike many of its social and biological precursors, this inscription is fundamentally a question of software -- the computation is cognized, designed and specified in parallel terms -- regardless of how it is implemented. Of course, implementing it on distributed hardware is a natural choice, but not essential, and parallel proceedures can be made to run -- with varying degrees of difficulty and artificiality to be sure -- on serial machines.

The separate, distributed computing elements can vary greatly. They can be fully autonomous computers wired together in a network distibuted within a building, across a country or in different parts of the planet. They can be simple, stripped down computers hard wired to each other to form a supercomputer like the Connection Machine. They can be specialized computational engines, limited, at their most basic, to simple finite state mechanisms or automata governed entirely by local rules. Or they can be virtual versions of these, simulated engines within the memory of a single computer.

This last, usually called Cellular Automata (CA), has proved to be an extraordinarily fecund computational, explanatory and investigative technoscientific tool. The flocking behaviour of starlings we encountered at the beginning, as well as aspects of the behaviour of ants or bees in a colony, or cars in traffic patterns, are all examples of situations that can be illuminatingly modelled by a CA: thus, each starling is identified with an individual cell and the requirement to keep a fixed distance from its neighbors is its local rule; and likewise for the distributed behaviour of ants and bees. A quite different example of a CA comes from fluid dynamics: The Navier-Stokes equation in that subject, a major triumph of 19th century partial differential calculus, summarises the behaviour of an incompressible fluid. It turns out to be simulatable by a not very complicated CA which uses a hexagonal grid: each cell of which models a single drop of fluid subject to the flow in and out of it along the six directions governed by identical local rules. Parallel computational methods, which include all kinds of distributed and decentralized processes are increasing at almost an exponential rate in cognitive science, evolutionary theory, complexity studies and throughout technoscience from the level of abstract theorizing through heterogeneous modelling and simulation projects to base-level engineeering practices.

These include the generation of artificial life forms, including their habitats and ecosystems together with the simulation of evolutionary possibilities open to them; the invention and simulated creation of compounds, alloys and molecules with specified properties and functionality; pattern recognition and learning behaviour within expert systems via simulation techniques using connectionist and neural networks; and genetic algorithms that evolve, refining their ability to solve a problem through the feedback of the results of repeated trials.

2.1.2.4 Inherent parallelism in Scientific and Engineering Applications

Multiple processing systems have additional overhead compared to uniprocessor systems. This is due to unbalanced load in the nodes of a multiple processing system and due to interprocessor communication.
Despite these drawbacks there are two reasons to build multiple processing systems:

To offer higher performance for computing large user problems than can be obtained with uniprocessors at comparable price/performance ratio, and to realize a fault-tolerant computing system, for instance in case of safety requirements. Obviously large multiple processing systems need some fault recognition and recovery mechanisms.

We only refer to the high performance issue. User problems with high computational and memory capacity requirements mostly belong to scientific or engineering application fields. Principal problem classes are:

(1)Numerical grid problems:
Approximate solution of sets of partial differential equations (PDEs) by discretization of space aod time. This problem class plays an essential role in scientific supercomputing as it comprises many application fields in science and engineering (e.g. numerical simulation of physical phenomena in material science, aerodynamics etc).

(2)Panicle and field models:
Such models are used for a broad spectrum of applications in plasma physics, solid state physics, aerodynamics, chemistry and others. For the mathematical description Molecular Dynamics and Monte-Carlo methods are applied

(3)Simulation of very large scale integrated (VLSI) electronic circuits (design and verification): According to the simulation level to be considered different methods and mathematical procedures are applied:

System level: Functional specification.
Logic and switch level: Logic operations.
Electric circuit and timing level: Sets of coupled, nonlinear ordinary differential equations, Fourier transform and others.
Device and process level: Modeling of physical phenomena in electronic devices and technological processes, e.g. mathematical description by sets of partial differential equations (PDEs).

(4) Ab initio quantum mechanical calculations of chemical systems for the theoretical prediction of the properties of new materials or for the interpretation of new physical phenomena (e.g. high temperature superconduction). Such problems comprise the calculation of 1- and 2-electron-integrals, operations of large matrices and the determination of eigen values.

These four problem dasses have in common a strong demand for East numerical operation unique and large memory capacity. The degree of inherent parallelism generally is high. In the case of VLSI simulation, this is only true for device and process simulation where highly parallel algorithms can be used. A quite different situation is found in the following problem class:

(5) Artificial intelligence and automation. This problem class comprises application areas as for instance image processing, pattern recognition, computer vision, speech understanding, intelligent robotics, expert systems and others. Depending on the particular problem to be solved, there are requirements for massively parallel operations (e.g. image preprocessing) up to complex computations with low inherent parallelism (e.g. knowledge engineering, decision making).

This coarse classification of user problems in science and engineering shows the heterogeneity of computational structures involved. Moreover, given one type of a problem, there can be a broad variation of computational parameters even between similar problems. As an example, the number of floating point operations to be executed per data access can vary by orders of magnitude. For performance optimization, the designer has to balance processor performance and data supply rate (e.g. memory accesses per second). Maximum performance can be achieved for only one case, that is, for one set of parameter values.

Due to the heterogeneity and broad variation of the computational structure of a great deal of user problems each processor node must work under its own control unit in order to allow for independent processing of subtasks and flexible interprocessor communication.

Appropriate parallel computers belong to the category of multiple-instnictioa-multiple-data (MLMD) systems. Such systems are realized as multiprocessor or multicomputer machines based on various architectural concepts. Such systems can be used with great versatility as the nodes can execute the same or different instructions and operate on the same or different data.

2.1.2.5 Inherent parallelism in general physical models

Parallel/serial as universal duo

To respond, and to show what might be at stake in such a question, I want to back off from computing as the prime site of instantiation of serial/parallel, and move to other arenas. I want to give various examples of the duo in operation. What will emerge from these is how the two poles operate together, impinging on each other as a combinatorial tool everywhere from our pre-mammalian origins to presentday culture. Thus, not only is the serial/parallel opposition a widespread organizing and creative principle across various humanistic, artistic, mathematical, technoscientific, linguistic, and

epistemological practices, but it is also to be found within biological systems as a hard wired functioning binary.

Mathematics: an entire subject organized around and predicated on the serial/parallel opposition. As Tobias Dantzig, Number: The Language of Science, in his discussion of the two conceptual moves needed to handle whole numbers observes: "Correspondence and succession, the two principles which permeate all mathematics -- nay, all realms of exact thought -- are woven into the fabric of our number system.". The first refers to the one-to-one correspondence whereby the elements of one collection are matched or tallied with those of another; the second refers to the process of ordering the elements into a sequence as part of counting them. Thus correspondence allows one to judge which of two collections has more elements in the absence of any need (or ability) to count them; succession determines how many elements are in each collection.

Thus number is a serial/parallel construction. But, as Dantzig declares, the opposition is implicated throughout mathematics and beyond. Certainly, serial (succession) as against parallel (correspondence), in the form of dependence of one thing on a given other versus independence of two co-occurrently given things, is fundamental to the construction of all post-Renaissance mathematics. As such it is, for example, the founding abstraction of co ordinate geometry, as well as that of an algebraic variable and the notion of a function; it institutes the separation of independent and dependent events and hence founds the idea of a random variable in the theory of probability. More primitively, as indicated the parallel is the all-at-once magnitude of cardinal numbers, their determination as unordered collections or combinations against the sequential, counted-into-being ranking of ordinals or permutations.

On the relation between ordinal and cardinal, there is the anecdote of the clocks: A man heard the clock strike two times one day, just as he was falling asleep, and he counted like this: "One, one." Then, when he realized how ridiculous that was, he said, "The clock has gone crazy: it struck one o'olock twice!" Quoted in George Ifrah From One to Zero Or again: there is the difference, crucial in the theory of sets, between the ordered pair (a,b) and the unordered pair {a,b} of two objects, and the propriety (discussed by mathematical logicians) of Kuratowski's formal definition of the former in terms of the latter.

Not only do "the two principles permeate ... all exact thought", and prove to be "woven into the fabric of our number system", but they also -- well outside the field of mathematics or of so-called exact thought -- form a ubiquitous and formidable constitutive principle. Put differently, the interplay of parallel and serial principles in the manufacture and replication of concepts gives rise to an enormous idea machine, a combinatorial tool or technology that permits the signifying, patterning, imagining -- constructing/discovering -- of an unsurveyable plenitude of 'objects'. Objects whose viability and creative potential lie precisely in the way they neutralize the very difference between serial and parallel that allowed them to be brought into existence.

By way of elaboration consider three examples: the code of Western classical music, the language of traditional arithmetic, and the code of mathematical theory of infinite sets. In each case the 'objects' making up the code -- musical compositions, integers, infinite

numbers -- arise from an initial formal constraint. They are manufactured via a principle of equality or interchangeability that operates as a built-in insistence that -- despite the evident opposition between them upon which music, arithmetic, set theory are founded -- any parallel object be equivalent to a serial one and vice versa.

In classical music, with its enormously rich, intensely specialized mass of composition based on key harmonies, this folding of serial and parallel into each other is correspondingly complex and detailed. At bottom, however, it amounts to a vast algebra of forms: compositions which arise from the different ways musicians have formulated of re-writing and arranging sequential progressions into simultaneous chords and spilling harmonies over time to be the successive notes of arpeggios and the like.

In traditional arithmetic the principle of ordinal/cardinal interchangeability is so ingrained, and the proliferation of objects so effortless, that it's difficult to detach the principle of parallel-serial interchangeability from the familiar idea of 'whole number'. Thus, not only is it too obvious to even remark that an ordinal is necessarily a cardinal, but the reverse is unasked: why can every collection, however named or described or defined -- and independently of any method of achieving such a thing -- be 'counted' into a sequence? What hidden necessity guarantees the possibility - the eventuality -- of totally ordering anything nameable?

In the theory of infinite sets ordinals are defined to be sets and so are automatically possess a cardinal magnitude, whilst the reverse is precisely the content of the notorious axiom of choice, the axiomatic principle needed to guarantee that all sets can be well-ordered. No exaggeration to point out the possibility of this cardinal/ordinal interchange as the constitutive armature of Cantor's infinite arithmetic: certainly without it the entire theory of sets as developed during the twentieth century would be impossible.

One can press harder on the phenomenon of serial/parallel interchange, and identify what might be called an *horizon* effect: in each case the technology of production, the means of creating the plenitude of objects, is subject to an insurmountable limit, an unanswered or even unanswerable question, whereby an horizon of the machine is revealed; and with this emerges the impossibility of running the machine *from within*, as before without reference to the presence of its external features. For Western classical music composition the system of vertical-horizontal equivalences collapsed early in the 20th century, when the key-based harmonies which controlled the chord/arpeggio trade-off were repudiated by a movement appropriately calling itself *serialist.* For set theory the horizon of the machine was revealed through the proof in 1963 of the independence of the continuum hypothesis, which left unsolvable and essentially unresolvable the question of the magnitude of the continuuum (as well as the independence of the axiom of choice that allowed the question of this magnitude to be posed). For the classical integers and their arithmetic the horizon -- less obvious, more contentious and needing considerable groundwork to reveal it -- arises from the challenge to the orthodox account of infinity, and the subsequent emergence elaborated in Ad Infinitum ... the Ghost in Turing's Machine of non-infinitistic, in fact non-Euclidean arithmetic.

In natural language, the opposition of serial/parallel is basic; an intrinsic and constitutive binary. It appears, as a very general linguistic distinction according to the Glossary of Semiotics, as the opposition of syntagmatic ("relationships ... of linear, temporal sequence") and associative or paradigmatic ("relationships [which] do not as such occur in time [but] make up an array of possibilities"). Or again, according to Roman Jakobson, Fundamentals of Language, as a completely abstract and general feature opeative at all levels of speech: "The concurrence of simultaneous entities and the concatenation of successive entities are the two ways speakers combine linguitic constituents.". Jakobson goes on to observe that "The fundamental role which these two operations play in language was clearly recognized by Ferdinand de Saussure. Yet of the two varieties of combination -- concurrence and concatenation -- it was only the latter, the temporal sequence, which was recognized by the Geneva linguist.", a fact which, according to Jakobson, stems from Saussure's immersion in the traditional belief "qui exclut la possibilité de prononcer deux elements a la fois". Evidently, the serial/parallel duo functions at all levels of speech: phoneme as simultaneous bundle of distinctive features, syllable as succession of phonemes, the inherent parallelism of intonation/gesture, the combined linearity and simultaneous unity of utterances, and so on.

A final example: twentieth century physics. There is the well known parallelist phenomenon of *superposition* in the standard (Copenhagen) interpretation of quantum physics, where all the mutually contradictory states of a quantum system, ghost tendencies that Heisenberg called *potentia*, are taken to be simultaneously present but unrealized. This is opposed to an actual or 'real' state of the system resulting from a measurement (the so-called collapse of the wave function), where such actualities are understood as occurring one after the other. What can legitimately collapse the parallel into a serial, what in other words constitutes a measurement is a major mystery -- the so-called 'measurement problem' -- for such a view. Interestingly, the main rival theoretical model of quantum events, the many-worlds interpretation, eschews superposed parallel tendencies and so eliminates the measurement problem. By positing one totally determined, unghostly state at a time in each of a multitude of 'simultaneously occurring' worlds, it replaces parallel unreal occurrences in one world with real occurrences in parallel unreal worlds.

These examples, from brain morphology, Western classical music, mathematics, spoken human language, and quantum physics, and the instance of computing we started from, demonstrate the importance of the parallel/serial duo, as a creative and organizing principle across many terrains. Further, as is evident from the depth at which it operates and the dynamics it gives rise to, the duo acts in some sense like a zero sum game. By which I mean not that there is a hidden equilibrating force or larger matrix of control ensuring their balance, but that by virtue of their mutual enfolding within specific cultural practices changes -- of status, scope, attributed importance, aesthetic worth, semiotic transparence -- to one pole are accompanied by changes in the other. It follows from this that the shift to parallelism being charted here will be associated, as we shall see, with a countervailing, newly emergent form of serialism.
The claim we need to flesh out comes, then, to this: two co-occurrent, synergistic transformations -- the ongoing move to parallel and distibuted computing and the explosive growth in visualization -- are reconfiguring contemporary technoscience. The

effects of this emergent parallelism as it circulates through cultural space are being felt at every level from how we read, write, and see to the ways we understand ourselves as 'selves'. And that, in the process of its unfolding this parallelism is giving rise to a hitherto unavailable, and yet to be adequately identified, serialism.

Paraselves and multi-IDs

Am I beside myself or are their two or more of me/us? Can I, can 'one'(but one can't say 'one'), have more than one identity?

"Now we are one, or two, or three" A recent headline from the New Scientist acknowledges our widespread pluralization, and the multi-disciplinary, multi-cultural, multi-tasking, multi-plex environment we inhabit. It could easily be the title of a piece on cognitive science's idea that the mind is not and never was a single agent, but an assemblage of different and competing agents; or a report on neuroscience's understanding of the mind/brain as a many-sided modular organ whose morphology indicates two or three or more independent functioning units; or a human interest piece about the recent increase in the number of people doing two things at once like using mobile phones and wrecking their cars.

In fact, it's about Multiple Personality or Dissociative Identity Disorder. MPD/DID is a hot topic with many books, hundreds of articles, debates, etc., made ultra hot by the (widely accepted) view of it as a disorder of childhood abuse, which automatically links it to FAQs from schizophrenia to alt.abuse.transcendence. I'll mention two books.

First Person Plural by Stephen Braude, acquiesces in this origin of Multiple Personality in abuse and understands the phenomenon as the disruption of a natural unpathological, unitary self. The vocabulary of pathology and the need to justify the naturality of a single personality leads him to theorize the *necessary* existence "of an underlying synthesizing subject". Braude is an analytic philosopher with a conventional -- that is to say in the present context enlightenment -- epistemological agenda, so no surprise to his polemic against the possibility that being plural or multi is anything other than deviant departure from a prior (Kantian) subject which is the condition for the possibility of any (rational) thought.

The abuse etiology is directly challenged, however, by Ian Hacking Re-writing the Soul which situates the multiple personality effect within a history of memory, locating it in the "conceptual space for the idea of multiplicity" constructed by French medicine in the 19th century, used by patients to describe their symptoms and then looped back through the doctor/patient circuit into confirmatory evidence of a disease. Though sharply argued and historically focussed, Hacking's analysis elides the issue of contemporaneity: even if, as he maintains, sufferers from the syndrome are creating and fitting their symptoms to pre-given diagnosis, we're still left with the question: why *these* symptoms right *now*? Why the irruption of this kind of multiplicity within presentday culture?

One body with many (up to 96 so far) identities or "alters" differently related: most claiming to be solo, but some aware of their co-inhabitants, some fully worked out personae, but most persona-fragments and generic functions, such as The Angry One,

the Innocent Child, etc. Maybe schizophrenia opens here into a generalized obverse: instead of the original unity -- we were all one once -- become split and fragmented, we have an originary collectivity mainfesting as a barely -- and not necessarily -- unitizable ensemble: Stevenson's multifarious, incongruous denizens.

If, as Louis Sass Modernism and Madness has it, schizophrenics were the sensitives, the "town criers of modern consciousness ... existing not just as a product *of* but also a reaction *against* the prevailing social order", too easily able to internalize their rent and disordered times, then perhaps the presentday multiples are their successors: emblems of the multiplex instabilities of 21st century psychic reality whose ur-myth is nearer to Osiris than Oedipus. This is not to deny that multiples aren't strange, aberrant, frequently traumatized and needing help, but rather to suggest -- leaving aside their disputed etiology -- that their aberrance might serve, at least for now, as "the best paradigm we have for postmodern consciousness" and, beyond that, might presage an inescapable aspect of future normalcy.

It is worth observing here that at least one account of multiple personality, a neurological-based picture that evidences clinical and hypnosis findings, sees it as part of an underlying multiplicity of brain function. Thus, for Oakley and Eames, in The Plurality of Consciousness, the syndrome is a divergence form a normal and ongoing mental parallelism: "It seems likely that the multiple parallel streams of conscious activity [i.e. activity we are aware of], which are implied by the multiple personality data, are no different from those which are present in normal individuals ... . The difference in the multiple personality case is that these processes can be attached to different self represntations, and so when re-reprented are revealed as the thoughts of different individuals. When only one self representation is available to self-awareness all conscious processes, covert of othewise, is attibuted to a single 'me'." On this account, then, multiples whilst undeniably aberrant, are closer to all of us than we have ever, from the perspective of a natural, unfractured singularity, imagined.

If multiples deny the indivisible subject and the equation of one self with one body from within, then MUDs or Multiple User Domains effect the inequality externally. Though both uncouple the self/body unit (powerfully contested site of contemporary reality that, for example, Allucquere Stone focuses on in The War of Technology and Desire), the virtual (or cyber or net or web) communities that emerge from MUDs have no history of deviancy attached to them, and as a result pose a more complex and less easily dismissed effect of pluralization than those with multiple personae.

Wherever a collective presence is constructed, from primitive bulletin boards and conference calls to sophisticated chat rooms on the net, and so on, virtual presences arise from a separation between the physical substrate and the persona: the body parked at the terminal or jacked into a VR rig and the self, ranging solo around simland, or engaged in any manner of intimacies anywhere on the net with sundry other disembodied, masked and anonymized virtual presences. In these contexts, the Rastafarian usage "I-and-I" for "we" takes on a special and useful ambiguity, since the first person 'I' is neither plural nor singular but an archaic misnomer for an emergent I/me/us construct.

Thus, as parallel computation writes collective cognition into a thinking machine understand for millenia as an individual, so multiples and verts do the same for the consciousness machine and its software we call the psyche: they effect a corresponding inscription -- what might be called a phenomenological collectivization -- at the level of 'individual' perception and experience. In this, they realize in well-defined, repeatable socio-technologic form, a pluralized I/me.

But of course, there is no separation here between interior and exterior: the experiential and the collective fold into each other. All thought, even the most private, individual and enclosed is social. Being socially present, mobilized and used is co-creative with the psyche -- a phenomenon that seems difficult to theorize in any general way outside a techno-ecology of the mind/brain. It is in this sense that one should interpret Merlin Donald's contention that the key principle of the biological and social evolution of individual cognition is the symbiosis of cognitive collectivities and external memory systems, a linkage that allows new cultural formations and technologies to reconfigure the thought diagrams inside (as we still say) our heads.

Summary

Very different computational requirements of user problems given rise to a broad spectrum of computers ranging from general purpose via specialized multipurpose to special architectures. For the choice of the appropriate computer system, the user has to take into account how the computational demands of the target application and involved economic considerations. After a short characterization of scientific and engineering problems, main hardware and software means for specialization art described. Because of the versatility of multiprocessors, such systems are especially well suited for adapting the architecture to the user problem. This issue is discussed for architectural concepts such as hierarchical bus systems and shared memory systems.

Model Questions:

1. What is inherent parallelism and write a brief note on its applications in scientific and engg. Applications?
2. What is parallelism and its importance in modern computing?

References:

1. www.thesimguy.com/ernie/papers/unref/dissert/node10.html
2. www.wideopenwest.com/~brian_rotman/parallel.html
3. Exploiting Inherent Parallelism In Non-Linear Finite Element Analysis,M. W. S. **Jaques, C. T. F. Ross** And **P. Strickland** Computers & Swucrures Vol. 58, No. 4, Pp. 801-807, 1996

AUTHOR:

B.M.REDDY **M.Tech. (HBTI, Kanpur)**

Lecturer, Centre for Biotechnology

Acharya Nagarjuna University.

Lesson 2.1.3

INHERENT PARALLELISM IN BIOLOGICAL PHENOMENA AND ITS MODELS

Objective

2.1.3.1 Introduction
2.1.3.2 Parallel computing in biological processes
2.1.3.3 Intelligent Computing Using Graphics Processors (GPUs)
2.1.3.4 Simulation Modeling Methodolgy
Summary
Model Questions
References

2.1.3.1 Introduction

Over the past decade whole genome sequencing has evolutionised the biologloical sciences. A new era of data rich science in biology has arisen based on the whole genome projects and the wealth of post-genomic studies that they facilitate, aimed at understanding the many complex cellular processes that occur in living organisms. Hence there is now an unprecedented amount of data available about biological systems. These new data resources have enabled the systems wide study of biological systems, referred to as Systems Biology, in which the goal is to understand how the individual biological parts interact to yield the behaviour of the whole system. One important approach in Systems Biology is the modeling and simulation of a biological net- works to help understand and predict the behaviour of these complex systems. The simulation of biological networks are carried out with either deterministic simulators, stochastic simulators, or hybrid simulators, each of which have their own advantages and disadvantages. Stochastic simulations have been shown to capture the fine grain behaviour and randomness of outcome of biological networks not captured by deterministic techniques and as such are becoming an increasingly important technique. However, current efforts in the stochastic simulation of biological networks are hampered by two main problems: (i) First, there is a lack of quantitative data on molecular concentrations and kinetic parameters that are essential to the successful simulation of biological networks. (ii) Secondly there is a large computational cost to stochastic simulation of these networks. A recent review suggested that an average personal computer would take a whole day to simulate 100 minutes of a 100 reaction system. This problem is exacerbated since multiple repetitions of simulations are often required.

2.1.3.2 Parallel computing in biological processes

Parallel computational methods, which include all kinds of distributed and decentralized processes are increasing at almost an exponential rate in cognitive science, evolutionary theory, complexity studies and throughout technoscience from the level of abstract theorizing through heterogeneous modelling and simulation projects to base-level engineeering practices.

On the understanding that parallel computing inscribes distributed biosocial and biological phenomena, in particular collective cognition, one would predict this explosion of use to be open ended: what is created is work, designs, proceedures and routines not previously doable or often even thinkable from the perspective of individualized cognition. On the other hand, it follows that the effects, consequences and cultural disruptions inherent in parallel thinking are not easily predictable, since collective cognition is heterogeneous, unschematized, and emergently different from individual thought in ways that, as we saw earlier from the critique of cognitivism, have scarcely begun to be articulated.

Inherent Parallelism

The power of the DNA computing model lies in the fact that very large numbers of oligos can anneal to form a vast number of combinations in a short period of time and in a small volume of solution. As an example, let's consider the directed hamiltonian cycle problem on a graph with n vertices. Given the starting vertex v_s, there are a possible $(n-1)!$ permutations of the vertices between beginning and ending at v_s. To explore each permutation, a traditional computer must perform $O(n!)$ operations to explore all possible cycles. However, the DNA computing model only requires the representative oligos. Once placed in solution, those oligos will anneal in parallel, providing all possible paths in the graph at roughly the same time. That is equivalent to $O(1)$ operations, or constant time. In addition, no more space than what was originally provided is needed to contain the constructed paths

2.1.3.3 Intelligent Computing Using Graphics Processors (GPUs)

Background & Description of Technology:

High-performance 3D graphics systems have become as ubiquitous as floating-point hardware. They are now a part of almost every personal computer or game console and some handhelds. In fact, the two major computational components of a computer system are its main processor (CPU) and its graphics processor, also known as the GPU. While the CPUs are used for general purpose computation, the GPUs have been primarily designed for drawing and filling primitives, geometric transformations and texturing. The main application of GPUs has been fast rendering of lighted, smooth shaded, depth buffered, texture mapped, anti-aliased triangles for visual simulation, virtual reality, and computer

gaming. Some of the recent GPUs also include advanced features like multi-texturing, pixel textures, programmable shading and programmable vertex engines, and support for floating-point fragment pipelines and frame buffers. As graphics hardware becomes more programmable, the barrier between the CPU and the GPU is being redefined. The GPU can also be regarded as an efficient processor of images or a useful co-processor for many diverse applications.

Following are the applications of GPU:

Modeling of Simulated Forces: Complex reasoning in real-world environments requires the ability to simulate, reason, and plan for semi-autonomous or autonomous agents used in various simulators for training, prototyping, mission planning purposes. Critical issues that require attention are route planning for large number of agents, robot controllers, complex reasoning, navigation and guidance of autonomous agents, computational support for behavior modeling, and interactive visualization of battle space on programmable graphics processors.

Physically-based Simulation: Several aspects of synthetic environments require fundamental advancements in physically-based modeling and simulation techniques for modeling complex physical and biological systems, as well as natural phenomena (e.g. freezing rain, sand storms, etc). Collision detection and response are essential in modeling complex interaction among multiple simulated agents and entities. For example, in Computer-generated Force computations, physically-based simulations take up a majority of computational cycle. Other geometric problems essential to physical simulation include distance computation, visibility, etc. Performing physical simulations on the GPUs can potentially enable us to incorporate physics-based modeling techniques into many interactive applications. Moreover, they can run on mobile platforms or embedded systems.

Network and Communications: Communication requirements of the simulated environment are distinct from those based on conventional interactions. One of the key issues is the “line-of-sight” computation for large number of simulated agents, especially under critical conditions such as varying weather conditions and non-optimal visibility conditions often encountered in the battlefield.

Database and Data Mining: The enablement of fast database operations allows for the rapid query and data search on any application systems. Research advances in this area can exploit the inherent parallelism within programmable graphics processors. It can considerably speedup the performance of database queries on complex databases.

Compiler Support and Software Environments: In addition to examining the possibility of performing various computations mentioned above, other fundamental issues include programmability, language and compiler support, and development of flexible software environment for GPUs.

Development status/technical maturity/limitations: Although over the last year and two, several new algorithms and applications that exploit the inherent parallelism and vector processing capabilities of graphics processors (GPUs) have been proposed, the basic technology for computing on GPUs still is not fully developed. The Technology Readiness Level is Level 4. It needs further research and development to live up to their full potential.

Forecast of Capability in the next 3-5 years: The growth rate of GPUs for the next 3-5 years is shown in the graph. We have also compared its growth rate with Moore's Law. While the CPUs are expected to grow at the rate of 30-50 times, the GPUs peak performance might grow by almost 2-3 orders of magnitude during the same period. They could have the capability to perform tera-flop operations per second by 2007 and this makes them an attractive candidate for simulation and Computer-generated force computations.

2.1.3.4 Simulation Modeling Methodolgy

Investigation in discrete event simulation modeling methodology has persisted for over thirty years. Fundamental is the recognition that the overriding objectives for simulation must involve decision support. Rapidly advancing technology is today exerting major influences on the course of simulation in many areas, e.g. distributed interactive simulation and parallel discrete event simulation, and evidence suggests that the role of decision support is being subjugated to accommodate new technologies and system-level constraints. Two questions are addressed by this research: (1) can the existing theories of modeling methodology contribute to these *new* types of simulation, and (2) how, if at all, should directions of modeling methodological research be redefined to support the needs of advancing technology.

Requirements for a *next-generation modeling framework* (NGMF) are proposed, and a *model development abstraction* is defined to support the framework. The abstraction identifies three levels of model representation: (1) modeler-generated specifications, (2) transformed specifications, and (3) implementations. This hierarchy may be envisaged as consisting of either a set of narrow-spectrum languages, or a single wide-spectrum language. Existing formal approaches to discrete event simulation modeling are surveyed and evaluated with respect to the NGMF requirements. All are found deficient in one or more areas. The Conical Methodology (CM), in conjunction with the Condition Specification (CS), is identified as a possible NGMF candidate. Initial assessment of the CS relative to the model development abstraction indicates that the CS is most suited for the middle level of the hierarchy of representations - specifically functioning as a form for analysis.

The CS is extended to provide wide-spectrum support throughout the entire hierarchy via revisions of its supportive facilities for both model representation and model execution. Evaluation of the pertinent model representation concepts is accomplished through a complete development of four models. The collection of primitives for the CS is extended to support CM facilities for set definition. A higher-level form for the report specification is defined, and the concept of an *augmented* specification is outlined whereby the object

specification and transition specification may be automatically transformed to include the objects, attributes and actions necessary to provide statistics gathering. An experiment specification is also proposed to capture details, e.g. the condition for the start of steady state, necessary to produce an experimental model.

In order to provide support for model implementation, the semantic rules for the CS are refined. Based on a model of computation provided by the action cluster incidence graph (ACIG), an implementation structure referred to as a direct execution of action clusters (DEAC) simulation is defined. A DEAC simulation is simply an execution of an augmented CS transition specification. Two algorithms for DEAC simulations are presented.

Support for parallelizing model execution is also investigated. Parallel discrete event simulation (PDES) is presented as a case study. PDES research is evaluated from the modeling methodological perspective espoused by this effort, and differences are noted in two areas: (1) the enunciation of the relationship between simulation and decision support, and the guidance provided by the life cycle in this context, and (2) the focus of the development effort. Recommendations are made for PDES research to be reconciled with the ``mainstream'' of DES.

The capability of incorporating parallel execution within the CM/CS approach is investigated. A new characterization of inherent parallelism is given, based on the time and state relationships identified in prior research. Two types of inherent parallelism are described: (1) inherent event parallelism, which relates to the independence of attribute value changes that occur during a given instant, and (2) inherent activity parallelism, which relates to the independence of attribute value changes that occur over all instants of a given model execution. An analogy between an ACIG and a Petri net is described, and a synchronous model of parallel execution is developed based on this analogy. Revised definitions for the concepts time ambiguity and state ambiguity in a CS are developed, and a necessary condition for state ambiguity is formulated. A critical path algorithm for parallel direct execution of action clusters (PDEAC) simulations is constructed. The algorithm is an augmentation of the standard DEAC algorithm and computes the synchronous critical path for a given model representation. Finally, a PDEAC algorithm is described.

In terms of human semiosis, episodes and procedures corespond to the opposition between pictures and words, between the parallel co-occurrence of the information in a scene and the sequential delivery of speech. The two forms, found in birds as well as all mammals, employ entirely different neural mechanisms, are morphologically distinct and functionally incompatible: "Whereas procedural memories generalize across situations and events, episodic memory stores specific details of situations and life events" . Of course, identifying the opposition here in no way claims for it a total coverage of the field of memory, and indeed, with the advent of language a third, conceptual form of memory emerged. But whilst this adjoined, re-organized and in much of culture dominates the more primitive substrate it found, it in no sense obliterated the episodic/procedural couple.

This last, usually called Cellular Automata (CA), has proved to be an extraordinarily fecund computational, explanatory and investigative techno scientific tool. The flocking behaviour of starlings we encountered at the beginning, as well as aspects of the behaviour of ants or bees in a colony, or cars in traffic patterns, are all examples of situations that can be illuminatingly modeled by a CA: thus, each starling is identified with an individual cell and the requirement to keep a fixed distance from its neighbors is its local rule; and likewise for the distributed behaviour of ants and bees. A quite different example of a CA comes from fluid dynamics: The Navier-Stokes equation in that subject, a major triumph of 19th century partial differential calculus, summarises the behaviour of an incompressible fluid. It turns out to be simulatable by a not very complicated CA which uses a hexagonal grid: each cell of which models a single drop of fluid subject to the flow in and out of it along the six directions governed by identical local rules.

Summary

No question any more that an event -- global, all penetrative, encompassing, inescapable -- is arriving and being bidden by us to happen. Within this event we are going parallel and becoming plural in ways and for the reasons . . One important approach in Systems Biology is the modeling and simulation of a biological net- works to help understand and predict the behaviour of these complex systems. The simulation of biological networks are carried out with either deterministic simulators, stochastic simulators, or hybrid simulators, each of which have their own advantages and disadvantages.

Model Questions:

1. What is inherent parallelism and write a brief note on its applications in scientific and engg. Applications?
2. What is parallelism and its importance in modern computing?

References:

1. www.thesimguy.com/ernie/papers/unref/dissert/node10.html
2. www.wideopenwest.com/~brian_rotman/parallel.html
3. Exploiting Inherent Parallelism In Non-Linear Finite Element Analysis,**M. W. S. Jaques, C. T. F. Ross** And **P. Strickland** Computers & Swucrures Vol. 58, No. 4, Pp. 801-807, 1996

AUTHOR:

B.M.REDDY **M.Tech. (HBTI, Kanpur)**

Lecturer, Centre for Biotechnology

Acharya Nagarjuna University.

Lesson 2.1.4

PARALLEL VERSUS SEQUENTIAL COMPUTING

Objective
2.1.4.1 Introduction
2.1.4.2 Eras of computing
2.1.4.3 What is Parallelism?
2.1.4.4 Parallel Computing
2.1.4.5 Parallelism in real life
2.1.4.6 Inter-thread communication
2.1.4.7 Sequential Computing
2.1.4.8 Parallel vs sequential solutions
Summary
Model Questions
References

Objective

This lesson explains the eras of computing and differentiates the parallel and sequential computing.

2.1.4.1 Introduction

It is now clear that silicon based processor chips are reaching their physical limits in processing speed, as they are constrained by the speed of electricity, light and certain thermodynamic laws. A viable solution to overcome this limitation is to connect multiple processors working in coordination with each other to solve grand challenge problems. Hence, high performance computing requires the use of Massively Parallel Processing (MPP) systems containing thousands of powerful CPUs. A dominant representative computing system (hardware) built using MPP approach is C-DAC's PARAM supercomputer.

2.1.4.2 Eras of computing

The most prominent two eras of computing are : sequential and parallel era. In the past decade, parallel machines have become significant competitors to vector machines in the quest for high performance computing. A century wide view of development of computing eras is shown in fig. The computer era starts with a development in hardware architectures, followed by system software, application, and reachingits saturation point with its growth in problem solving environments.

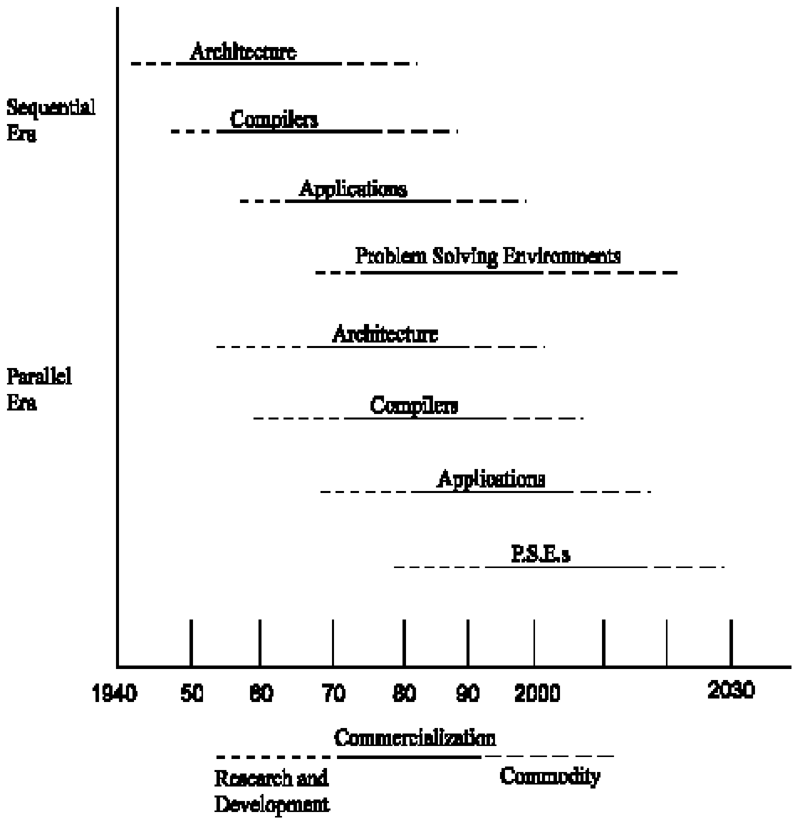


Figure 1.1: Two Eras of Computing

2.1.4.3 What is Parallelism?

A strategy for performing large, complex tasks faster.

A large task can either be performed serially, one step following another, or can be decomposed into smaller tasks to be performed simultaneously, i.e., in parallel.

Parallelism is done by:

- Breaking up the task into smaller tasks
- Assigning the smaller tasks to multiple workers to work on simultaneously
- Coordinating the workers

Parallel problem solving is common. Examples: building construction; operating a large organization; automobile manufacturing plant

Parallel computing is the simultaneous execution of the *same task* (split up and specially adapted) on multiple processors in order to obtain faster results.The term parallel processor is sometimes used for a computer with more than one processor, available for parallel processing. Systems with thousands of such processors are known as massively parallel.

There are many different kinds of parallel computers (or "parallel processors"). They are distinguished by the kind of interconnection between processors (known as "processing elements" or PEs) and between processors and memories. Flynn's taxonomy also classifies parallel (and serial) computers according to whether all processors execute the same instructions at the same time (**single instruction/multiple data** -- SIMD) or each processor executes different instructions (**multiple instruction/multiple data** -- MIMD). Parallel processor machines are also divided into symmetric and asymmetric multiprocessors, depending on whether all the processors are capable of running all the operating system code and, say, accessing I/O devices or if some processors are more or less privileged.

Performance vs. cost

While a system of *n* parallel processors is less efficient than one *n*-times-faster processor, the parallel system is often cheaper to build. For tasks which require very large amounts of computation, have time constraints on completion and *especially* for those which can be divided into *n* execution threads, parallel computation is an excellent solution. In fact, in recent years, most high performance computing systems, also known as supercomputers, have a parallel architecture

Algorithms

It should not be imagined that successful parallel computing is a matter of obtaining the required hardware and connecting it suitably. The difficulty of cooperative problem solving is aptly demonstrated by the following dubious reasoning:

> If it takes one man one minute to dig a post-hole then sixty men can dig it in one second.

In practice, linear speedup (i.e., speedup proportional to the number of processors) is very difficult to achieve. This is because many algorithms are essentially sequential in nature.

Up to a certain point, certain workloads can benefit from pipeline parallelism when extra processors are added. This uses a factory assembly line approach to divide the work. If the work can be divided into *n* stages where a discrete deliverable is passed from stage to stage, then up to *n* processors can be used. However, the slowest stage will hold up the other stages so it is rare to be able to fully use *n* processors.

Most algorithms must be redesigned in order to make effective use of parallel hardware. Programs which work correctly in a single CPU system may not do so in a parallel environment. This is because multiple copies of the same program may interfere with each other, for instance by accessing the same memory location at the same time. Therefore, careful programming is required in a parallel system.

Superlinear speedup - the effect of a N processor machine completing a task more than N times faster than a machine with a single processor similar to that in the multiprocessor has at times been a controversial issue (and lead to much benchmarking) but can be brought about by such effects as the multiprocessor machine having not just N times the processing power but also N times cache and memory thus flattening the cache-memory-disk hierarchy, more efficient use of memory by the individual processors due to partitioning of the problem and a number of other effects. Similar boosted efficiency claims are sometimes aired for the use of a cluster of

cheap computers as a replacement of a large multiprocessor, but again the actual results depend much on the problem at hand and the ability to partition the problem in a way that is conductive to clustering.

2.1.4.4 Parallel Computing

Traditional Supercomputers

Technology

- Single processors were created to be as fast as possible.
- Peak performance was achieved with good memory bandwidth.

Benefits

- Supports sequential programming (Which many people understand)
- 30+ years of compiler and tool development
- I/O is relatively simple

Limitations

- Single high performance processors are extremely expensive
- Significant cooling requirements
- Single processor performance is reaching its asymptotic limit

Parallel Supercomputers

Technology

- Applying many smaller cost efficient processors to work on a part of the same task
- Capitalizing on work done in the microprocessor and networking markets

Benefits

- Ability to achieve performance and work on problems impossible with traditional computers.
- Exploit "off the shelf" processors, memory, disks and tape systems.
- Ability to scale to problem.
- Ability to quickly integrate new elements into systems thus capitalizing on improvements made by other markets.
- Commonly much cheaper.

Limitations

- New technology. Programmers need to learn parallel programming approaches.
- Standard sequential codes will not "just run".
- Compilers and tools are often not mature.
- I/O is not as well understood yet.

Parallel computing requires:

- Multiple processors (The workers)
- Network (Link between workers)
- Environment to create and manage parallel processing
    - Operating System
    (Administrator of the system that knows how to handle multiple workers)
    - Parallel Programming Paradigm
        - Message Passing
            - MPI
            - PVM
        - Data Parallel
            - Fortran 90 / High Performance Fortran
        - Others
            - OpenMP
            - shmem
- A parallel algorithm and a parallel program
(The decomposition of the problem into pieces that multiple workers can perform)

2.1.4.5 Parallelism in real life

The world of computing until recently was dominated by the *sequential way of thinking*. In fact, sequential processing has been very successful and has set high standards that parallel processing will have to try hard to match. Before introducing the *parallel way of thinking*, let us make sure that we have a clearer idea of what problems the sequential way of thinking may have. For example, let us consider the simple problem of assigning 0 to the 100 memory locations of an integer array *A*, initialization performed by most compilers every time arrays are used. The processor would have to execute the code typically produced by the compiler, that visits sequentially every memory location *A*[1..100] of the array and assigns the value 0 to it. This code could look like this 2:

```
i = 0;
while (i <= 100) {
i++;
A[i] = 0;
}
```

This is an operation many of you were taught in your .rst CS course. It has been observed that students in that introductory course often have some troubles coming up with such a solution, and one of the reasons may be that it seems “unnatural” to those who are not yet used to “think like the computer,” as the popular expression goes. Let us explain what we mean by that.

Consider the logically equivalent problem of an instructor distributing handouts to her 100 students in the beginning of a class period. It is very unlikely that she would walk around the class giving the handouts to each and every student in a sequential fashion:

```
take a deep breath;
while (there are students you have not visited yet) {
visit a student that you have not visited yet;
give a copy of the handouts to that student;
}
```

Such an action would be very time consuming and would occupy her for most of the class period, while the students stay idle — not to mention bored. 3 Instead, the instructor would hand the whole package to the student sitting, say, at an end of the .rst row, and then return to her previous activity. Each student will be occupied for only a short period of time: receiving the handouts when they reach him, keeping one copy, and handing the remaining copies to the nearest student that has not received handouts yet.

```
Instructor's action:
visit the first student;
give the pile with the handouts to that student;
continue teaching the class;
Student's action:
while (the pile with the handouts have not reached you)
attend the class;
pick up a copy of the handouts;
if (there is a student that has not received handouts yet)
pass up the handouts to this student;
```

This is a special parallel method used commonly in assembly lines, known as the *pipeline*, and in the classroom example is, apparently, preferable. In parallel computing terms, we say that this technique has better *load balance* than the sequential one, because the task of distributing the handouts is evenly divided among the number of persons involved. It is irrelevant here that almost certainly this habit would make the instructor wellknown around campus.

There is not, however, a single parallel way of doing this job, but several. When time is at a premium, for example, when a midterm exam is being handed out, it is important that every students gets a copy of the exam as soon as possible. In this case, the instructor may speed up this process by handing a portion of the exams to the students sitting at the end of each desk row. This process is faster than the previous one by a factor that equals, roughly, the number of desk rows. This approach is more *e.cient* than the .rst because the total amount of time to execute the same task is less than the time associated with the first.

2.1.4.6 Inter-thread communication

Parallel computers are theoretically modeled as Parallel Random Access Machines (PRAMs). The PRAM model ignores the cost of interconnection between the constituent

computing units, but is nevertheless very useful in providing upper bounds on the parallel solvability of many problems. In reality the interconnection plays a significant role.

The processors may either communicate in order to be able to cooperate in solving a problem or they may run completely independently, possibly under the control of another processor which distributes work to the others and collects results from them (a "processor farm").

Processors in a parallel computer may communicate with each other in a number of ways, including shared (either multiported or multiplexed) memory, a crossbar, a shared bus or an interconnect network of a myriad of topologies including star, ring, tree, hypercube, fat hypercube (an hypercube with more than one processor at a node), a n-dimensional mesh, etc. Parallel computers based on interconnect network need to employ some kind of routing to enable passing of messages between nodes that are not directly connected. The communication medium used for communication between the processors is likely to be hierarchical in large multiprocessor machines. Similarly, memory may be either private to the processor, shared between a number of processors, or globally shared. Systolic array is an example of a multiprocessor with fixed function nodes, local-only memory and no message routing.

Approaches to parallel computers include:

- Multiprocessing
- Computer cluster
- Parallel supercomputers
- Distributed computing
- NUMA vs. SMP vs. massively parallel computer systems
- Grid computing

2.1.4.7 Sequential Computing

Almost all computation done during the first forty years of the history of computers could be called *sequential*. One of the characteristics of sequential computation is that it employs a single processor to solve some problem. (Here, the term “problem” is used in the broad sense, i.e., performing some task.) These processors had become continuously faster (and cheaper) during the first three decades, doubling their speed every two or three years. However, due to the limit that the speed of light imposes on us, it seems extremely unlikely that we can build uni-processor computers (i.e., computers that contain only one processor) that can achieve performance signi.cantly higher than 1,000,000,000 .oating-point operations per second — usually called 1 G.ops. The unit *.ops* is a widely used measure of memory access performance. It equals the rate at which a machine can perform single-precision floating point operations, i.e., how many such operations the computer can perform in a unit of time – second in our case. As with physical quantities, computing power is measured using the notations kilo ($1K$ = 103), mega ($1M$ = 106), giga ($1G$ = 109) and tera ($1T$ = 1012). If the computer does not have the So, *parallel computation* is de.ned as the practice of employing a (usually large1) number of cooperating processors, communicating among themselves to solve large problems fast. It has quickly become an important area in computer

science. During the past .ve years, parallel computation has grown so wide and strong that most of the research conducted in the .elds of design and analysis of algorithms, computer languages, computer applications and computer architectures are within its context.

2.1.4.8 Parallel vs sequential solutions – The magic box

Probably the first approach one may think for designing a parallel algorithm is to modify and parallelize an existing sequential one. It would be nice if someone had written a program s2p.c that takes as input a sequential program and produces an equivalent parallel program which runs much faster and exhibits good load balancing (see figure 1.4). After all, this is not very difficult to do for the initialization problem we saw earlier: one has only.

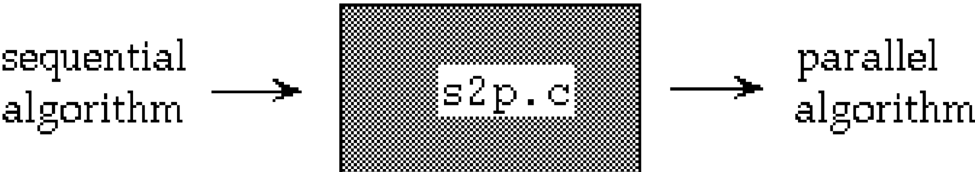


Figure 1.4: The magic box that converts sequential code to parallel. to write a program that can recognize a code fragment with the general structure

```
var1 = val1;
while (var1 < val2) {
var1++;
var2[var1] = val3;
}
```

and convert it to an equivalent parallel code which divides up the operations performed by the number of available processors. This simple solution, of course, requires that there are no data dependencies among operations assigned to different processors. Unfortunately, it seems that very few sequential algorithms can be modified to produce significantly faster parallel algorithms. At the same time, even fewer of them have obvious or simple parallel modifications. Moreover, problems that happen to have simple sequential solutions do not necessarily have a practical parallel solution, sometimes do not have an efficient parallel solution at all!

An interesting example of the latter case is *depth-.rst search*, a technique which has applications in almost every area of computer science. It has been used as a basis for many problem solutions and, as is well known, it has a simple linear-time sequential implementation. For example, it is used as a technique to search the nodes of a graph for some property. 5 However, despite considerable research efforts, an efficient parallel implementation for this technique has not yet been found. 6 What is worse, there is some evidence that an efficient parallel algorithm for this problem may not even exist!

As a result of that, all these sequential algorithms that have been developed using depth first search cannot be easily converted to parallel ones through some magic box. We have to find new parallel solutions that use different methods.

Summary:

The most prominent two eras of computing are : sequential and parallel era. Parallel computing is the simultaneous execution of the *same task* (split up and specially adapted) on multiple processors in order to obtain faster results. Almost all computation done during the first forty years of the history of computers could be called *sequential*. One of the characteristics of sequential computation is that it employs a single processor to solve some problem.

Model Questions:

1. Briefly explain the parallel computing?
2. Differentiate the parallel and sequential computing?

References:

1. SCIENTIFIC COMPUTING: An Introductory Survey, Second Edition by Michael T. Heath, published by McGraw-Hill, New York, 2002.

AUTHOR:

B.M.REDDY M.Tech. (HBTI, Kanpur)

Lecturer, Centre for Biotechnology

Acharya Nagarjuna University.

Lesson 2.2.1

SYSTEM SOFTWARE

Objective
2.2.1.1 Introduction
2.2.1.2 Programming Environments
2.2.1.3 A Unifying Framework
2.2.1.4 Operating System
2.2.1.5 Driver
2.2.1.6 BIOS
2.2.1.7 TP Monitor
2.2.1.8 Communications Protocols
2.2.1.9 DBMS
2.2.1.10 Programming Language
summary
Model Questions
References

Objective

- **To know what a system software is**
- **To study the various components of system software**

2.2.1.1 Introduction

If it were not for system software, all programming would be done in machine code, and applications programs would directly use hardware resources such as input-output devices and physical memory. In such an environment, much of a programmers time would be spent on the relatively clerical problems of program preparation and translation, and on the interesting but unproductive job of reinventing effective ways to use the hardware. System software exists to relieve programmers of these jobs, freeing their time for more productive activities. As such, system software can be viewed as establishing a programming environment which makes more productive use of the programmer's time than that provided by the hardware alone.

2.2.1.2 Programming Environments

The term *programming environment* is sometimes reserved for environments containing language specific editors and source level debugging facilities; here, the term will be used in its broader sense to refer to all of the hardware and software in the environment used by the programmer. All programming can therefore be properly described as takin place in a programming environment.

Programming environments may vary considerably in complexity. An example of a simple environment might consist of a text editor for program preparation, an assembler for translating programs to machine language, and a simple operating system consisting of input-output drivers and a file system. Although card input and non-interactive operation characterized most early computer systems, such simple environments were supported on early experimental time-sharing systems by 1963.

Although such simple programming environments are a great improvement over the bare hardware, tremendous improvements are possible. The first improvement which comes to mind is the use of a high level language instead of an assembly language, but this implies other changes. Most high level languages require more complicated run-time support than just input-output drivers and a file system. For example, most require an extensive library of predefined procedures and functions, many require some kind of automatic storage management, and some require support for concurrent execution of threads, tasks or processes within the program.

Many applications require additional features, such as window managers or elaborate file access methods. When multiple applications coexist, perhaps written by different programmers, there is frequently a need to share files, windows or memory segments between applications. This is typical of today's electronic mail, database, and spreadsheet applicatons, and the programming environments that support such applications can be extremely complex, particularly if they attempt to protect users from malicious or accidental damage caused by program developers or other users.

A programming environment may include a number of additional features which simplify the programmer's job. For example, library management facilities to allow programmers to extend the set of predefined procedures and functions with their own routines. Source level debugging facilities, when available, allow run-time errors to be interpreted in terms of the source program instead of the machine language actually run by the hardware. As a final example, the text editor may be language specific, with commands which operate in terms of the syntax of the language being used, and mechanisms which allow syntax errors to be detected without leaving the editor to compile the program.

Historical Note

Historically, system software has been viewed in a number of different ways since the invention of computers. The original computers were so expensive that their use for such clerical jobs as language translation was viewed as a dangerous waste of scarce resources. Early system developers seem to have consistently underestimated the difficulty of producing working programs, but it did not take long for them to realize that letting the computer spend a few minutes on the clerical job of assembling a user program was less expensive than having the programmer hand assemble it and then spend hours of computer time debugging it. As a result, by 1960, assembly language was widely accepted, the new high level language, FORTRAN, was attracting a growing user community, and

there was widespread interest in the development of new languages such as Algol, COBOL, and LISP.

Early operating systems were viewed primarily as tools for efficiently allocating the scarce and expensive resources of large central computers among numerous competing users. Since compilers and other program preparation tools frequently consumed a large fraction of an early machine's resources, it was common to integrate these into the operating system. With the emergence of large scale general purpose operating systems in the mid 1960's, however, the resource management tools available became powerful enough that they could efficiently treat the resource demands of program preparation the same as any other application.

The separation of program preparation from program execution came to pervade the computer market by the early 1970's, when it became common for computer users to obtain editors, compilers, and operating systems from different vendors. By the mid 1970's, however, programming language research and operating system development had begun to converge. New operating systems began to incorporate programming language concepts such as data types, and new languages began to incorporate traditional operating system features such as concurrent processes. Thus, although a programming language must have a textual representation, and although an operating system must manage physical resources, both have, as their fundamental purpose, the support of user programs, and both must solve a number of the same problems.

The minicomputer and microcomputer revolutions of the mid 1960's and the mid 1970's involved, to a large extent, a repetition of the earlier history of mainframe based work. Thus, early programming environments for these new hardware generations were very primitive; these were followed by integrated systems supporting a single simple language (typically some variant of BASIC on each generation of minicomputer and microcomputer), followed by general purpose operating systems for which many language implementations and editors are available, from many different sources.

The world of system software has varied from the wildly competitive to domination by large monopolistic vendors and pervasive standards. In the 1950's and early 1960's, there was no clear leader and there were a huge number of wildly divergent experiments. In the late 1960's, however, IBM's mainframe family, the System 360, running IBM's operating system, OS/360, emerged as a monopolistic force that persists to the present in the corporate data processing world (the IBM 390 Enterprise Server is the current flagship of this line, running the VM operating system).

The influence of IBM's near monopoly of the mainframe marketplace cannot be underestimated, but it was not total, and in the emerging world of minicomputers, there was wild competition in the late 1960's and early 1970's. The Digital Equipment Corporation PDP-11 was dominant in the 1970's, but never threatened to monopolize the market, and there were a variety of different operating systems for the 11. In the 1980's,

however, variations on the Unix operating system originally developed at Bell Labs began to emerge as a standard development environment, running on a wide variety of computers ranging from minicomputers to supercomputers, and featuring the new programming language C and its descendant C++.

The microcomputer marketplace that emerged in the mid 1970's was quite diverse, but for a decade, most microcomputer operating systems were rudimentary, at best. Early versions of Mac OS and Microsoft Windows presented sophisticated user interfaces, but on versions prior to about 1995 these user interfaces were built on remarkably crude underpinnings.

The marketplace of the late 1990's, like the marketplace of the late 1960's, came to be dominated by a monopoly, this time in the form of Microsoft Windows. The chief rivals are MacOS and Linux, but there is yet another monopolistic force hidden behind all three operating systems, the pervasive influence of Unix and C. MacOS X is fully Unix compatable. Windows NT offers full compatability, and so, of course, does Linux. Much of the serious development work under all three systems is done in C++, and new languages such as Java seem to be simple variants on the theme of C++. It is interesting to ask, when we will we have a new creastive period when genuinely new programming environments will be developed the way they were on the mainframes of the early 1960's or the minicomputers of the mid 1970's?

2.2.1.3 A Unifying Framework

In all programming environments, from the most rudimentary to the most advanced, it is possible to identify two distinct components, the program preparation component and the program execution component. On a bare machine, the program preparation component consists of the switches or push buttons by which programs and data may be entered into the memory of the machine; more advanced systems supplement this with text editors, compilers, assemblers, object library managers, linkers, and loaders. On a bare machine, the program execution component consists of the hardware of the machine, the central processors, any peripheral processors, and the various memory resources; more advanced systems supplement this with operating system services, libraries of predefined procedures, functions and objects, and interpreters of various kinds.

Within the program execution component of a programming environment, it is possible to distinguish between those facilities needed to support a single user process, and those which are introduced when resources are shared between processes. Among the facilities which may be used to support a single process environment are command language interpreters, input-output, file systems, storage allocation, and virtual memory. In a multiple process environment, processor allocation, interprocess communication, and resource protection may be needed. Figure 1.1 lists and classifies these components.
Editors
Compilers

Assemblers Program Preparation
Linkers
Loaders
Command Languages
Sequential Input/Output
Random Access Input/Output
File Systems Used by a Single Process
Window Managers
Storage Allocation
Virtual Memory
------------------------------ Program Execution Support
Process Scheduling
Interprocess Communication
Resource Sharing Used by Multiple Processes
Protection Mechanisms

The distinction between preparation and execution is the basis of the division between the first and second parts, while the distinction between single process and multiple process systems is the basis of the division between the second and third parts.

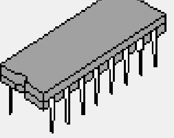
SYSTEM SOFTWARE

OPERATING SYSTEM
Manages the computer system. Provides file, task and job management. All application programs "talk to" the operating system.

Windows, DOS, UNIX, Linux, Mac OS, OS/2, MVS and OS/400 are examples.

DRIVER
Software that supports a peripheral device, such as a display adapter or CD-ROM. The driver contains the detailed machine language necessary to activate all functions in the device. The operating system commands the driver, which in turn commands the hardware device.

BIOS (Basic Input/Output System)
In a PC, a set of software routines built into a chip that boots the machine and serves as an interface between the drivers and the peripheral devices.



TP MONITOR
Mainframe/midrange program that distributes input from multiple terminals to the appropriate application. This function is also provided in LAN operating systems.

CICS is widely used in IBM mainframes, and Tuxedo and Encina are widely used in UNIX systems.

NETWORK OPERATING SYSTEM
Manage traffic and security between clients and servers in a network. Examples are UNIX, Linux, NetWare, Windows NT and Windows 2000.

COMMUNICATIONS PROTOCOL
Set of rules, formats and functions for sending data across the network. There are many protocol layers starting at the top application layer to the bottom physical layer

Popular transport protocols are TCP/IP, NetBEUI and IPX/SPX. Popular data link protocols (access methods) used to transmit data from point to point are Ethernet, Token Ring, SDLC and RS-232.

MESSAGING PROTOCOL
Set of rules, formats and functions for sending, storing and forwarding e-mail in a network. The major messaging protocols are SMTP (Internet), SNADS, MHS, X.400, cc:Mail and Microsoft Mail.

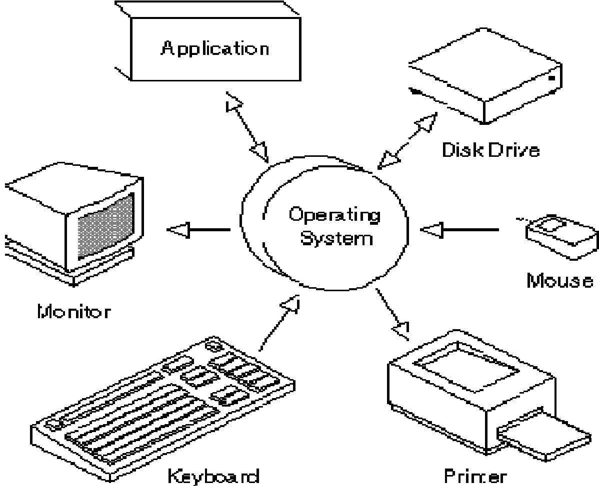
DATABASE MANAGEMENT SYSTEM (DBMS)
Manages the storage, retrieval, security and integrity of the database. A DBMS may provide interactive data entry, updating, query and reporting or rely entirely on the application program for such functions. The DBMS may reside in a mainframe or in a file server in a client/server architecture. Most DBMSs support the SQL language, and many include a complete programming language for application development.

Examples of popular mainframe and client/server DBMSs are DB2, Oracle, Sybase, SQL Server and Informix. Access, dBASE and Paradox are used in PCs.

PROGRAMMING LANGUAGE
Translate source language into machine language using assemblers, compilers, interpreters and application generators. All system software and application software must be programmed in a programming language and turned into machine language for execution.

Examples of programming languages are assembly language, BASIC, FORTRAN, C, C++, Pascal, dBASE, Paradox, Visual Basic and COBOL.

2.2.1.4 Operating System



The most important program that runs on a computer. Every general-purpose computer must have an operating system to run other programs. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

For large systems, the operating system has even greater responsibilities and powers. It is like a traffic cop -- it makes sure that different programs and users running at the same time do not interfere with each other. The operating system is also responsible for *security*, ensuring that unauthorized users do not access the system.

Operating systems can be classified as follows:

- **multi-user :** Allows two or more users to run programs at the same time. Some operating systems permit hundreds or even thousands of concurrent users.
- **multiprocessing :** Supports running a program on more than one CPU.
- **multitasking :** Allows more than one program to run concurrently.
- **multithreading :** Allows different parts of a single program to run concurrently.
- **real time:** Responds to input instantly. General-purpose operating systems, such as DOS and UNIX, are not real-time.

Operating systems provide a software platform on top of which other programs, called *application programs,* can run. The application programs must be written to run on top of a particular operating system. Your choice of operating system, therefore, determines to a great extent the applications you can run. For PCs, the most popular operating systems are DOS, OS/2, and Windows, but others are available, such as Linux.

As a user, you normally interact with the operating system through a set of commands. For example, the DOS operating system contains commands such as COPY and RENAME for copying files and changing the names of files, respectively. The commands are accepted and executed by a part of the operating system called the command processor or command line interpreter. Graphical user interfaces allow you to enter commands by pointing and clicking at objects that appear on the screen.

Kinds of operating systems

Functionality

Operating systems can be grouped according to functionality: operating systems for supercomputing, render farms, mainframes, servers, workstations, desktops, handheld devices, real time systems, or embedded systems.

Supercomputing is primarily scientific computing, usually modelling real systems in nature. Render farms are collections of computers that work together to render animations

and special effects. Work that previously required supercomputers can be done with the equivalent of a render farm.

Mainframes used to be the primary form of computer. Mainframes are large centralized computers. At one time they provided the bulk of business computing through **time sharing**. Mainframes and **mainframe replacements** (powerful computers or clusters of computers) are still useful for some large scale tasks, such as centralized billing systems, inventory systems, database operations, etc. When mainframes were in widespread use, there was also a class of computers known as **minicomputers** which were smaller, less expensive versions of mainframes for businesses that couldn't afford true mainframes.

Servers are computers or groups of computers used for internet serving, intranet serving, print serving, file serving, and/or application serving. Servers are also sometimes used as mainframe replacements.

Desktop operating systems are used for **personal computers**.

Workstations are more powerful versions of personal computers. Often only one person uses a particular workstation (like desktops) and workstations often run a more powerful version of a desktop operating system, but workstations run on more powerful hardware and often have software associated with larger computer systems.

Handheld operating systems are much smaller and less capable than desktop operating systems, so that they can fit into the limited memory of handheld devices.

Real time operating systems (RTOS) are specifically designed to respond to events that happen in real time. This can include computer systems that run factory floors, computer systems for emergency room or intensive care unit equipment (or even the entire ICU), computer systems for air traffic control, or embedded systems. RTOSs are grouped according to the response time that is acceptable (seconds, milliseconds, microseconds) and according to whether or not they involve systems where failure can result in loss of life.

Embedded systems are combinations of processors and special software that are inside of another device, such as the electronic ignition system on cars.

2.2.1.5 Driver

Device management controls peripheral devices by sending them commands in their own proprietary language. The software routine that knows how to deal with each device is called a "driver." The operating system contains all the drivers for the peripherals attached to the computer. When a new peripheral is added, that device's driver is installed into the

operating system. A hardware device (typically a transistor) that provides signals or electrical current to activate a transmission line or display screen pixel.

Also called a "device driver," it is a program routine that links the operating system to a peripheral device. Written by programmers who understand the peripheral hardware's command language and characteristics, the driver contains the precise machine language necessary to perform the functions requested by the application. When a new peripheral device is added to the computer, such as a display adapter, its driver must be installed in order to use it. The application calls the operating system, the operating system calls the driver, and the driver makes the device work (it "drives" the device). Routines that perform internal functions, such as memory managers and disk caches, are also drivers.

2.2.1.6 BIOS

(bī´ōs) Acronym for ***b**asic **i**nput/**o**utput **s**ystem*, the built-in software that determines what a computer can do without accessing programs from a disk. On PCs, the BIOS contains all the code required to control the keyboard, display screen, disk drives, serial communications, and a number of miscellaneous functions.

The BIOS is typically placed in a ROM chip that comes with the computer (it is often called a *ROM BIOS*). This ensures that the BIOS will always be available and will not be damaged by disk failures. It also makes it possible for a computer to boot itself. Because RAM is faster than ROM, though, many computer manufacturers design systems so that the BIOS is copied from ROM to RAM each time the computer is booted. This is known as *shadowing*.

Many modern PCs have a *flash BIOS,* which means that the BIOS has been recorded on a flash memory chip, which can be updated if necessary.

The PC BIOS is fairly standardized, so all PCs are similar at this level (although there are different BIOS versions). Additional DOS functions are usually added through software modules. This means you can upgrade to a newer version of DOS without changing the BIOS.

PC BIOSes that can handle Plug-and-Play (PnP) devices are known as *PnP BIOSes*, or *PnP-aware BIOSes*. These BIOSes are always implemented with flash memory rather than ROM.

2.2.1.7 TP Monitor

Short for ***t**ransaction **p**rocessing monitor,* a program that monitors a transaction as it passes from one stage in a process to another. The TP monitor's purpose is to ensure that the transaction processes completely or, if an error occurs, to take appropriate actions.

TP monitors are especially important in three-tier architectures that employ load balancing because a transaction may be forwarded to any of several servers. In fact, many TP monitors handle all the load balancing operations, forwarding transactions to different servers based on their availability.

Messaging Protocols

Also called: Messaging Standards, Email Protocols, Communications Standards, Electronic Messaging Protocols, e-mail Protocols, Electronic Messaging Standards, Communications Protocols, and Mail Protocols

messaging system

Software that provides an electronic mail delivery system. It is made up of the following functional components, which may be packaged together or independently.

Mail User Agent

The mail user agent (MUA or UA) is the client e-mail program, such as Outlook, Eudora or Mac Mail, that submits and receives the message.

Message Transfer Agent

The message transfer agent (MTA) forwards the message to another mail server or delivers it to its own message store (MS). Sendmail is the most widely used MTA on the Internet. In a large enterprise, there may be several MTA servers (mail servers) dedicated to Internet e-mail while others support internal e-mail.

Message Store

The message store (MS) holds the mail until it is selectively retrieved and deleted by an access server. In the Internet world, a delivery agent writes the messages from the MTA to the message store, and typical access servers are either POP or IMAP servers.

The Internet's SMTP

Internet e-mail, the most ubiquitous messaging system in the world, is based on the SMTP protocol. Prior to the Internet's enormous growth in the late 1990s, numerous proprietary messaging systems were widely used, including cc:Mail, Microsoft Mail, PROFS and DISOSS. See messaging middleware and SMTP.

2.2.1.8 Communications Protocols

email, newsgroups and chat

These are the messaging protocols that allow users to communicate both asynchronously (sender and receiver aren't required to both be connected to the Internet at the same time; e.g. email) and synchronously (as with chatting in "real time").

Email
This method of Internet communication has become the standard. A main computer acts as a "post office" by sending and receiving mail for those who have accounts. This mail can be retrieved through any number of email software applications (MS Outlook, Eudora, etc.) or from Web based email accounts (Yahoo, Hotmail). Email is an example of asynchronous Internet communication.

Email also provides the ability to access email lists. You can subscribe to an email list covering any number of topics or interests and will receive messages posted by other subscribers. Email communities evolve from interaction between subscribers who have similar interests or obsessions.

Usenet
Usenet is something like a bulletin board or an email list without the subscription. Anyone can post a message to or browse through a Usenet newsgroup. Usenet messages are retained on the serving computer only for a predetermined length of time and then are automatically deleted, whereas email list messages are retained on the serving computer until the account holder downloads them. Many email applications, as well as Web browsers, allow you to set up Usenet newsgroup accounts.

IRC (Internet Relay Chat)

This protocol allows for synchronous communication: users on different computers anywhere in the world can communicate in "real time" or simultaneously. You can instantly see a response to a typed message by several people at the same time. This protocol requires a special software application that can be downloaded from the Web, generally for free.

2.2.1.9 DBMS
A collection of programs that enables you to store, modify, and extract information from a database. There are many different types of DBMSs, ranging from small systems that run on personal computers to huge systems that run on mainframes. The following are examples of database applications:

- computerized library systems

- automated teller machines
- flight reservation systems
- computerized parts inventory systems

From a technical standpoint, DBMSs can differ widely. The terms *relational*, *network*, *flat,* and *hierarchical* all refer to the way a DBMS organizes information internally. The internal organization can affect how quickly and flexibly you can extract information.

Requests for information from a database are made in the form of a *query*, which is a stylized question. For example, the query

SELECT ALL WHERE NAME = "SMITH" AND AGE > 35

requests all records in which the NAME field is SMITH and the AGE field is greater than 35. The set of rules for constructing queries is known as a *query language*. Different DBMSs support different query languages, although there is a semi-standardized query language called *SQL (structured query language)*. Sophisticated languages for managing database systems are called *fourth-generation languages*, or *4GLs* for short.

The information from a database can be presented in a variety of formats. Most DBMSs include a *report writer program* that enables you to output data in the form of a report. Many DBMSs also include a graphics component that enables you to output information in the form of graphs and charts.

2.2.1.10 Programming Language

Introduction to Programming

A program is a set of instructions that tell the computer to do various things; sometimes the instruction it has to perform depends on what happened when it performed a previous instruction. This section gives an overview of the two main ways in which you can give these instructions, or “commands” as they are usually called. One way uses an *interpreter*, the other a *compiler*. As human languages are too difficult for a computer to understand in an unambiguous way, commands are usually written in one or other languages specially designed for the purpose.

Interpreters

With an interpreter, the language comes as an environment, where you type in commands at a prompt and the environment executes them for you. For more complicated programs, you can type the commands into a file and get the interpreter to load the file and execute the commands in it. If anything goes wrong, many interpreters will drop you into a debugger to help you track down the problem.

The advantage of this is that you can see the results of your commands immediately, and mistakes can be corrected readily. The biggest disadvantage comes when you want to share your programs with someone. They must have the same interpreter, or you must have some way of giving it to them, and they need to understand how to use it. Also users may not appreciate being thrown into a debugger if they press the wrong key! From a performance point of view, interpreters can use up a lot of memory, and generally do not generate code as efficiently as compilers

Compilers

Compilers are rather different. First of all, you write your code in a file (or files) using an editor. You then run the compiler and see if it accepts your program. If it did not compile, grit your teeth and go back to the editor; if it did compile and gave you a program, you can run it either at a shell command prompt or in a debugger to see if it works properly. [1]

Obviously, this is not quite as direct as using an interpreter. However it allows you to do a lot of things which are very difficult or even impossible with an interpreter, such as writing code which interacts closely with the operating system--or even writing your own operating system! It is also useful if you need to write very efficient code, as the compiler can take its time and optimize the code, which would not be acceptable in an interpreter. Moreover, distributing a program written for a compiler is usually more straightforward than one written for an interpreter--you can just give them a copy of the executable, assuming they have the same operating system as you.

Compiled languages include Pascal, C and C++. C and C++ are rather unforgiving languages, and best suited to more experienced programmers; Pascal, on the other hand, was designed as an educational language, and is quite a good language to start with. As the edit-compile-run-debug cycle is rather tedious when using separate programs, many commercial compiler makers have produced Integrated Development Environments (IDEs for short). FreeBSD does not include an IDE in the base system, but devel/kdevelop is available in the ports tree and many use **Emacs** for this purpose

Language Levels

All students are placed according to communicative ability in the language. There are six levels of language proficiency as outlined in the pyramid below:

LEVEL 1: Low-Beginner Unable to function in the spoken language.

LEVEL 1: Mid-Beginner

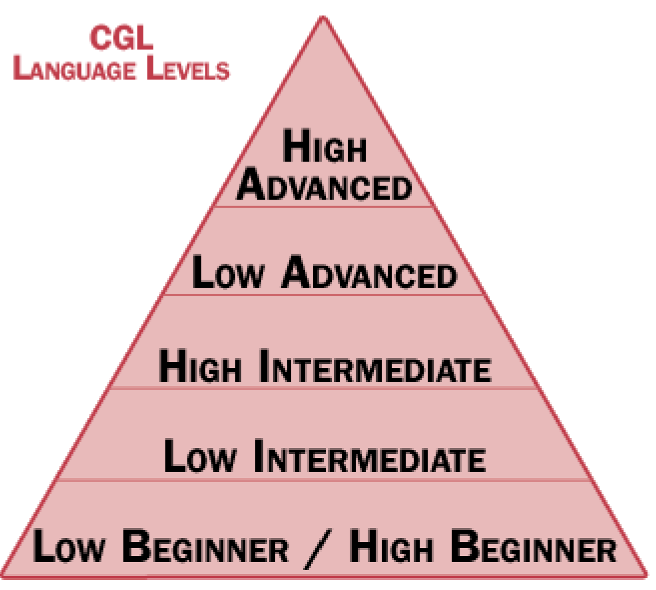
Able to function in only a very limited capacity by using a number of memorized words and phrases.

LEVEL 2: High-Beginner

Able to satisfy immediate needs with learned utterances. Does not speak in complete sentences.

LEVEL 3: Low-Intermediate

Able to handle a variety of tasks in previously learned, uncomplicated social situations. Speech is generally limited to the present tense and sentences may not always be complete.



LEVEL 4: High-Intermediate

Able to successfully handle a variety of communicative tasks in uncomplicated social situations. Can ask and respond to questions, make requests for information, and express personal meaning, but responses may still contain hesitancy and grammatical inaccuracies.

LEVEL 5: Low-Advanced

Able to narrate and describe in major time frames and can talk about a wide range of concrete social and work topics. May still make errors with basic grammatical structures, but they have a much stronger control of the grammatical system than the Intermediate level.

LEVEL 6: High-Advanced

Able to participate actively in most formal and informal exchanges on a variety of topics with linguistic ease, confidence, and competence. May still make some high-level grammatical and translation errors.

Summary

It quite probable to run into these two terms: system software and application software. The distinction between the two is important. Without the former, your computer won't

run. And without the latter, your computer—no matter how powerful—won't do much to help run your business. System software constitutes the following such as Operating system, drivers, protocols, DBMS and so on.

Model Questions

1. What is System software? What are its constituents?
2. How many levels of Programming Languages are there? What are they?
3. Explain communication and messaging protocols.

References

1. http://www.valenciacc.edu/institute/cgl/levels.asp
2. http://computing-dictionary.thefreedictionary.com/system%20software
3. System Software and Software Systems: Systems Methodology for System Software by R. Rus

Author:-
Asha Smitha. B.
Centre for Biotechnology
Acharya Nagarjuna University

Lesson 2.2.2

GENERATIONS OF COMPUTERS

CONTENTS

Objective
2.2.2.1 Introduction
2.2.2.2 The Mechanical Era
2.2.2.3 First Generation Electronic Computers
2.2.2.4 Second Generation Computers

2.2.2.5 Second Generation - Transistors
2.2.2.6 Third Generation - Integrated Circuits
2.2.2.7 Fourth Generation - Microprocessors
2.2.2.8 Fifth Generation Computers
2.2.2.9 Sixth Generation of Computers
2.2.2.10 Future Computer Generations
Summary
Model Questions
References

Objective

- To understand the evolution of computers through various generations.
- To know the chronological improvement in the computer technology

2.2.2.1 Introduction

Since humanity got the idea not to limit itself any longer to the usage of producing and receiving sounds for the purpose of mutual understanding, the rise of visual communication assumed large proportions. From a humble start, scratching stone or engraving leaves of plants, writers in due course proceeded in applying newer means again and again: hides of animals, gall-nut tincture, melting lead and laser radiants marked a development of increasing refinement and differentiation.

Yet, when faced with the greatest step forward in communication facilities, that of electronic data processing, the application of it for producing script seemed to require a start from scratch again. A method that was based on only two available tokens presented the most intractable medium mankind ever had been forced to handle. Nevertheless, the advantages offered by the speed of transfer were so great that generations of people got convinced that the elements of Morse-code, dots and dashes only, were a comfortable way of expression, and presented a real progress. Still more rigid were the demands of the

computer: only 0 or 1, on or off, flip or flop, yin or yang, to be or not to be, were the values that the atoms of understanding, the bits, were permitted to assume. If nothing had happened later on, the Stone Age had returned by way of the computer.

The evolution of digital computing is often divided into *generations*. Each generation is characterized by dramatic improvements over the previous generation in the technology used to build computers, the internal organization of computer systems, and programming languages. Although not usually associated with computer generations, there has been a steady improvement in algorithms, including algorithms used in computational science. The following history has been organized using these widely recognized generations as mileposts.

2.2.2.2 The Mechanical Era (1623--1945)

The idea of using machines to solve mathematical problems can be traced at least as far as the early 17th century. Mathematicians who designed and implemented calculators that were capable of addition, subtraction, multiplication, and division included Wilhelm Schickhard, Blaise Pascal (Pascal's contribution to computing was recognized by computer scientist Nicklaus Wirth, who in 1972 named his new computer language Pascal (and insisted that it be spelled Pascal, not PASCAL)) and Gottfried Leibnitz.

The first multi-purpose, i.e. *programmable*, computing device was probably Charles Babbage's Difference Engine, which was begun in 1823 but never completed. A more ambitious machine was the Analytical Engine. It was designed in 1842, but unfortunately it also was only partially completed by Babbage. Babbage was truly a man ahead of his time: many historians think the major reason he was unable to complete these projects was the fact that the technology of the day was not reliable enough. In spite of never building a complete working machine, Babbage and his colleagues, most notably Ada (Another pioneer with a programming language named after her. Naming languages after mathematicians is somewhat of a tradition in computer science. Other such languages include Russel, Euclid, Turning, and Goedel.) Countess of Lovelace, recognized several important programming techniques, including conditional branches, iterative loops and index variables.

A machine inspired by Babbage's design was arguably the first to be used in computational science. George Scheutz read of the difference engine in 1833, and along with his son Edvard Scheutz began work on a smaller version. By 1853 they had constructed a machine that could process 15-digit numbers and calculate fourth-order differences. Their machine won a gold medal at the Exhibition of Paris in 1855, and later they sold it to the Dudley Observatory in Albany, New York, which used it to calculate the orbit of Mars. One of the first commercial uses of mechanical computers was by the US Census Bureau, which used punch-card equipment designed by Herman Hollerith to tabulate data for the 1890 census. In 1911 Hollerith's company merged with a competitor to found the corporation which in 1924 became International Business Machines.

When in the fifties the first computers appeared, a coding system had to be developed for letters and digits, and if possible also for some punctuation marks and other small stuff. But there existed yet a tradition of punched cards and chain printers, where 48 characters were the accepted limit. Though a 6-bit code allows for 64, it lasted several years before that number was adopted for computer hardware and software alike. For generations of computers this continued to be the upper limit, with the result that texts were restricted to capital letters, and in fact, to English. To Scandinavians it was allowed to sacrifice some special characters to include their own extra letters

2.2.2.3 First Generation Electronic Computers

Three machines have been promoted at various times as the first electronic computers. These machines used electronic switches, in the form of vacuum tubes, instead of electromechanical relays. In principle the electronic switches would be more reliable, since they would have no moving parts that would wear out, but the technology was still new at that time and the tubes were comparable to relays in reliability. Electronic components had one major benefit, however: they could ``open'' and ``close'' about 1,000 times faster than mechanical switches.

The earliest attempt to build an electronic computer was by J. V. Atanasoff, a professor of physics and mathematics at Iowa State, in 1937. Atanasoff set out to build a machine that would help his graduate students solve systems of partial differential equations. By 1941 he and graduate student Clifford Berry had succeeded in building a machine that could solve 29 simultaneous equations with 29 unknowns. However, the machine was not programmable, and was more of an electronic calculator.

A second early electronic machine was Colossus, designed by Alan Turing for the British military in 1943. Turing's main contribution to the field of computer science was the idea of the Turing machine, a mathematical formalism widely used in the study of computable functions. The existence of Colossus was kept secret until long after the war ended, and the credit due to Turing and his colleagues for designing one of the first working electronic computers was slow in coming.

The first general purpose programmable electronic computer was the Electronic Numerical Integrator and Computer (ENIAC), built by J. Presper Eckert and John V. Mauchly at the University of Pennsylvania. Work began in 1943, funded by the Army Ordnance Department, which needed a way to compute ballistics during World War II. The machine wasn't completed until 1945, but then it was used extensively for calculations during the design of the hydrogen bomb. By the time it was decommissioned in 1955 it had been used for research on the design of wind tunnels, random number generators, and weather prediction. Eckert, Mauchly, and John von Neumann, a consultant to the ENIAC project, began work on a new machine before ENIAC was finished. The main contribution of EDVAC, their new project, was the notion of a *stored program.* There is some controversy over who deserves the credit for this idea, but none over how important

the idea was to the future of general purpose computers. ENIAC was controlled by a set of external switches and dials; to change the program required physically altering the settings on these controls. These controls also limited the speed of the internal electronic operations. Through the use of a memory that was large enough to hold both instructions and data, and using the program stored in memory to control the order of arithmetic operations, EDVAC was able to run orders of magnitude faster than ENIAC. By storing instructions in the same medium as data, designers could concentrate on improving the internal structure of the machine without worrying about matching it to the speed of an external control.

Regardless of who deserves the credit for the stored program idea, the EDVAC project is significant as an example of the power of interdisciplinary projects that characterize modern computational science. By recognizing that functions, in the form of a sequence of instructions for a computer, can be encoded as numbers, the EDVAC group knew the instructions could be stored in the computer's memory along with numerical data. The notion of using numbers to represent functions was a key step used by Goedel in his incompleteness theorem in 1937, work which von Neumann, as a logician, was quite familiar with. Von Neumann's background in logic, combined with Eckert and Mauchly's electrical engineering skills, formed a very powerful interdisciplinary team.

Software technology during this period was very primitive. The first programs were written out in machine code, i.e. programmers directly wrote down the numbers that corresponded to the instructions they wanted to store in memory. By the 1950s programmers were using a symbolic notation, known as assembly language, then hand-translating the symbolic notation into machine code. Later programs known as assemblers performed the translation task.

As primitive as they were, these first electronic machines were quite useful in applied science and engineering. Atanasoff estimated that it would take eight hours to solve a set of equations with eight unknowns using a Marchant calculator, and 381 hours to solve 29 equations for 29 unknowns. The Atanasoff-Berry computer was able to complete the task in under an hour. The first problem run on the ENIAC, a numerical simulation used in the design of the hydrogen bomb, required 20 seconds, as opposed to forty hours using mechanical calculators. Eckert and Mauchly later developed what was arguably the first commercially successful computer, the UNIVAC; in 1952, 45 minutes after the polls closed and with 7% of the vote counted, UNIVAC predicted Eisenhower would defeat Stevenson with 438 electoral votes (he ended up with 442).

General Features of First Generation Computers

1. The First Generation was from 1946 to 1956
   - Computers in this generation did from 2,000 to 16,000 additions per second
   - Had main memory from 100 bytes to 2 kilobytes (2,000 bytes)
2. All computers of this generation used vacuum tubes to perform calculations

- Vacuum tubes are expensive because of the amount of materials and skill needed to make them.
- Vacuum tubes get hot and burn out light an incandescent light bulb.

3. All computers in this generation where very large machines
   - Needed special rooms to house them with air conditioning because of the heat generated by the vacuum tubes
   - All required specially trained technicians to run and maintain them

2.2.2.4 **Second Generation Computers**

The second generation saw several important developments at all levels of computer system design, from the technology used to build the basic circuits to the programming languages used to write scientific applications.

Electronic switches in this era were based on discrete diode and transistor technology with a switching time of approximately 0.3 microseconds. The first machines to be built with this technology include TRADIC at Bell Laboratories in 1954 and TX-0 at MIT's Lincoln Laboratory. Memory technology was based on magnetic cores which could be accessed in random order, as opposed to mercury delay lines, in which data was stored as an acoustic wave that passed sequentially through the medium and could be accessed only when the data moved by the I/O interface.

Important innovations in computer architecture (The term ``computer architecture'' generally refers to aspects of a computer's internal organization that are visible to programmers or compiler writers; see Chapter CA.) included index registers for controlling loops and floating point units for calculations based on real numbers. Prior to this accessing successive elements in an array was quite tedious and often involved writing self-modifying code (programs which modified themselves as they ran; at the time viewed as a powerful application of the principle that programs and data were fundamentally the same, this practice is now frowned upon as extremely hard to debug and is impossible in most high level languages). Floating point operations were performed by libraries of software routines in early computers, but were done in hardware in second generation machines.

During this second generation many high level programming languages were introduced, including FORTRAN (1956), ALGOL (1958), and COBOL (1959). Important commercial machines of this era include the IBM 704 and its successors, the 709 and 7094. The latter introduced I/O processors for better throughput between I/O devices and main memory.

The second generation also saw the first two supercomputers designed specifically for numeric processing in scientific applications. The term ``supercomputer'' is generally reserved for a machine that is an order of magnitude more powerful than other machines of its era. Two machines of the 1950s deserve this title. The Livermore Atomic Research Computer (LARC) and the IBM 7030 (aka Stretch) were early examples of machines that

overlapped memory operations with processor operations and had primitive forms of parallel processing.

General Features of the Second Generation

1. From 1959 to around 1965
2. Smaller, faster, and more reliable than the First Generation of computers
    - Used transistors instead of vacuum tubes for performing calculations
    - 6,000 to 3,000,000 operations per second
    - 6 kilobytes to 1.3 megabytes of main memory
    - Contained in four cabinets about 6 feet high by 4 feet wide, each weighing 250 pounds
3. Cost about one-tenth the price of a First Generation computer
4. Computers become common in larger businesses and universities

2.2.2.5 Second Generation - Transistors

Transistors

1. Invented in 1947 by William Shockley, John Bardeen, and William Brattain
    - Picture of the first transistor
    - Was made of silicon
    - Lead solid state to solid state electronics
2. Advantages of a transistor when compared to a vacuum tube
    - 200 transistors are about the same size as one vacuum tube in a computer
    - Much less expensive than a vacuum tube
    - A transistor can work 40 times faster than a vacuum tube
    - Do not get hot and burn out like a vacuum tube

2.2.2.6 Third Generation - Integrated Circuits

The third generation brought huge gains in computational power. Innovations in this era include the use of integrated circuits, or ICs (semiconductor devices with several transistors built into one physical component), semiconductor memories starting to be used instead of magnetic cores, microprogramming as a technique for efficiently designing complex processors, the coming of age of pipelining and other forms of parallel processing (described in detail in Chapter CA), and the introduction of operating systems and time-sharing.

The first ICs were based on small-scale integration (SSI) circuits, which had around 10 devices per circuit (or ``chip''), and evolved to the use of medium-scale integrated (MSI) circuits, which had up to 100 devices per chip. Multilayered printed circuits were developed and core memory was replaced by faster, solid state memories. Computer designers began to take advantage of parallelism by using multiple functional units, overlapping CPU and I/O operations, and pipelining (internal parallelism) in both the instruction stream and the data stream. In 1964, Seymour Cray developed the CDC 6600, which was the first architecture to use functional parallelism. By using 10 separate functional units that could operate simultaneously and 32 independent memory banks, the CDC 6600 was able to attain a computation rate of 1 million floating point operations per second (1 Mflops).

Five years later CDC released the 7600, also developed by Seymour Cray. The CDC 7600, with its pipelined functional units, is considered to be the first vector processor and was capable of executing at 10 Mflops. The IBM 360/91, released during the same period, was roughly twice as fast as the CDC 660. It employed instruction look ahead, separate floating point and integer functional units and pipelined instruction stream. The IBM 360--195 was comparable to the CDC 7600, deriving much of its performance from a very fast cache memory. The SOLOMON computer, developed by Westinghouse Corporation, and the ILLIAC IV, jointly developed by Burroughs, the Department of Defense and the University of Illinois, were representative of the first parallel computers. The Texas Instrument Advanced Scientific Computer (TI--ASC) and the STAR--100 of CDC were pipelined vector processors that demonstrated the viability of that design and set the standards for subsequent vector processors.

Early in the this third generation Cambridge and the University of London cooperated in the development of CPL (Combined Programming Language, 1963). CPL was, according to its authors, an attempt to capture only the important features of the complicated and sophisticated ALGOL. However, like ALGOL, CPL was large with many features that were hard to learn. In an attempt at further simplification, Martin Richards of Cambridge developed a subset of CPL called BCPL (Basic Computer Programming Language, 1967). In 1970 Ken Thompson of Bell Labs developed yet another simplification of CPL called simply B, in connection with an early implementation of the UNIX operating system. comment):

General Features of The Third Generation

1. From 1965 to around 1972
2. Used integrated circuits - many transistors on one piece of silicon
3. Computers become smaller, faster, more reliable, and lower in price
    - Size of a stove or refrigerator, some can fit on desktops
    - Can do 100,000 to 400,000,000 operations per second
    - Cost about one-tenth the amount of second generation computers
4. Computers become very common in medium to large businesses
5. The concept of the IC was developed by Jack St. Clair Kilby in 1958

6. First IC was invented in 1961 separately by Jack Kilby and Robert Noyce
    - Picture of the first IC
    - IC were incorporated from 1961 in computers
7. An IC is called a silicon chip
8. An IC was about 1/4 inch square and can contain thousands of transistors

The Space Race

9. The Space Race that started in the late 1950's between the United States and the former Soviet Union help to lead to the development of third generation computers
10. Needed a computer small enough to fit in a space capsule

Minicomputer Invented

11. Much smaller and lower in price than previous computers
12. Was really the first general purpose computers used by many businesses.

2.2.2.7 Fourth Generation - Microprocessors

The next generation of computer systems saw the use of large scale integration (LSI -- 1000 devices per chip) and very large scale integration (VLSI -- 100,000 devices per chip) in the construction of computing elements. At this scale entire processors will fit onto a single chip, and for simple systems the entire computer (processor, main memory, and I/O controllers) can fit on one chip. Gate delays dropped to about 1ns per gate.

Semiconductor memories replaced core memories as the main memory in most systems; until this time the use of semiconductor memory in most systems was limited to registers and cache. During this period, high speed vector processors, such as the CRAY 1, CRAY X--MP and CYBER 205 dominated the high performance computing scene. Computers with large main memory, such as the CRAY 2, began to emerge. A variety of parallel architectures began to appear; however, during this period the parallel computing efforts were of a mostly experimental nature and most computational science was carried out on vector processors. Microcomputers and workstations were introduced and saw wide use as alternatives to time--shared mainframe computers.

Developments in software include very high level languages such as FP (functional programming) and Prolog (programming in logic). These languages tend to use a *declarative* programming style as opposed to the *imperative* style of Pascal, C, FORTRAN, et al. In a declarative style, a programmer gives a mathematical specification of what should be computed, leaving many details of how it should be computed to the compiler and/or runtime system. These languages are not yet in wide use, but are very promising as notations for programs that will run on massively parallel computers (systems with over

1,000 processors). Compilers for established languages started to use sophisticated optimization techniques to improve code, and compilers for vector processors were able to vectorize simple loops (turn loops into single instructions that would initiate an operation over an entire vector).

Two important events marked the early part of the third generation: the development of the C programming language and the UNIX operating system, both at Bell Labs. In 1972, Dennis Ritchie, seeking to meet the design goals of CPL and generalize Thompson's B, developed the C language. Thompson and Ritchie then used C to write a version of UNIX for the DEC PDP--11. This C--based UNIX was soon ported to many different computers, relieving users from having to learn a new operating system each time they change computer hardware. UNIX or a derivative of UNIX is now a de facto standard on virtually every computer system.

An important event in the development of computational science was the publication of the Lax report. In 1982, the US Department of Defense (DOD) and National Science Foundation (NSF) sponsored a panel on Large Scale Computing in Science and Engineering, chaired by Peter D. Lax. The Lax Report stated that aggressive and focused foreign initiatives in high performance computing, especially in Japan, were in sharp contrast to the absence of coordinated national attention in the United States. The report noted that university researchers had inadequate access to high performance computers. One of the first and most visible of the responses to the Lax report was the establishment of the NSF supercomputing centers. Phase I on this NSF program was designed to encourage the use of high performance computing at American universities by making cycles and training on three (and later six) existing supercomputers immediately available. Following this Phase I stage, in 1984--1985 NSF provided funding for the establishment of five Phase II supercomputing centers.

The Phase II centers, located in San Diego (San Diego Supercomputing Center); Illinois (National Center for Supercomputing Applications); Pittsburgh (Pittsburgh Supercomputing Center); Cornell (Cornell Theory Center); and Princeton (John von Neumann Center), have been extremely successful at providing computing time on supercomputers to the academic community. In addition they have provided many valuable training programs and have developed several software packages that are available free of charge. These Phase II centers continue to augment the substantial high performance computing efforts at the National Laboratories, especially the Department of Energy (DOE) and NASA sites.

General Features of Fourth Generation Computers

1. Form 1972 until now
2. Used large scale to very large scale integrated circuits
    - Put more than one IC on a silicon chip
    - Can do more than one function
3. Computers become smaller, faster, more reliable, and lower in price

- Size of a television or much smaller
- Can do 500,000 to 1,000,000,000 operations per second
- Cost one-tenth, or less, the amount of third generation computers

4. Computers become very common in homes and business

The Microprocessor

5. The microprocessor is a complete computer on a chip
    - Can do all the functions of a computer - input, process, and output data
    - The first microprocessor was produced by Ted Hoff for Intel in 1971 - the Intel 4004
6. Modern microprocessors are usually less than one inch square and can contain million of electronic circuits (picture of a Pentium II circuits)
7. Used in many electronic devices today, from wrist watches to microwave ovens to cars

The Microcomputer is Invented

8. A microcomputer is any general purpose computer that uses a microprocessor for a CPU
9. In 1972 the first microcomputer was introduced - the MITS 816 - with no keyboard or display.
10. In 1976, the first real assembled and complete computer was produced - the Apple II
    - Used by schools and colleges
    - Apple Corporation founded at this time by Steve Jobs and Steve Wozniak

The Personal Computer

11. The PC was introduced by IBM in 1981
12. Use the DOS operating system developed by Microsoft Corporation
13. Changed the public's view of computers
    - Wanted computers that could do useful tasks
    - Wanted a computer that was easy to use
    - Could do work at home that would be transferable to the company's computer

The Macintosh Computer

14. Introduced in 1984 by the Apple Corporation

First home and small business computer to use a Graphic User Interface (GUI)

- Used a mouse as a pointing device
- Used icons (small pictures) to represent disks, files, and programs
- Based on ideas from Xerox PARC Alto system

15. With the invention of the Laser Printer by Apple a year later, desktop publishing took off

The Internet

16. The Internet was started in 1969 as ARPAnet by the US military to connect research facilities together
17. The Internet went public in 1991
    - Connects computers together by using phone lines and other networks
    - Allows for the rapid sharing of information and resources
18. Because of small powerful computers, the Internet is rapidly changing our society

2.2.2.8 Fifth Generation Computers

The development of the next generation of computer systems is characterized mainly by the acceptance of parallel processing. Until this time parallelism was limited to pipelining and vector processing, or at most to a few processors sharing jobs. The fifth generation saw the introduction of machines with hundreds of processors that could all be working on different parts of a single program. The scale of integration in semiconductors continued at an incredible pace --- by 1990 it was possible to build chips with a million components --- and semiconductor memories became standard on all computers.

Other new developments were the widespread use of computer networks and the increasing use of single-user workstations. Prior to 1985 large scale parallel processing was viewed as a research goal, but two systems introduced around this time are typical of the first commercial products to be based on parallel processing. The Sequent Balance 8000 connected up to 20 processors to a single shared memory module (but each processor had its own local cache). The machine was designed to compete with the DEC VAX--780 as a general purpose Unix system, with each processor working on a different user's job. However Sequent provided a library of subroutines that would allow programmers to write programs that would use more than one processor, and the machine was widely used to explore parallel algorithms and programming techniques.

The Intel iPSC--1, nicknamed ``the hypercube'', took a different approach. Instead of using one memory module, Intel connected each processor to its own memory and used a network interface to connect processors. This *distributed memory* architecture meant memory was no longer a bottleneck and large systems (using more processors) could be

built. The largest iPSC--1 had 128 processors. Toward the end of this period a third type of parallel processor was introduced to the market. In this style of machine, known as a *data-parallel* or SIMD, there are several thousand very simple processors. All processors work under the direction of a single control unit; i.e. if the control unit says ``add a to b'' then all processors find their local copy of a and add it to their local copy of b. Machines in this class include the Connection Machine from Thinking Machines, Inc., and the MP--1 from MasPar, Inc.

Scientific computing in this period was still dominated by vector processing. Most manufacturers of vector processors introduced parallel models, but there were very few (two to eight) processors in this parallel machines. In the area of computer networking, both wide area network (WAN) and local area network (LAN) technology developed at a rapid pace, stimulating a transition from the traditional mainframe computing environment toward a distributed computing environment in which each user has their own workstation for relatively simple tasks (editing and compiling programs, reading mail) but sharing large, expensive resources such as file servers and supercomputers. RISC technology (a style of internal organization of the CPU) and plummeting costs for RAM brought tremendous gains in computational power of relatively low cost workstations and servers. This period also saw a marked increase in both the quality and quantity of scientific visualization.

2.2.2.9 Sixth Generation of Computers

Combinations of parallel/vector architectures are well established, and one corporation (Fujitsu) has announced plans to build a system with over 200 of its high end vector processors. Manufacturers have set themselves the goal of achieving teraflops arithmetic operations per second) performance by the middle of the decade, and it is clear this will be obtained only by a system with a thousand processors or more. Workstation technology has continued to improve, with processor designs now using a combination of RISC, pipelining, and parallel processing. As a result it is now possible to purchase a desktop workstation for about $30,000 that has the same overall computing power (100 megaflops) as fourth generation supercomputers. This development has sparked an interest in heterogeneous computing: a program started on one workstation can find idle workstations elsewhere in the local network to run parallel subtasks.

One of the most dramatic changes in the sixth generation will be the explosive growth of wide area networking. Network bandwidth has expanded tremendously in the last few years and will continue to improve for the next several years. T1 transmission rates are now standard for regional networks, and the national ``backbone'' that interconnects regional networks uses T3. Networking technology is becoming more widespread than its original strong base in universities and government laboratories as it is rapidly finding application in K--12 education, community networks and private industry. A little over a decade after the warning voiced in the Lax report, the future of a strong computational science infrastructure is bright. The federal commitment to high performance computing

has been further strengthened with the passage of two particularly significant pieces of legislation: the High Performance Computing Act of 1991, which established the High Performance Computing and Communication Program (HPCCP) and Sen. Gore's Information Infrastructure and Technology Act of 1992, which addresses a broad spectrum of issues ranging from high performance computing to expanded network access and the necessity to make leading edge technologies available to educators from kindergarten through graduate school.

In bringing this encapsulated survey of the development of a computational science infrastructure up to date, we observe that the President's FY 1993 budget contains $2.1 billion for mathematics, science, technology and science literacy educational programs, a 43% increase over FY 90 figures.

2.2.2.10 Future Computer Generations

Hard to Predict

1. Most inventions or technologies that have changed computers are not usually predicted to far in advance of when they are first used
2. Most likely the following will happen to computer technology
   - It will become lower in price
   - Computers will become smaller and faster
   - Computers will have larger memories and more storage space
3. Computers will become an integral part of everyone's life

People Will Become More Interconnected

1. Computer technology and the World Wide Web will greatly reduce the distance between people and cultures in the world
2. People will connect to information at any place or time
   - Libraries and other information sources will always be open
   - Cell phone technology will let you connect to information and people any where

Computers Will Become Small Enough to Wear

1. The technology is already being developed
   - Eyeglasses with a display
   - Research is being done to find the best place to put computers so they will not interfere with the body's movement
2. The computer will always be with you to help you in tasks, communicate, and find information

Summary

The evolution of digital computing is often divided into *generations*. Each generation is characterized by dramatic improvements over the previous generation in the technology used to build computers, the internal organization of computer systems, and programming languages. Although not usually associated with computer generations, there has been a steady improvement in algorithms, including algorithms used in computational science. The first multi-purpose, i.e. *programmable*, computing device was probably Charles Babbage's Difference Engine. Three machines have been promoted at various times as the first electronic computers.

The First Generation was from 1946 to 1956. All computers of this generation used vacuum tubes to perform calculations. Second generation of computers are Smaller, faster, and more reliable than the First Generation of computers. Third generation was from 1965 to around 1972. They Used integrated circuits - many transistors on one piece of silicon and microcomputer was also invented in the this generation of computers. Fourth generation used microprocessor is a complete computer on a chip and apple incorporation introduced Macintosh systems and the internet also came into existence. The First Generation was from 1946 to 1956. The generations from then took a faster leap and are now in the faster and never dreamt of state with the fastest communication and greater and powerful processing abilities.

Model Questions

1. What is meant by generation of computers? How many generations of computers are there so far?
2. How is the evolution and development of computers explained in various generations of computers
3. Explain improvement in computers the fourth, fifth and sixth generation of computers?
4. What would be the future predicted generation of computers?

References

1. Fifth Generation Management: Dynamic Teaming, Virtual Enterprising and Knowledge Networking by Charles M Savage.
2. Fundamentals of Computer Science Using Java by David Hughes.
3. http://csep1.phy.ornl.gov/ov/node8.html

Author:-
Asha Smitha. B.
Center For Biotechnology

Lesson 2.2.3

OPERATING SYSTEM

Contents

2.2.3.1 Objective
2.2.3.2 Introduction
2.2.3.3 Origin
2.2.3.4 Operating System
2.2.3.5 Classification
2.2.3.6 Allocating Main Memory
2.2.3.7 Processes
2.2.3.8 Summary
2.2.3.9 Model Questions
2.2.3.10 References

2.2.3.1 Objective

- To understand what an operating system is
- To explain the concepts of operating systems like processes, memory management etc.

2.2.3.2 Introduction

An operating system is an important part of almost every computer system. A computer system can be divided roughly into four components: the hardware, the operating system, the applications programs, and the users.

The software that the rest of the software depends on to make the computer functional. On most PCs this is Windows or the Macintosh OS. Unix and Linux are other operating systems often found in scientific and technical environments.

Early computers lacked any form of operating system. The user had sole use of the machine; he would arrive at the machine armed with his program and data, often on punched paper tape. The program would be loaded into the machine, and the machine set to work, until the program stopped, or maybe more likely, crashed. Programs could generally be debugged via a front panel using switches and lights; it is said that Alan Turing was a master of this on the early Manchester Mark I machine.

Later, machines came with libraries of support code which were linked to the user's program to assist in operations such as input and output. This would become the genesis of the modern-day operating system. However, machines still ran a single job at a time; at Cambridge University in England the job queue was at one time a washing line from which tapes were hung with clothes pegs. The color of the pegs indicated the priority of the job.

As machines became more powerful, the time needed for a run of a program diminished and the time to hand off the equipment became very large by comparison. Accounting for and paying for machine usage went from checking the wall clock to using the computer to do the timing. Run queues went from being people waiting at the door to stacks of media waiting on a table to using the hardware of the machine such as switching which magnetic tape drive was online or stacking punch cards on top of the previous jobs cards in the reader. Operating the computer went from a task performed by the program developer to a job for full time dedicated machine operators. When commercially available computer centers found they had to deal with accidental or malicious tampering of the accounting information, equipment vendors were encouraged to enhance the properties of the runtime libraries to prevent misuse of the systems resources. Accounting practices were also expanded beyond recording CPU usage to also count pages printed, cards punched, cards read, disk storage used, and even operator action required by jobs such as changing magnetic tapes. Eventually, the runtime libraries became a program that was started before the first customer job, that read in the customer job, controlled its execution, cleaned up after it, recorded its usage, and immediately went on to process the next job. Jobs also evolved from being binary images produced by hand encoding to symbolic programs that were translated by the computer. An operating system, or "monitor" as it was sometimes called, permitted jobs to become multistep with the monitor running several programs in sequence to effect the translation and subsequent run of the user's program.

The conceptual bridge between the precise description of an operating system and the colloquial definition is the tendency to bundle widely, or generally, used utilities and applications (such as text editors or file managers) with the basic OS for the sake of convenience; as OSes progressed, a larger selection of 'second class' OS software came to be included, such that now, an OS without a graphical user interface or various file viewers is often considered not to be a true or complete OS. To accommodate this evolution of the meaning most of what was the original "operating system" is now called the "kernel", and OS has come to mean the complete package.

The broader categories of systems and application software are discussed in the computer software article.

2.2.3.3 Origin

The first computers were built for military purposes during World War II, and the first commercial computers were built during the 50's. They were huge (often filling a large room with tons of equipment), expensive (millions of dollars, back when that was a lot of money), unreliable, and slow (about the power of today's $1.98 pocket calculator). Originally, there was no distinction between programmer, operator, and end-user (the person who wants something done). A physicist who wanted to calculate the trajectory of a missile would sign up for an hour on the computer. When his time came, he would come

into the room, feed in his program from punched cards or paper tape, watch the lights flash, maybe do a little debugging, get a print-out, and leave.

The first card in the deck was a *bootstrap loader*. The user/operator/programmer would push a button that caused the card reader to read that card, load its contents into the first 80 locations in memory, and jump to the start of memory, executing the instructions on that card. Those instructions read in the rest of the cards, which contained the instructions to perform all the calculations desired: what we would now call the "application program".

This set-up was a lousy way to debug a program, but more importantly, it was a waste of the fabulously expensive computer's time. Then someone came up with the idea of *batch processing*. User/programmers would punch their jobs on decks of cards, which they would submit to a professional operator. The operator would combine the decks into batches. He would precede the batch with a *batch executive* (another deck of cards). This program would read the remaining programs into memory, one at a time, and run them. The operator would take the printout from the printer, tear off the part associated with each job, wrap it around the associated deck, and put it in an output bin for the user to pick up. The main benefit of this approach was that it minimized the wasteful down time between jobs. However, it did not solve the growing I/O bottleneck.

Card readers and printers got faster, but since they are mechanical devices, there were limits to how fast they could go. Meanwhile the central processing unit (CPU) kept getting faster and was spending more and more time idly waiting for the next card to be read in or the next line of output to be printed. The next advance was to replace the card reader and printer with magnetic tape drives, which were much faster. A separate, smaller, slower (and presumably cheaper) *peripheral* computer would copy batches of input decks onto tape and transcribe output tapes to print. The situation was better, but there were still problems. Even magnetic tapes drives were not fast enough to keep the mainframe CPU busy, and the peripheral computers, while cheaper than the mainframe, were still not cheap (perhaps hundreds of thousands of dollars).

Then the card reader and printer were hooked up to the mainframe (along with the tape drives) and the mainframe CPU was reprogrammed to switch rapidly among several tasks. First it would tell the card reader to start reading the next card of the next input batch. While it was waiting for that operation to finish, it would go and work for a while on another job that had been read into "core" (main memory) earlier. When enough time had gone by for that card be read in, the CPU would temporarily set aside the main computation, start transferring the data from that card to one of the tape units (say tape 1), start the card reader reading the next card, and return to the main computation. It would continue this way, servicing the card reader and tape drive when they needed attention and spending the rest of its time on the main computation. Whenever it finished working on one job in the main computation, the CPU would read another job from an input tape that had been prepared earlier (tape 2). When it finished reading in and

executing all the jobs from tape 2, it would swap tapes 1 and 2. It would then start executing the jobs from tape 1, while the input "process" was filling up tape 2 with more jobs from the card reader. Of course, while all this was going on, a similar process was copying output from yet another tape to the printer. This amazing juggling act was called Simultaneous Peripheral Operations On Line, or SPOOL for short.

The hardware that enabled SPOOLing is called *direct memory access*, or DMA. It allows the card reader to copy data directly from cards to core and the tape drive to copy data from core to tape, while the expensive CPU was doing something else. The software that enabled SPOOLing is called *multiprogramming*. The CPU switches from one activity, or "process" to another so quickly that it appears to be doing several things at once.

In the 1960's, multiprogramming was extended to ever more ambitious forms. Hardware developments supporting this extension included decreasing cost of core memory (replaced during this period by semi-conductor random-access memory (RAM)), and introduction of direct-access storage devices (called DASD - pronounced "dazdy" - by IBM and "disks" by everyone else). With larger main memory, multiple jobs could be kept in core at once, and with input spooled to disk rather than tape, each job could get directly at its part of the input. With more jobs in memory at once, it became less likely that they would all be simultaneously blocked waiting for I/O, leaving the expensive CPU idle.

Another break-through idea from the 60's based on multiprogramming was *timesharing*, which involves running multiple *interactive* jobs, switching the CPU rapidly among them so that each interactive user feels as if he has the whole computer to himself. Timesharing let the programmer back into the computer room - or at least a *virtual* computer room. It allowed the development of interactive programming, making programmers much more productive. Perhaps more importantly, it supported new applications such as airline reservation and banking systems that allowed 100s or even 1000s of agents or tellers to access the same computer "simultaneously". Visionaries talked about an "computing utility" by analogy with the water and electric utilities, which would delivered low-cost computing power to the masses.

Today, computers are used for a wide range of applications, including personal interactive use (word-processing, games, desktop publishing, web browsing, email), real-time systems (patient care, factories, missiles), embedded systems (cash registers, wrist watches, toasters), and transaction processing (banking, reservations, e-commerce).

The goal of an OS is to make hardware look better than it is.

- More regular, uniform (instead of lots of idiosyncratic devices)
- Easier to program (e.g., don't have to worry about speeds, asynchronous events)

Closer to what's needed for applications:

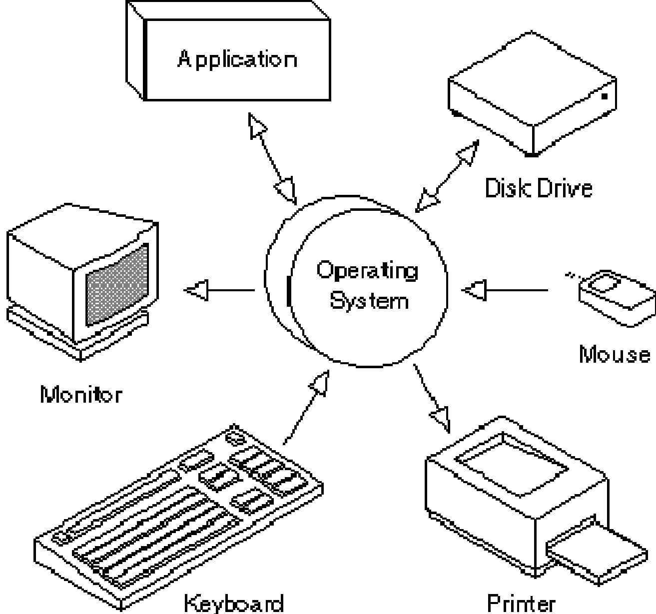
- named, variable-length files, rather than disk blocks
- multiple "CPU's", one for each user (in shared system) or activity (in single-user system)

- multiple large, dynamically-growing memories (virtual memory)

2.2.3.4 Operating System

An operating system is a layer of software which takes care of technical aspects of a computer's operation. It shields the user of the machine from the low-level details of the machine's operation and provides frequently needed facilities. There is no universal definition of what an operating system consists of. You can think of it as being the software which is already installed on a machine, before you add anything of your own.

Normally the operating system has a number of key elements: (i) a *technical layer of software* for driving the hardware of the computer, like disk drives, the keyboard and the screen; (ii) a *filesystem* which provides a way of organizing files logically, and (iii) a simple *command language* which enables users to run their own programs and to manipulate their files in a simple way. Some operating systems also provide text editors, compilers, debuggers and a variety of other tools. Since the operating system (OS) is in charge of a computer, all requests to use its resources and devices need to go through the OS. An OS therefore provides (iv) *legal entry points* into its code for performing basic operations like writing to devices.



2.2.3.5 Classification

Operating systems can be classified as follows:

Operating systems may be classified by both how many tasks they can perform `simultaneously' and by how many users can be using the system `simultaneously'. That is: *single-user* or *multi-user* and *single-task* or *multi-tasking*.

- **multi-user :** Allows two or more users to run programs at the same time. Some operating systems permit hundreds or even thousands of concurrent users.
- **multiprocessing :** Supports running a program on more than one CPU.
- **multitasking :** Allows more than one program to run concurrently.
- **multithreading :** Allows different parts of a single program to run concurrently.
- **real time:** Responds to input instantly. General-purpose operating systems, such as DOS and UNIX, are not real-time.

Operating systems provide a software platform on top of which other programs, called *application programs,* can run. The application programs must be written to run on top of a particular operating system. Your choice of operating system, therefore, determines to a great extent the applications you can run. For PCs, the most popular operating systems are DOS, OS/2, and Windows, but others are available, such as Linux.

As a user, you normally interact with the operating system through a set of commands. For example, the DOS operating system contains commands such as COPY and RENAME for copying files and changing the names of files, respectively. The commands are accepted and executed by a part of the operating system called the command processor or command line interpreter. Graphical user interfaces allow you to enter commands by pointing and clicking at objects that appear on the screen.

2.2.3.6 Allocating Main Memory

Let us first consider how to manage main (``core'') memory (also called *random-access memory* (RAM)). In general, a memory manager provides two operations:

```
Address allocate(int size);
void deallocate(Address block);
```

The procedure allocate receives a request for a contiguous block of size bytes of memory and returns a pointer to such a block. The procedure deallocate releases the indicated block, returning it to the *free pool* for reuse. Sometimes a third procedure is also provided,

```
Address reallocate(Address block, int new_size);
```

which takes an allocated block and changes its size, either returning part of it to the free pool or extending it to a larger block. It may not always be possible to grow the block without copying it to a new location, so reallocate returns the new address of the block.

Memory allocators are used in a variety of situations. In Unix, each process has a *data segment*. There is a system call to make the data segment bigger, but no system call to make it smaller. Also, the system call is quite expensive. Therefore, there are library

procedures (called malloc, free, and realloc) to manage this space. Only when malloc or realloc runs out of space is it necessary to make the system call. The operating system also uses a memory allocator to manage space used for OS data structures and given to ``user'' processes for their own use. As we saw before, there are several reasons why we might want multiple processes, such as serving multiple interactive users or controlling multiple devices.

Suppose there are n processes in memory (this is called the *level of multiprogramming*) and each process is blocked (waiting for I/O) a fraction p of the time. In the best case, when they ``take turns'' being blocked, the CPU will be 100% busy provided $n(1-p) >= 1$. For example, if each process is ready 20% of the time, $p = 0.8$ and the CPU could be kept completely busy with five processes. Of course, real processes aren't so cooperative. In the worst case, they could all decide to block at the same time, in which case, the CPU *utilization* (fraction of the time the CPU is busy) would be only $1 - p$ (20% in our example). If each processes decides randomly and independently when to block, the chance that all n processes are blocked at the same time is only p^n, so CPU utilization is $1 - p^n$. Continuing our example in which $n = 5$ and $p = 0.8$, the expected utilization would be $1 - .8^5 = 1 - .32768 = 0.67232$. In other words, the CPU would be busy about 67% of the time on the average.

How does the memory manager know how big the returned block is? The usual trick is to put a small *header* in the allocated block, containing the size of the block and perhaps some other information. The allocate routine returns a pointer to the body of the block, not the header, so the client doesn't need to know about it. The deallocate routine subtracts the header size from its argument to get the address of the header. The client thinks the block is a little smaller than it really is. So long as the client ``colors inside the lines'' there is no problem, but if the client has bugs and scribbles on the header, the memory manager can get completely confused. To make it easier to coalesce adjacent holes, the memory manager also adds a flag (called a ``boundary tag'') to the beginning and end of each hole or allocated block, and it records the size of a hole at *both* ends of the hole.

When the block is deallocated, the memory manager adds the size of the block (which is stored in its header) to the address of the beginning of the block to find the address of the first word following the block. It looks at the tag there to see if the following space is a hole or another allocated block. If it is a hole, it is removed from the free list and merged with the block being freed, to make a bigger hole. Similarly, if the boundary tag preceding the block being freed indicates that the preceding space is a hole, we can find the start of that hole by subtracting its size from the address of the block being freed (that's why the size is stored at both ends), remove it from the free list, and merge it with the block being freed. Finally, we add the new hole back to the free list. Holes are kept in a doubly-linked list to make it easy to remove holes from the list when they are being coalesced with blocks being freed.

Compaction and Garbage Collection

What do you do when you run out of memory? Any of these methods can fail because all the memory is allocated, or because there is too much fragmentation. Malloc, which is being used to allocate the data segment of a Unix process, just gives up and calls the (expensive) OS call to expand the data segment. A memory manager allocating real physical memory doesn't have that luxury. The allocation attempt simply fails. There are two ways of delaying this catastrophe, compaction and garbage collection.

Compaction attacks the problem of fragmentation by moving all the allocated blocks to one end of memory, thus combining all the holes. Aside from the obvious cost of all that copying, there is an important limitation to compaction: Any pointers to a block need to be updated when the block is moved. Unless it is possible to find all such pointers, compaction is not possible. Pointers can stored in the allocated blocks themselves as well as other places in the client of the memory manager. In some situations, pointers can point not only to the start of blocks but also into their bodies. For example, if a block contains executable code, a branch instruction might be a pointer to another location in the same block. Compaction is performed in three phases. First, the new location of each block is calculated to determine the distance the block will be moved. Then each pointer is updated by adding to it the amount that the block it is pointing (in)to will be moved. Finally, the data is actually moved. There are various clever tricks possible to combine these operations.

Garbage collection finds blocks of memory that are inaccessible and returns them to the free list. As with compaction, garbage collection normally assumes we find all pointers to blocks, both within the blocks themselves and ``from the outside.'' If that is not possible, we can still do ``conservative'' garbage collection in which every word in memory that contains a value that appears to be a pointer is treated as a pointer. The conservative approach may fail to collect blocks that are garbage, but it will never mistakenly collect accessible blocks. There are three main approaches to garbage collection: reference counting, mark-and-sweep, and generational algorithms.

Reference counting keeps in each block a count of the number of pointers to the block. When the count drops to zero, the block may be freed. This approach is only practical in situations where there is some ``higher level'' software to keep track of the counts (it's much too hard to do by hand), and even then, it will not detect cyclic structures of garbage: Consider a cycle of blocks, each of which is only pointed to by its predecessor in the cycle. Each block has a reference count of 1, but the entire cycle is garbage. Mark-and-sweep works in two passes: First we mark all non-garbage blocks by doing a depth-first search starting with each pointer ``from outside'':

There are two problems with mark-and-sweep. First, the amount of work in the mark pass is proportional to the amount of *non*-garbage. Thus if memory is nearly full, it

will do a lot of work with very little payoff. Second, the mark phase does a lot of jumping around in memory, which is bad for virtual memory systems, as we will soon see.

The third approach to garbage collection is called *generational* collection. Memory is divided into *spaces*. When a space is chosen for garbage collection, all subsequent references to objects in that space cause the object to be copied to a new space. After a while, the old space either becomes empty and can be returned to the free list all at once, or at least it becomes so sparse that a mark-and-sweep garbage collection on it will be cheap. As an empirical fact, objects tend to be either short-lived or long-lived. In other words, an object that has survived for a while is likely to live a lot longer. By carefully choosing where to move objects when they are referenced, we can arrange to have some spaces filled only with long-lived objects, which are very unlikely to become garbage. We garbage-collect these spaces seldom if ever.

2.2.3.7 Processes

A process is a ``little bug'' that crawls around on the program executing the instructions it sees there. Normally (in so-called *sequential* programs) there is exactly one process per program, but in *concurrent* programs, there may be several processes executing the same program. The details of what constitutes a ``process'' differ from system to system. The main difference is the amount of *private state* associated with each process. Each process has its own *program counter*, the register that tells it where it is in the program. It also needs a place to store the return address when it calls a subroutine, so that two processes executing the same subroutine called from different places can return to the correct calling points. Since subroutines can call other subroutines, each process needs its own *stack* of return addresses.

Processes with very little private memory are called *threads* or *light-weight processes*. At a minimum, each thread needs a program counter and a place to store a stack of return addresses; all other values could be stored in memory shared by all threads. At the other extreme, each process could have its own private memory space, sharing only the read-only program text with other processes. This essentially the way a Unix process works. Other points along the spectrum are possible. One common approach is to put the local variables of procedures on the same private stack as the return addresses, but let all global variables be shared between processes. A stack *frame* holds all the local variables of a procedure, together with an indication of where to return to when the procedure returns, and an indication of where the calling procedure's stack frame is stored. This is the approach taken by Java threads. Java has no global variables, but threads all share the same *heap*. The heap is the region of memory used to hold objects allocated by new. In short, variables declared in procedures are local to threads, but objects are all shared. Of course, a thread can only ``see'' an object if it can reach that object from its ``base'' object (the one containing its run method, or from one of its local variables.

```
class Worker implements Runnable {
   Object arg, other;
   Worker(Object a) { arg = a; }
   public void run() {
      Object tmp = new Object();
      other = new Object();
      for(int i = 0; i < 1000; i++) // do something
   }
}
class Demo {
   static public void main(String args[]) {
      Object shared = new Object();

      Runnable worker1 = new Worker(shared);
      Thread t1 = new Thread(worker1);

      Runnable worker2 = new Worker(shared);
      Thread t2 = new Thread(worker2);

      t1.start(); t2.start();
      // do something here
   }
}
```

There are three treads in this program, the main thread and two child threads created by it. Each child thread has its own stack frame for Worker.run(), with space for tmp and i. Thus there are two copies of the variable tmp, each of which points to a different instance of Object. Those objects are in the shared heap, but since one thread has no way of getting to the object created by the other thread, these objects are effectively ``private'' to the two threads.[1] Similarly, the objects pointed to by other are effectively private. But both copies of the field arg and the variable shared in the main thread all point to the same (shared) object.
Other names sometimes used for processes are *job* or *task*.

Why Use Processes

Processes are basically just a programming convenience, but in some settings they are such a great convenience, it would be nearly impossible to write the program without them. A process allows you to write a single thread of code to get some task done, without worrying about the possibility that it may have to wait for something to happen along the way. Examples:
A server providing services to others.
One thread for each client.
A timesharing system.
One thread for each logged-in user.
A real-time control computer controlling a factory.

One thread for each device that needs monitoring.

A network server.

One thread for each connection.

Resource allocation: A ``resource'' can be defined as something that costs money. The philosophers represent processes, and the forks represent resources.
There are three kinds of resources:

- sharable
- serially reusable
- consumable

Sharable resources can be used by more than one process at a time. A consumable resource can only be used by one process, and the resource gets ``used up.'' A serially reusable resource is in between. Only process can use the resource at a time, but once it's done, it can give it back for use by another process. Examples are the CPU and memory. These are the most interesting type of resource. A process requests a (serially reusable) resource from the OS and holds it until it's done with it; then it releases the resource. The OS may delay responding to a request for a resource. The requesting process is blocked until the OS responds. A set of processes is *deadlocked* if each process in the set is waiting for an event that only a process in the set can cause.

2.2.3.8 Summary

An operating system is an important part of almost every computer system. A computer system can be divided roughly into four components: the hardware, the operating system, the applications programs, and the users. The procedure allocate receives a request for a contiguous block of size bytes of memory and returns a pointer to such a block. The procedure deallocate releases the indicated block, returning it to the *free pool* for reuse. A process is a ``little bug'' that crawls around on the program executing the instructions it sees there.

2.2.3.9 Model Questions

1. What is an operating system? How is it classified?
2. How did the operating system originate?
3. what is a process? Why are they used?
4. Explain the classification of an operating system?

2.2.3.10 References

1. Operatins systems Concepts – Silberschatz Galvin
2. Computers and Commonsense – Roger Hunt, John Shilly
3. Operating Systems (3rd Edition) -- Harvey M. Deitel, et al

Lesson 2.2.4

INTERNAL AND EXTERNAL COORDINATE SYSTEM

2.2.4.1 Objective
2.2.4.2 Introduction
2.2.4.3 In Brief
2.2.4.4 Internal Coordinate Mechanics
2.2.4.5 Applications of ICM
2.2.4.6 Modules of ICM
Summary
Model Questions
References

2.2.4.1 Objective

To get a knowledge of internal coordinate system and its applications in biological research.

2.2.4.1 Introduction

Internal coordinates define the distances, angles and torsional angles between the atoms of a molecule. They allow the precise description of a molecular structure without an external coordinate system. If the starting point is chosen well enough, even complicated structures can be built easily just by following the bonding topology, without the need for graphical structure editors or crystal structure data. The resulting dataset is particularly easy to modify and extend, as the placement of substituents is a much more straightforward task than in cartesian space. The closing of ring structures, however, may prove adventurous to the beginner.

In the past ten years efficient algorithms have made computationally tractable the use of internal coordinates in molecular dynamics simulations of systems of biological interest (having more than say 100 atoms). Internal coordinates are an attractive alternative to the Cartesian coordinates of each atom when particular degrees of freedom are not of interest. For example, in the process of NMR structure determination and refinement in which one seeks molecular structures consistent with experimental

NMR data, the bond lengths and bond angles are generally taken as fixed—and no information about these features is generally available from the NMR experiments. If these known coordinates are removed from the local optimizations and molecular dynamics simulations, the conformational search space becomes smaller and can be more rapidly sampled. For example, typical proteins have approximately Na=3 torsion angles compared with $3Na$ coordinates in atomic Cartesian space, where Na is the number of atoms. Hence, the conformational space is about an order of magnitude smaller if torsion angles are used. Furthermore, in torsional angle molecular dynamics

(TAMD), it is typical that the timestep required to maintain a given level of energy conservation is about 10 times larger than that required in atomic Cartesian space because the high frequency bond bending and stretching motions have been removed. Other aspects of the simulation might also be made more efficient because bond and bond-angle forces no longer need be calculated and because there are fewer coordinates to update in the integrator. However, these final two aspects have not been found to make a significant contribution to dynamics run times in practice.

An efficient recursive algorithm for dynamics in internal coordinateswas originally introduced in the robotics literature (*1–4*). This algorithm was then implemented for TAMD in X-ray and NMR refinement packages (*5–7*) and in a more general purpose molecular dynamics package (*8, 9*). In this paper we report the implementation of a general internal variable dynamics module (IVM) for efficient molecular dynamics. It allows general hinge definitions including those used in TAMD, but it also allows more general coordinates which are appropriate when some degrees of freedom are of interest and others are not; for instance, in the refinement problem of a two protein complex in which the backbone coordinates of the isolated protein structures are already known. The IVM also includes local minimization routines (Powell method conjugate gradient and steepest descent) so that these techniques can be conveniently employed in the same coordinate system. Our package employs an efficient sixth-order predictorcorrector integrator, which requires one force evaluation per timestep and allows for automatic timestep adjustment.We have implemented loop constraints to maintain bond lengths in ring topologies, although as yet we have found the feature to be of limited use. Finally, the code has been developed in a highly modular fashion to make the addition of new hinge definitions, integrators, and minimizers a relatively simple task. The IVM is not a stand-alone program as it does not have code to evaluate forces and lacks support for file formats, etc.

2.2.4.2 In Brief

If internal coordinates are to be kept fixed during a calculation, or if they are to be varied automatically (as in the determination of a rotational barrier), the starting structure must be given as internal coordinates.

This dataset starts with atomic number and type code of the first atom. For the second atom, the distance from the first (in angstrom units, 1 Å=100pm) has to be given as well. This must not necessarily be identical to a chemical bond, as the bond matrix is determined solely on the basis of internuclear distances. However, it will usually be easier to use actual bonds here. Definition of the third atom includes the angle it forms with the second and first atom. Beginning with the fourth atom, the torsional angles, measured clockwise viewing from the second to the first reference atom, have to be given as well. While the definition is unequivocal for the first three atoms, all others require the numbers of the reference atoms to be given. As this list is processed sequentially, only atoms that are already defined to the program may be used as references. This becomes particularly important, if an atom or a group of atoms is to be deleted from the middle of a dataset, e.g. in the removal of a keto group. This will usually change the reference atoms of all following atoms !

Input ends with an atom of atomic number zero, or simply with a blank line.

The following attributes may be used in addition:

The variable **IFIX**, read between the type code and the atomic distance, specifies whether the cartesian coordinates of the atom are to be fixed :

- **IFIX**=0, the atom is not fixed to its position,
- **IFIX**=1, the cartesian coordinates of this atom are fixed,
- **IFIX**=2, the Z coordinate of this atom is fixed (useful for enforcing planarity)

IFIX is followed by the molecule number **IMOL**. As the program does not assume bonds between fragments with different molecule numbers, this option may be useful in the study of molecular complexes. However it should be used only as a last resort, if it is not possible to rewrite the input. While it is easy to suppress unwanted bonding between the two molecules, the short distance between them will give rise to extreme van-der-Waals interaction.

The value of the torsional angle may be followed by the permutation flag:

- **ITOR**=0 no permutation
- **ITOR**=1 assumes angles of 0° and 180°
- **ITOR**=2 selects 60°, 180° and 300°
- **ITOR**=3 selects 0°, 90°, 180° and 270°
- **ITOR**=4 selects 0°, 45°, 90°, 135° and 180°
- **ITOR**=5 selects 0°, 60°, 120°, 180°, 240°, 300° and 360°
- **ITOR**=6 initiates a sequence of seven random values

Up to ten torsional angles may be varied in this way, evaluating all combinations of their values. The value of the restriction flag **NH** in the options line determines whether these values are fixed throughout the calculation, or treated only as starting points in a free optimization.

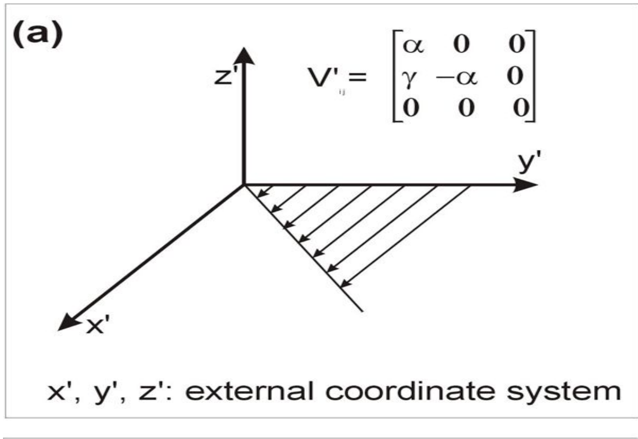
NB: The actual permutation is carried out in a sequential calculation FORTRAN fixed record format: (4I3,3F10.4,I2,8X,3I3)

```
column    1-2   atomic number
column    3-5   atom type code
column    6-8   restraint flag IFIX
column   9-11   molecule number IMOL
column  12-20   distance from first reference atom
column  31-40   angle with second reference atom
column  51-60   torsional angle with third reference atom
column  61-62   permutation flag ITOR
column  71-73   sequence number of first reference atom
column  74-76   sequence number of second reference atom
column  77-79   sequence number of third reference atom
```

Integer numbers should be entered right justified, floating point numbers must contain a decimal point. According to the FORTRAN standard, if a floating point number is read without a decimal point, zeroes will be appended to it up to the end of the input field before the decimal point is added ! This feature will almost certainly lead to unexpected and sometimes spectacular results.

{Example :

```
benzene
 6  3  0  2  2.00  2  0 80  3  5  1 20 50  0
 6  1  0  0  0.          0.          0.    0      0  0  0
 6  1  0  0  1.4         0.          0.    0      0  0  0
 6  1  0  0  1.4       120.          0.    0      0  0  0
 6  1  0  0  1.4       120.          0.    0      3  2  1
 6  1  0  0  1.4       120.          0.    0      4  3  2
 6  1  0  0  1.4       120.          0.    0      5  4  3
 1 19        1.        120.        180.            1  2  3
 1 19        1.        120.        180.            2  3  4
 1 19        1.        120.        180.            3  4  5
 1 19        1.        120.        180.            4  5  6
 1 19        1.        120.        180.            5  6  1
 1 19        1.        120.        180.            6  1  2
```



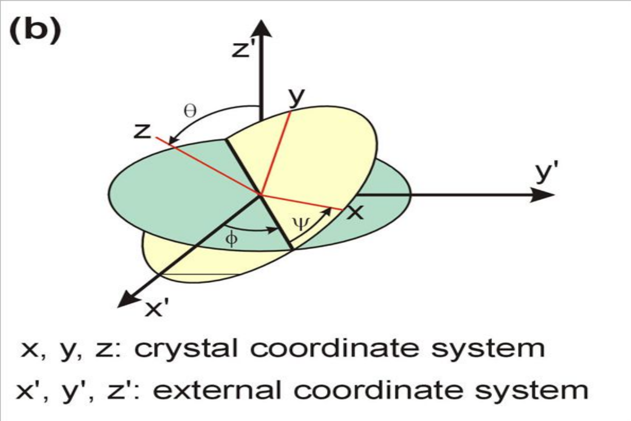
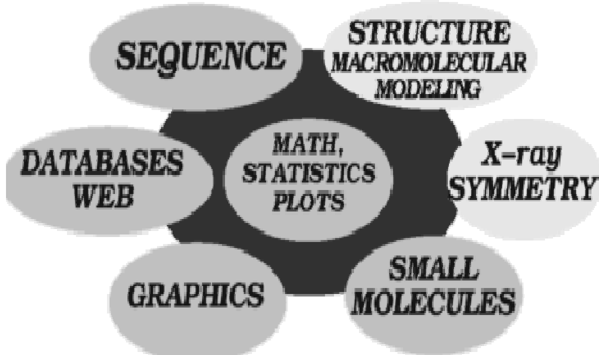


Fig 1: Coordinate Systems

2.2.4.3 Internal Coordinate Mechanics

What is ICM anyway? Does 'I' mean integrated or interactive? Does 'C' stand for computational, coordinate or communication? Is 'M' method, modeling or molecules? Actually, even though all the above translations are relevant, the original meaning was the **Internal Coordinate Mechanics** because the program started in 1985 was aimed at energy optimization of several biopolymers with respect to an arbitrary subset of *internal coordinates* such as bond lengths, bond angles torsion angles and phase angles. The efficient and general global optimization method which evolved from the original ICM method is still the central piece of the program. It is this basic algorithm which is used for peptide prediction, homology modeling and loop simulations, flexible macromolecular docking and refinement. However the complexity of problems related to structure prediction and analysis, as well as the desire for perfection, compactness and consistency, lead to the program expansion into neighboring areas such as graphics, chemistry, sequence analysis and database searches, mathematics, statistics and plotting.

The original meaning became too narrow, but the name sounded good and was kept. The current integrated ICM shell contains hundreds of variables, functions, commands, database and web tools, novel algorithms for structure prediction and analysis into a powerful yet compact program which is still called ICM. The seven principal areas are the following:



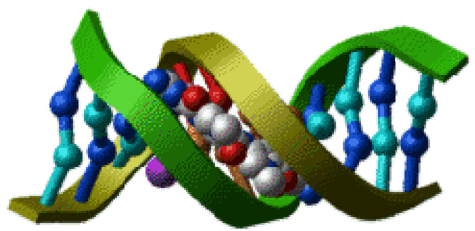
2.2.4.4 What can you do with ICM?

Let us go through the short overview of ICM **applications**.

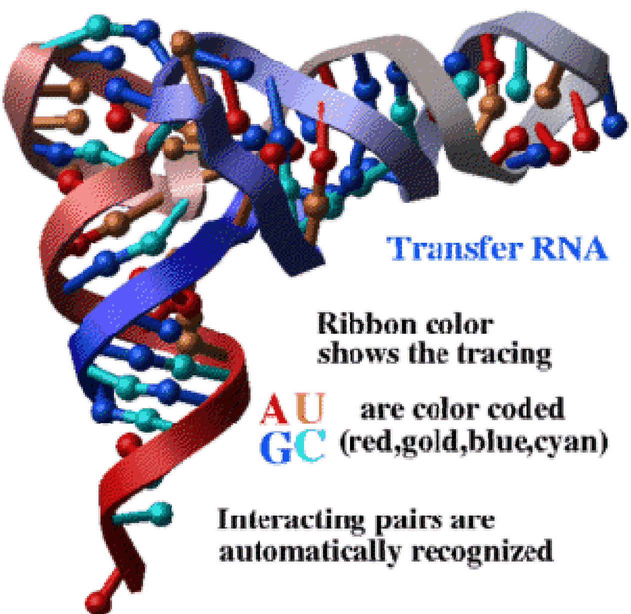
Graphics

Simplified molecular representations are build automatically (e.g. the protein-dna complex is shown with one command: nice "1dnk"). You can combine different types of molecular representations with solid or wire geometrical objects. Molecular representations include wire models, ball-and-stick models, ribbons, space filling models, and skin representation.

The contour-buildup alrorithm calculates the smooth and accurate analytical molecular surface in seconds! This surface can be saved as a geometrical object, saved as a vectorized postscript file.



PDB entry: 101d.brk; ICM command: nice "101d"



PDB entry: 4tna.brk; ICM commands:

```
nice "4tna"
color ribbon a_N/* Count(Nof(a_N/*))
```

Simulations

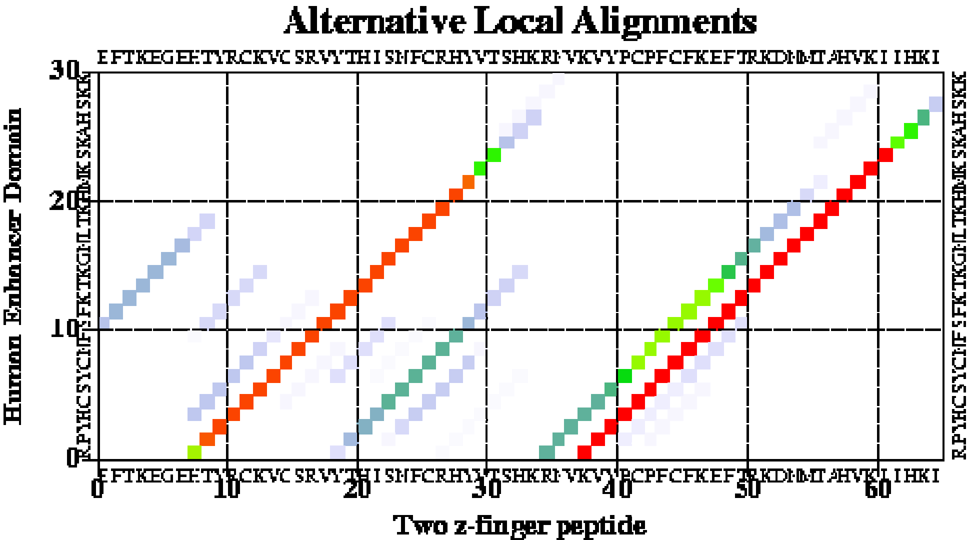
Take a peptide sequence and predict its three-dimensional structure. Of course, the success is not guaranteed, especially if the peptide is longer than about 25 residues but some prelimary tests are encouraging. You will also get a movie of your peptide folding up. Just type the peptide sequence in the _folding file and go ahead.

ICM has a good record in building accurate models by homology. The procedure will build the framework, shake up the side-chains and loops by global energy optimization. You can also color the model by local reliability to identify the potentially wrong parts of the model.

ICM was used to design two new 7 residue loops and in both cases the designs were successful. Moreover, the predicted conformations turned out to be exactly right (accuracy of 0.5A after the crystallographic structures of the designed proteins were determined in Rik Wierenga's lab. Use the _loop script to predict loop conformations and dsEnergyStrain to identify the strained parts of the design.

Sequence analysis

It looks like this:



Here the color shows the local significance of the alignment. You can change the method to calculate probability, color scheme and residue comparison matrices and calculate it interactively of in batch.

Make a pairwise sequence alignment and evaluate the probability that the two aligned sequences share the same structural fold. The alignment is performed with the Needleman and Wunsch algorithm modified to allow zero gap-end penalties (so called ZEGA alignment). The ZEGA probability is a more sensitive indicator of structural significance than the BLAST P-value. The structural statistics was derived by Abagyan and Batalov, 1997: read sequence s_icmhome + "sh3.seq" show Align(Fyn Spec) # the probability will be shown

You can change residue comparison matrices, gap penalties and do many alignments in batch.

Read any number of sequences in fasta or swissprot formats and automatically align the sequences, interactively or in batch. It will look like this:

Consensus ...#.^.YD%..+~..-#~# K~-.#~##.~~..~WW.#. ~~.~G%#P.

Fyn ----VTLFVALYDYEARTEDDLSFHKGEKFQILNSSEGDWWEARSLTTGETGYIPS

Spec DETGKELVLALYDYQEKSPREVTMKKGDILTLLNSTNKDWWKVE--VNDRQGFVP-

Eps8 KTQPKKYAKSKYDFVARNSSELSM-KDDVLELILDDRRQWWKVR---NSGDGFVPN

```
# nID 7 Lmin 56 ID 11.5 %
    #MATGAP gonnet 2.4 0.15
```

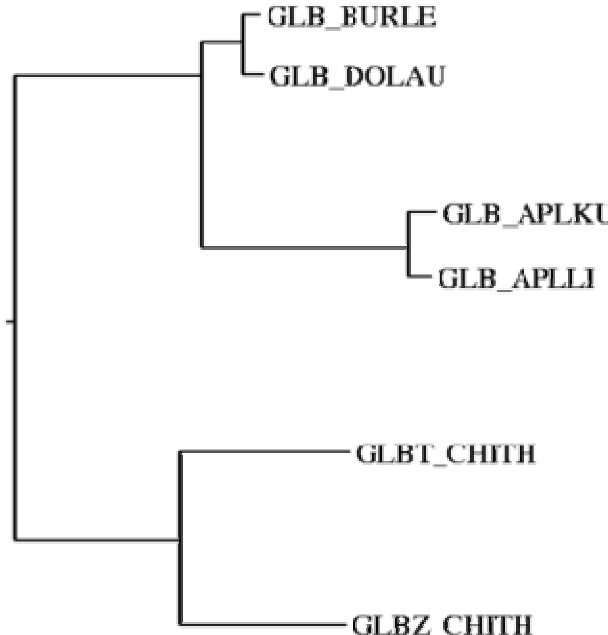
ICM commands:

```
read sequence s_icmhome + "sh3.seq"
group sequences sh3
align sh3
show sh3
```

The Win95/NT version of ICM also has a great interactive alignment editor with dynamic coloring according to conservation. It will automatically show secondary structure and other features.

Relationships between sequences can be presented in three forms:

- as evolutionary trees (ICM uses the neighbor-joining method for tree construction);
- as 2D distribution of sequences using the two main principal axes (use plot2Dseq macro);
- as 3D distribution. This can be analyzed in stereo using controls of molecular graphics (use ds3D macro: ds3D Distance(alig) Name(alig)).



GLB1_ANABR

3D Distribution
of sequences

GLB1_PARCH
GLB1_MORMR
GLB1_ARTSX GLB1_CALSO
GLB1_CHITH GLB1_LUMTE
GLB1_GLYDI

Search your sequence interactively of in batch through any database and generate a list of possible homologues which are sorted and evaluated by probability of structural significance. The sensitive and rigorous ZEGA alignment is used for each comparison. This search may give you more homologues that a BLAST search!

2.2.4.5 ICM consists of four separate modules:

The formal list of features is as follows:

ICM-main:

- shell for molecules, numbers, strings, vectors, matrices, tables, sequences, alignments, profiles, maps
- ICM-language and macros
- graphics, stereo
- imaging and vectorized postscript
- animation and movies
- mathematics, statistics, plotting
- WEB: presentation of the results in the html form
- WEB: user-definition and automated interpretation of web links
- WEB: HTML-form-output interpretation
- molecular conferencing
- pairwise and multiple sequence alignments, evolutionary trees, clustering
- secondary structure prediction and assignment, property profiles, pattern searching
- superpositions, structural alignment, Ramachandran plots
- protein quality check
- analytical molecular surface

- calculations of surface areas and volumes
- cavity analysis
- symmetry operations, access to 230 groups
- database fragment search
- identification of common substructures in PDB
- read pdb, mol2, csd, build from sequence
- energy, solvation, MIMEL, side-chain entropies, soft van der Waals, tethers, distance and angular restraints
- local minimization
- ab initio peptide structure prediction by the Biased Probability Monte Carlo method
- loop simulations
- side-chain placement

ICM-REBEL

- electrostatic free energy calculated by the boundary element method
- coloring molecular surface by electrostatic potential
- binding energy (electrostatic solvation component)
- maps of electrostatic potential and its isopotential contours

ICM-dock

- indexing of chemical databases in SD, mol2 and csd format
- searching and extracting from the indexed databases
- fast grid potentials
- scripts for flexible ligand docking
- scripts for protein-protein docking
- SMILES
- refinement in full atom representation

ICM-bioinformatics

- database indexing and manipulations
- functions to evaluate sequence-structure similarity
- scripts to recognize remote similarities in the protein sequence and PDB databases
- search a pattern through a database
- searching profiles and patterns from the Prosite database through a sequence
- HTML representation of the search results with interpretation of links
- interactive editor of sequence-structure alignment
- automated building of models by homology with loop sampling and side-chain placement

As a method for structure prediction, ICM (internal coordinate mechanics) offers a new efficient way of global energy optimization and versatile modeling operations on arbitrarily fixed multimolecular systems. It is aimed at predicting large structural rearrangements in biopolymers. The ICM-method uses a generalized description of biomolecular structures in which bond lengths, bond angles, torsion and phase angles are considered as independent variables. Any subset of those variables can be fixed.

Rigid bodies formed after exclusion of some variables (i.e. all bond lengths, bond angles and phase angles, or all the variables in a protein domain, etc.) can be treated efficiently in energy calculations, since no interactions withing a rigid body are calculated. Analytical energy derivatives are calculated to allow fast local minimization. To allow large scale conformational sampling and powerful molecular manipulations ICM employs a family of new **global optimization** techniques such as: Biased Probability Monte Carlo (Abagyan and Totrov, 1994), pseudo-Brownian docking method (Abagyan, Totrov and Kuznetsov, 1994) and local deformation loop movements Abagyan and Mazur, 1989).

A set of ECEPP/3 **energy** terms is complemented with solvation energy, **electrostatic polarization energy** and side-chain **entropic** effects (Abagyan and Totrov, 1994), making the total calculated energy a more realistic approximation of the true free energy. Powerful molecular graphics, the ICM-command language, and a set of structure manipulation tools and penalty functions (such as multidimensional variable restraints, tethers, distance restraints) allow the user to address a wide variety of problems concerning biomolecular structures.

Summary:

In the past ten years efficient algorithms have made computationally tractable the use of internal coordinates in molecular dynamics simulations of systems of biological interest (having more than say 100 atoms). Internal coordinates are an attractive alternative to the Cartesian coordinates of each atom when particular degrees of freedom are not of interest. For example, in the process of NMR structure determination and refinement in which one seeks molecular structures consistent with experimental NMR data, the bond lengths and bond angles are generally taken as fixed—and no information about these features is generally available from the NMR experiments.

Model Questions:

1. What are internal coordinates and explain how they can help in structure prediction of biomolecules?
2. What is ICM and explain its biological applications?

References:

1. Internal Coordinates for Molecular Dynamics and Minimization in Structure Determination and Refinement, Journal of Magnetic Resonance **152,** 288–302 (2001), Charles D. Schwieters
2. Abagyan, R.A., Frishman, D., and Argos, P. (1994). Recognition of distantly related proteins through energy calculations. *Proteins* 19, 132-140.

AUTHOR:

B.M.REDDY M.Tech. (HBTI, Kanpur)

Lecturer, Centre for Biotechnology

Acharya Nagarjuna University.

Lesson 2.3.1

NUMERICAL METHODS

Objective

2.3.1.1 Introduction

2.3.1.2 Representation of Physical Phenomena

2.3.1.3 Basis of Numerical Methods

2.3.1.4 Errors and Their Propagation

2.3.1.5 Direct Methods for the Solution of Linear Algebraic Equations

2.3.1.6 Solution of Linear Equations by Iterative Methods

2.3.1.7 Numerical Differentiation

2.3.1.8 Numerical Evaluation of Integrals: Quadrature

2.3.1.9 The Numerical Integration of Differential Equations

2.3.1.10 The Numerical Solution of Integral Equations

Summary

Model Questions

References

Objective:

The objective of this lesson is to know the importance of numerical methods in the exciting field biocomputing.

2.3.1.1 Introduction

The extreme speed of contemporary machines has tremendously expanded the scope of numerical problems that may be considered as well as the manner in which such computational problems may even be approached. However, this expansion of the degree and type of problem that may be numerically solved has removed the scientist from the details of the computation. For this, most would shout "Hooray"! But this removal of the investigator from the details of computation may permit the propagation of errors of various types to intrude and remain undetected. Modern computers will almost always produce numbers, but whether they represent the solution to the problem or the result of error propagation may not be obvious. This situation is made worse by the presence of programs designed for the solution of broad classes of problems. Almost every class of problems has its pathological example for which the standard techniques will fail.

Generally little attention is paid to the recognition of these pathological cases which have an uncomfortable habit of turning up when they are least expected.

The linear equations play an important role in transformation theory and that these equations could be simply expressed in terms of matrices. However, this is only a small segment of the importance of linear equations and matrix theory to the mathematical description of the physical world. Thus we should begin our study of numerical methods with a description of methods for manipulating matrices and solving systems of linear equations. However, before we begin any discussion of numerical methods, we must say something about the accuracy to which those calculations can be made.

2.3.1.2 Representation of Physical Phenomena

The historic representation of physical phenomena falls into two categories referred to as discrete representation and continuous representation. Both ideas appear at about the same time in the Aristotlean period of Greek science (approximately 300 BC). We can use the following definitions to help us understand the means by which physicists develop mental representations of natural phenonmena.

- ***Continuous phenomena*** are ones for which division into smaller and smaller geometrical scale sizes does not alter its fundamental description.

- ***Discrete phenomena*** are ones for which division into smaller and smaller goemetrical scale sizes does lead to a new fundamental description.

Using these definitions we can form a table characterizing both physical nature and physical behavior of phenonmena.

	Behavior	
Nature	**Discrete**	**Continuous**
Discrete	**X**	X
Continuous	**X**	X

The point here is that computers because they are limited to a finite number of digits in expressing a number thus represent the discrete and not the continuous which would require the ability to represent any number no matter how many digits in its expression. Thus when we do a numerical simulation of a continuous phenomena, we are approximating its nature or its behavior. This approximation may be perfectly acceptable for most purposes, but at some level of representing the infinitesimal it will become the limiting factor.

2.3.1.3 Basis of Numerical Methods

The basis of numerical methods is to devise means for approximating infinitesimal processes by finite processes which generally involves the development of an approximating function that can be used in the method in place of the actual function. The exceptions to this idea are algebraic equations in which we are already dealing with finite methods. When we approximate we introduce errors, and so the errors in numerical methods are extremely important to consider.

Errors in Numerical Methods

There are two major sources of errors in numerical methods. The first is called truncation error, while the second is known as round-off error.

Truncation error is caused by the approximations used in the discrete mathematical equations used in a mathematical algorithm to approximate continuous operations. Taylor series are one of the most important means used to derive numerical schemes for computation and to analyze these truncation errors. For a discussion of a Taylor series as an approximating device consult the document Taylor Series.

Round-off errors, on the other hand, are associated with the limited number of digits that are used to represent numbers in a computer. To understand the way in which round-off errors occur in computation, it is necessary to understand how numbers are stored in a computer and how additions and subtractions are performed. For a discussion of round-off errors consult the document Round-Off Error.

Numerical Analysis Topics

The following documents provide a discussion of the numerical techniques for handling various mathematical topics. The discussions are fairly restricted in scope with links to more advanced topics.

- Algebraic Equations
- Approximation
- Differential Calculus
- Integral Calculus
- Ordinary Differential Equations

2.3.1.4 Errors and Their Propagation

One of the most reliable aspects of numerical analysis programs for the electronic digital computer is that they almost always produce numbers. As a result of the considerable reliability of the machines, it is common to regard the results of their calculations with a certain air of infallibility. However, the results can be no better than the method of analysis and implementation program utilized by the computer and these are the works of highly fallible man. This is the origin of the aphorism "garbage in − garbage out". Because of the large number of calculations carried out by these machines, small errors at any given stage can rapidly propagate into large ones that destroy the validity of the result.

We can divide computational errors into two general categories: the first of these we will call round off error, and the second truncation error. Round off error is perhaps the more insidious of the two and is always present at some level. Indeed, its omnipresence indicates the first problem facing us. How accurate an answer do we require? Digital computers utilize a certain number of digits in their calculations and this base number of digits is known as the precision of the machine. Often it is possible to double or triple the number of digits and hence the phrase "double" or "triple" precision is commonly used to describe a calculation carried out using this expanded number of digits. It is a common practice to use more digits than are justified by the problem simply to be sure that one has "got it right". For the scientist, there is a subtle danger in this in that the temptation to publish all the digits presented by the computer is usually overwhelming. Thus published articles often contain numerical results consisting of many more decimal places than are justified by the calculation or the physics that went into the problem. This can lead to some reader unknowingly using the results at an unjustified level of precession thereby obtaining meaningless conclusions. Certainly the full machine precession is never justified, as after the first arithmetical calculation, there will usually be some uncertainty in the value of the last digit. This is the result of the first kind of error we called round off error. As an extreme example, consider a machine that keeps only one significant figure and the exponent of the calculation so that 6+3 will yield 9×10^{0}. However, 6+4, 6+5, and 6+8 will all yield the same answer namely 1×10^{1}. Since the machine only carries one digit, all the other information will be lost. It is not immediately obvious what the result of 6+9, or 7+9 will be. If the result is 2×10^{1}, then the machine is said to round off the calculation to the nearest significant digit. However, if the result remains 1×10^{1}, then the machine is said to truncate the addition to the nearest significant digit. Which is actually done by the computer will depend on both the physical architecture (hardware) of the machine and the programs (software) which instruct it to carry out the operation. Should a human operator be carrying out the calculation, it would usually be possible to see when this is happening and allow for it by keeping additional significant figures, but this is generally not the case with machines. Therefore, we must be careful about the propagation of round off error into the final computational result. It is tempting to say that the above example is only for a 1-digit machine and therefore unrealistic. However, consider the common 6-digit machine. It

will be unable to distinguish between 1 million dollars and 1 million and nine dollars. Subtraction of those two numbers would yield zero. This would be significant to any accountant at a bank. Repeated operations of this sort can lead to a completely meaningless result in the first digit.

This emphasizes the question of 'how accurate an answer do we need?'. For the accountant, we clearly need enough digits to account for all the money at a level decided by the bank. For example, the Internal Revenue Service allows taxpayers to round all calculations to the nearest dollar. This sets a lower bound for the number of significant digits. One's income usually sets the upper bound. In the physical world very few constants of nature are known to more than four digits (the speed of light is a notable exception). Thus the results of physical modeling are rarely important beyond four figures. Again there are exceptions such as in null experiments, but in general, scientists should not deceive themselves into believing their answers are better answers than they are.

How do we detect the effects of round off error? Entire studies have been devoted to this subject by considering that round off errors occurs in basically a random fashion. Although computers are basically deterministic (i.e. given the same initial state, the computer will always arrive at the same answer), a large collection of arithmetic operations can be considered as producing a random collection of round-ups and round-downs. However, the number of digits that are affected will also be variable, and this makes the problem far more difficult to study in general. Thus in practice, when the effects of round off error are of great concern, the problem can be run in double precession. Should both calculations yield the same result at the acceptable level of precession, then round off error is probably not a problem. An additional "rule of thumb" for detecting the presence of round off error is the appearance of a large number of zeros at the right-hand side of the answers. Should the number of zeros depend on parameters of the problem that determine the size or numerical extent of the problem, then one should be concerned about round off error. Certainly one can think of exceptions to this rule, but in general, they are just that - exceptions.

The second form of error we called truncation error and it should not be confused with errors introduced by the "truncation" process that happens half the time in the case of round off errors. This type of error results from the inability of the approximation method to properly represent the solution to the problem. The magnitude of this kind of error depends on both the nature of the problem and the type of approximation technique. For example, consider a numerical approximation technique that will give exact answers should the solution to the problem of interest be a polynomial (we shall show in chapter 3 that the majority of methods of numerical analysis are indeed of this form). Since the solution is exact for polynomials, the extent that the correct solution differs from a polynomial will yield an error. However, there are many different kinds of polynomials and it may be that a polynomial of higher degree approximates the solution more accurately than one of lower degree.

This provides a hint for the practical evaluation of truncation errors. If the calculation is repeated at different levels of approximation (i.e. for approximation methods that are

correct for different degree polynomials) and the answers change by an unacceptable amount, then it is likely that the truncation error is larger than the acceptable amount. There are formal ways of estimating the truncation error and some 'black-box' programs do this. Indeed, there are general programs for finding the solutions to differential equations that use estimates of the truncation error to adjust parameters of the solution process to optimize efficiency. However, one should remember that these estimates are just that - estimates subject to all the errors of calculation we have been discussing. It many cases the correct calculation of the truncation error is a more formidable problem than the one of interest. In general, it is useful for the analyst to have some prior knowledge of the behavior of the solution to the problem of interest before attempting a detailed numerical solution. Such knowledge will generally provide a 'feeling' for the form of the truncation error and the extent to which a particular numerical technique will manage it.

We must keep in mind that both round-off and truncation errors will be present at some level in any calculation and be wary lest they destroy the accuracy of the solution. The acceptable level of accuracy is determined by the analyst and he must be careful not to aim too high and carry out grossly inefficient calculations, or too low and obtain meaningless results.

We now turn to the solution of linear algebraic equations and problems involving matrices associated with those solutions. In general we can divide the approaches to the solution of linear algebraic equations into two broad areas. The first of these involve algorithms that lead directly to a solution of the problem after a finite number of steps while the second class involves an initial "guess" which then is improved by a succession of finite steps, each set of which we will call an iteration. If the process is applicable and properly formulated, a finite number of iterations will lead to a solution.

2.3.1.5 Direct Methods for the Solution of Linear Algebraic Equations

In general, we may write a system of linear algebraic equations in the form

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = c_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = c_2$$
$$a_{31}x_1 + a_{312}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n = c_3$$
$$\vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = c_n$$

$$\mathbf{A}\vec{x} = \vec{c} \qquad \vec{x} = \mathbf{A}^{-1}\vec{c}$$

which in vector notation is Here x is an n-dimensional vector the elements of which represent the solution of the equations. _c is the constant vector of the system of equations and **A** is the matrix of the system's coefficients.

We can write the solution to these equations as thereby reducing the solution of any algebraic system of linear equations to finding the inverse of the coefficient matrix. We shall spend some time describing a number of methods for doing just that. However, there are a number of methods that enable one to find the solution without finding the inverse of the matrix. Probably the best known of these is *Cramer's Rule*

a. Solution by Cramer's Rule

It is unfortunate that usually the only method for the solution of linear equations that students remember from secondary education is Cramer's rule or expansion by minors. As we shall see, this method is rather inefficient and relatively difficult to program for a computer. However, as it forms sort of a standard by which other methods can by judged, we will review it here. The more general definition is inductive so that the determinant of the matrix **A** would be given by

$$\text{Det } \mathbf{A} = \sum_{i=1}^{n} (-1)^{i+j} a_{ij} M_{ij} \text{ , } \forall j$$

Here the summation may be taken over either i or j, or indeed, any monotonically increasing sequence of both. The quantity M_{ij} is the determinant of the matrix **A** with the ith row and jth column removed and, with the sign carried by $(-1)^{(i+j)}$ is called the *cofactor* of the *minor* element a_{ij}. With all this terminology, we can simply write the determinant a

$$\text{Det } \mathbf{A} = \sum_{i-1}^{n} C_{ij} a_{ij} \quad , \quad \forall j \ , \ = \sum_{j-1}^{n} a_{ij} C_{ij} \quad , \forall i$$

By making use of theorems 2 and 7 in section 1.2, we can write the solution in terms of the determinant of **A** as

$$x_1 \begin{vmatrix} a_{11} & a_{12} \cdots & a_{1n} \\ a_{21} & a_{22} \cdots & a_{2n} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ a_{n1} & a_{n2} & a_{nn} \end{vmatrix} = \begin{vmatrix} a_{11}x_1 & a_{12} \cdots & a_{1n} \\ a_{21}x_1 & a_{22} \cdots & a_{2n} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ a_{n1}x_1 & a_{n2} & a_{nn} \end{vmatrix} = \begin{vmatrix} (a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n) & a_{12} \cdots & a_{1n} \\ (a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n) & a_{22} \cdots & a_{2n} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ (a_{1n}x_1 + a_{1n}x_2 + \cdots + a_{1n}x_n) & a_{n2} & a_{nn} \end{vmatrix}$$

$$= \begin{vmatrix} c_1 & a_{12} \cdots & a_{1n} \\ c_2 & a_{22} \cdots & a_{2n} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ c_n & a_{n2} & a_{nn} \end{vmatrix}$$

which means that the general solution of equation (2.2.1) is given by

$$x_j = \begin{vmatrix} a_{11} \cdots a_{1\,j-1} c_1 & a_{1\,j+1} \cdots a_{1n} \\ a_{21} \cdots a_{2\,j-1} c_2 & a_{2\,j+1} \cdots a_{2n} \\ \cdot \quad \cdot & \cdot \quad \cdot \\ \cdot \quad \cdot & \cdot \quad \cdot \\ a_{n1} \cdots a_{n\,j-1} c_n & a_{n\,j+1} \cdots a_{nn} \end{vmatrix} \times [\mathrm{Det}\,\mathbf{A}]^{-1}$$

This requires evaluating the determinant of the matrix **A** as well as an augmented matrix where the jth column has been replaced by the elements of the constant vector c_i. Evaluation of the determinant of an n×n matrix requires about $3n^2$ operations and this must be repeated for each unknown, thus solution by Cramer's rule will require at least $3n^3$ operations. In addition, to maintain accuracy, an optimum path through the matrix (finding the least numerically sensitive cofactors) will require a significant amount of logic. Thus, solution by Cramer's rule is not a particularly desirable approach to the numerical solution of linear equations either for a computer or a hand calculation. Let us consider a simpler algorithm, which forms the basis for one of the most reliable and stable direct methods for the solution of linear equations. It also provides a method for the inversion of matrices. Let begin by describing the method and then trying to understand why it works.

2.3.1.6 Solution of Linear Equations by Iterative Methods

So far we have dealt with methods that will provide a solution to a set of linear equations after a finite number of steps (generally of the order of n^3). The accuracy of the solution at the end of this sequence of steps is fixed by the nature of the equations and to a lesser extent by the specific algorithm that is used. We will now consider a series of algorithms that provide answers to a linear system of equations in considerably fewer steps, but at a level of accuracy that will depend on the number of times the algorithm is applied. Such methods are generally referred to as iterative methods and they usually require of the order of n^2 steps for each iteration. Clearly for very large systems of equations, these methods may prove very much faster than direct methods providing they converge quickly to an accurate solution.

a. Solution by the Gauss and Gauss-Seidel Iteration Methods

All iterative schemes begin by assuming that an approximate answer is known and then the scheme proceeds to improve that answer. Thus we will have a solution vector that is constantly changing from iteration to iteration. In general, we will denote this by a superscript in parentheses so that $x^{(i)}$ will denote the value of x at the ith iteration. Therefore in order to begin, we will need an initial value of the solution vector . The concept of the Gauss iteration scheme is extremely simple. Take the system of linear equations as expressed in equations (2.2.1) and solve each one for the diagonal value of x so that

$$x_i = \frac{\left[c_i - \sum_{i \neq j}^{n} a_{ij} x_j \right]}{a_{ii}} \quad . \qquad (2.3.1)$$

Now use the components of the initial value of on the right hand side of equation (2.3.1) to obtain an improved value for the elements. This procedure can be repeated until a solution of the desired accuracy is obtained. Thus the general iteration formula would have the form

$$x_i^{(k)} = \frac{\left[c_i - \sum_{i \ne j}^{n} a_{ij} x_j^{(k-1)} \right]}{a_{ii}} \quad . \qquad (2.3.2)$$

It is clear, that should any of the diagonal elements be zero, there will be a problem with the stability of the method. Thus the order in which the equations are arranged will make a difference to in the manner in which this scheme proceeds. One might suppose that the value of the initial guess might influence whether or not the method would find the correct answer, but as we shall see in section 2.4 that is not the case. However, the choice of the initial guess will determine the number of iterations required to arrive at an acceptable answer.

The Gauss-Seidel scheme is an improvement on the basic method of Gauss. Let us rewrite equations (2.3.1) as follows:

$$x_i^{(k)} = \frac{\left[c_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k-1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \right]}{a_{ii}} \quad . \qquad (2.3.3)$$

When using this as a basis for an iteration scheme, we can note that all the values of xj in the first summation for the kth iteration will have been determined before the value of xi(k) so that we could write the iteration scheme as

$$x_i^{(k)} = \frac{\left[c_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \right]}{a_{ii}} \quad . \qquad (2.3.4)$$

Here the improved values of x_i are utilized as soon as they are obtained. As one might expect, this can lead to a faster rate of convergence, but there can be a price for the improved speed. The Gauss-Seidel scheme may not be as stable as the simple Gauss method. In general, there seems to be a trade off between speed of convergence and the stability of iteration schemes.

Indeed, if we were to apply either if the Gauss iterative methods to equations (2.2.13) that served as an example for the direct method, we would find that the iterative solutions would not converge. We shall see later (sections 2.3d and 2.4) that those equations fail to satisfy the simple sufficient convergence criteria given in section 2.3d and the necessary and sufficient condition of section 2.4. With that in mind, let us consider another 3×3 system of equations which does satisfy those conditions. These equations are much more strongly diagonal than those of equation (2.2.13) so

$$\left.\begin{aligned} 3x_1 + x_2 + x_3 &= 8 \\ x_1 + 4x_2 + 2x_3 &= 15 \\ 2x_1 + x_2 + 5x_3 &= 19 \end{aligned}\right\} . \qquad (2.3.5)$$

For these equations, the solution under the Gauss-iteration scheme represented by equations (2.3.2) takes the form

$$\left.\begin{aligned} x_1^{(k+1)} &= \left[8 - x_2^{(k)} - x_3^{(k)}\right] / 3 \\ x_2^{(k+1)} &= \left[15 - x_1^{(k)} - 2x_3^{(k)}\right] / 4 \\ x_3^{(k+1)} &= \left[19 - 2x_1^{(k)} - x_2^{(k)}\right] / 5 \end{aligned}\right\} . \qquad (2.3.6)$$

However, if we were to solve equations (2.3.5) by means of the Gauss-Seidel method the iterative equations for the solution would be

$$\left.\begin{aligned} x_1^{(k+1)} &= \left[8 - x_2^{(k)} - x_3^{(k)}\right] / 3 \\ x_2^{(k+1)} &= \left[15 - x_1^{(k+1)} - 2x_3^{(k)}\right] / 4 \\ x_3^{(k+1)} &= \left[19 - 2x_1^{(k+1)} - x_2^{(k+1)}\right] / 5 \end{aligned}\right\} . \qquad (2.3.7)$$

If we take the initial guess to be

$$x_1^{(0)} = x_2^{(0)} = x_3^{(0)} = 1 , \qquad (2.3.8)$$

then repetitive use of equations (2.3.6) and (2.3.7) yield the results given in Table 2.1.

Table 2.1
Convergence of Gauss and Gauss-Seidel Iteration Schemes

K	0		1		2		3		4		5		10	
	G	GS	G	GS	G	GS	G	GS	G	GS	G	GS	G	GS
x_1	1.00	1.00	2.00	2.00	0.60	0.93	1.92	0.91	0.71	0.98	1.28	1.00	0.93	1.00
x_2	1.00	1.00	3.00	2.75	1.65	2.29	2.64	2.03	1.66	1.99	2.32	2.00	1.92	2.00
x_3	1.00	1.00	3.20	2.45	1.92	2.97	3.23	3.03	2.51	3.01	3.18	3.00	2.95	3.00

As is clear from the results labeled "G" in table 2.1, the Gauss-iteration scheme converges very slowly. The correct solution which would eventually be obtained is

$$\vec{x}^{(\infty)} = [\,1,2,3\,] \quad . \qquad (2.3.9)$$

There is a tendency for the solution to oscillate about the correct solution with the amplitude slowly damping out toward convergence. However, the Gauss-Seidel iteration method damps this oscillation very rapidly by employing the improved values of the solution as soon as they are obtained. As a result, the Gauss-Seidel scheme has converged on this problem in about 5 iterations while the straight Gauss scheme still shows significant error after 10 iterations.

2.3.1.7 Numerical Differentiation

Compared with other subjects to be covered in the study of numerical methods, little is usually taught about numerical differentiation. Perhaps that is because the processes should be avoided whenever possible. The reason for this can be seen in the nature of polynomials. As on interpolation, high degree polynomials tend to oscillate between the points of constraint. Since the derivative of a polynomial is itself a polynomial, it too will oscillate between the points of constraint, but perhaps not quite so wildly. To minimize this oscillation, one must use low degree polynomials which then tend to reduce the accuracy of the approximation. Another way to see the dangers of numerical differentiation is to consider the nature of the operator itself.

2.3.1.8 Numerical Evaluation of Integrals: Quadrature

While the term *quadrature* is an old one, it is the correct term to use for describing the numerical evaluation of integrals. The term *numerical integration* should be reserved for describing the numerical solution of differential equations (see chapter 5). There is a genuine necessity for the distinction because the very nature of the two problems is quite different. Numerically evaluating an integral is a rather common and usually stable task. One is basically assembling a single number from a series of independent evaluations of a function. Unlike numerical differentiation, numerical quadrature tends to average out random computational errors.

Because of the inherent stability of numerical quadrature, students are generally taught only the simplest of techniques and thereby fail to learn the more sophisticated, highly efficient techniques that can be so important when the integrand of the integral is extremely complicated or occasionally the result of a separate lengthy study. Virtually all numerical quadrature schemes are based on the notion of polynomial approximation. Specifically, the quadrature scheme will give the exact value of the integral if the integrand is a polynomial of some degree n. The scheme is then said to have a *degree of precision* equal to n. In general, since a nth degree polynomial has n+1 linearly independent coefficients, a quadrature scheme will have to have n+1 adjustable parameters in order to accurately represent the polynomial and its integral. Occasionally, one comes across a quadrature scheme that has a degree of precision that is greater than the number of adjustable parameters. Such a scheme is said to be hyper-efficient and there are a number of such schemes known for multiple integrals.

2.3.1.9 The Numerical Integration of Differential Equations

When we speak of a differential equation, we simply mean any equation where the dependent variable appears as well as one or more of its derivatives. The highest

derivative that is present determines the *order* of the differential equation while the highest power of the dependent variable or its derivative appearing in the equation sets its *degree*. Theories which employ differential equations usually will not be limited to single equations, but may include sets of simultaneous equations representing the phenomena they describe. Thus, we must say something about the solutions of sets of such equations. Indeed, changing a high order differential equation into a system of first order differential equations is a standard approach to finding the solution to such equations. Basically, one simply replaces the higher order terms with new variables and includes the equations that define the new variables to form a set of first order simultaneous differential equations that replace the original equation.

2.3.1.10 The Numerical Solution of Integral Equations

For reasons that I have never fully understood, the mathematical literature is crowded with books, articles, and papers on the subject of differential equations. Most universities have several courses of study in the subject, but little attention is paid to the subject of integral equations. The differential operator is linear and so is the integral operator. Indeed, one is just the inverse of the other. Equations can be written where the dependent variable appears under an integral as well as alone. Such equations are the analogue of the differential equations and are called *integral equations*. It is often possible to turn a differential equation into an integral equation which may make the problem easier to numerically solve. Indeed many physical phenomena lend themselves to description by integral equations. So one would think that they might form as large an area for analysis are do the differential equations. Such is not the case. Indeed, we will not be able to devote as much time to the discussion of these interesting equations as we should, but we shall spend enough time so that the student is at least familiar with some of their basic properties. Of necessity, we will restrict our discussion to those integral equations where the unknown appears linearly. Such linear equations are more tractable and yet describe much that is of interest in science.

Summary:

The extreme speed of contemporary machines has tremendously expanded the scope of numerical problems that may be considered as well as the manner in which such computational problems may even be approached. However, this expansion of the degree and type of problem that may be numerically solved has removed the scientist from the details of the computation. For this, most would shout "Hooray"! But this removal of the investigator from the details of computation may permit the propagation of errors of various types to intrude and remain undetected. Modern computers will almost always produce numbers, but whether they represent the solution to the problem or the result of error propagation may not be obvious. This situation is made worse by the presence of programs designed for the solution of broad classes of problems. Almost every class of problems has its pathological example for which the standard techniques will fail. Generally little attention is paid to the recognition of these pathological cases which have an uncomfortable habit of turning up when they are least expected.

Model Questions:

1. What are the applications of Numerical Methods?
2. Explain the errors of Numerical Methods and their propagation?

References:

1. Fundamental Numerical Methods and Data Analysis by George W. Collins, II
2. Numerical Analysis Basics by J. C. Evans

AUTHOR:

B.M.REDDY **M.Tech. (HBTI, Kanpur)**

Lecturer, Centre for Biotechnology

Acharya Nagarjuna University.

Lesson 2.3.2

OPTIMIZATION TECHNIQUES

Objective
2.3.2.1 Introduction
2.3.2.2 Techniques
2.3.2.3 Use of Optimization Techniques
2.3.2.4 Nonlinearly Constrained QN Optimization
2.3.2.5 Optimization and Iteration History
2.3.2.6 Line-Search Methods
2.3.2.7 Restricting the Step Length
2.3.2.8 Techniques of global optimization
Summary
Model Questions
References

Objective: The objective of this lesson is to have an exposure of optimization and global optimization techniques which have an application in structure prediction.

2.3.2.1 Introduction

In mathematics, the term **optimization** refers to the study of problems that have the form

Given: a function $f : A \longrightarrow \mathbf{R}$ from some set A to the real numbers

Sought: an element x_0 in A such that $f(x_0) \leq f(x)$ for all x in A ("minimization") or such that $f(x_0) \geq f(x)$ for all x in A ("maximization").

Such a formulation is sometimes called a *mathematical program* (a term not directly related to computer programming, but still in use for example for linear programming). Many real-world and theoretical problems may be modeled in this general framework.

Typically, A is some subset of the Euclidean space $\mathbf{R}^n$, often specified by a set of *constraints*, equalities or inequalities that the members of A have to satisfy. The elements of A are called *feasible solutions*. The function f is called an *objective function*, or *cost function*. A feasible solution that minimizes (or maximizes, if that is the goal) the objective function is called an *optimal solution*.

In general, there will be several local minima and maxima, where a *local minimum* $\mathbf{x}^*$ is defined as a point such that for some $\delta > 0$ and all x such that

$$\|\mathbf{x} - \mathbf{x}^*\| \leq \delta;$$

the formula

$$f(\mathbf{x}^*) \leq f(\mathbf{x})$$

holds; that is to say, on some ball around x^* all of the function values are greater than or equal to the value at that point. Local maxima are defined similarly. In general, it is easy to find local minima — additional facts about the problem (e.g. the function being convex) are required to ensure that the solution found is a global minimum.

Notation

Optimization problems are often expressed with special notation. Here are some examples:

$$\min_{x \in \mathbb{R}} \; x^2 + 1$$

This asks for the minimum value for the expression $x^2 + 1$, where x ranges over the real numbers **R**. The minimum value in this case is 1, occurring at $x = 0$.

$$\max_{x \in \mathbb{R}} \; 2x$$

This asks for the maximum value for the expression $2x$, where x ranges over the reals. In this case, there is no such maximum as the expression is unbounded, so the answer is "infinity" or "undefined".

$$\operatorname{argmin}_{x \in [-\infty, -1]} \; x^2 + 1$$

This asks for the value(s) of x in the interval $[-\infty, -1]$ which minimizes the expression $x^2 + 1$. (The actual minimum value of that expression does not matter.) In this case, the answer is $x = -1$.

$$\operatorname{argmax}_{x \in [-5,5],\, y \in \mathbb{R}} \; x \cdot \cos(y)$$

This asks for the (x, y) pair(s) that maximize the value of the expression $x \cdot \cos(y)$, with the added constraint that x lies in the interval $[-5, 5]$. (Again, the actual maximum value of the expression does not matter.) In this case, the solutions are the pairs of the form $(5, 2\pi k)$ and $(-5, (2k + 1)\pi)$, where k ranges over all integers.

2.3.2.2 Techniques

For twice-differentiable functions, unconstrained problems can be solved by finding the points where the gradient of the objective function is 0 (that is, the stationary points) and using the Hessian matrix to classify the type of each point. If the Hessian is positive definite, the point is a local minimum, if negative definite, a local maximum, and if indefinite it is some kind of saddle point.

One can find the stationary points by starting with a guess for a stationary point, and then iterate towards it by using methods such as

- gradient descent
- Newton's method
- conjugate gradient
- line search

Should the objective function be convex over the region of interest, then any local minimum will also be a global minimum. There exist robust, fast numerical techniques for optimizing doubly differentiable convex functions.

Constrained problems can often be transformed into unconstrained problems with the help of Lagrange multipliers.

2.3.2.3 Use of Optimization Techniques

No algorithm for optimizing general nonlinear functions exists that will always find the global optimum for a general nonlinear minimization problem in a reasonable amount of time. Since no single optimization technique is invariably superior to others, PROC CALIS provides a variety of optimization techniques that work well in various circumstances. However, you can devise problems for which none of the techniques in PROC CALIS will find the correct solution. All optimization techniques in PROC CALIS use $O(n^2)$ memory except the conjugate gradient methods, which use only $O(n)$ of memory and are designed to optimize problems with many parameters.

The PROC CALIS statement NLOPTIONS can be especially helpful for tuning applications with nonlinear equality and inequality constraints on the parameter estimates. Some of the options available in NLOPTIONS may also be invoked as PROC CALIS options. The NLOPTIONS statement can specify almost the same options as the SAS/OR NLP procedure.

Nonlinear optimization requires the repeated computation of

- the function value (optimization criterion)
- the gradient vector (first-order partial derivatives)
- for some techniques, the (approximate) Hessian matrix (second-order partial derivatives)
- values of linear and nonlinear constraints
- the first-order partial derivatives (Jacobian) of nonlinear constraints

For the criteria used by PROC CALIS, computing the gradient takes more computer time than computing the function value, and computing the Hessian takes *much* more computer time and memory than computing the gradient, especially when there are many parameters to estimate. Unfortunately, optimization techniques that do not use the Hessian usually require many more iterations than techniques that do use the (approximate) Hessian, and so they are often slower. Techniques that do not use the Hessian also tend to be less reliable (for example, they may terminate at local rather than global optima).

Problems in rigid body dynamics (in particular articulated rigid body dynamics) often require mathematical programming techniques, since you can view rigid body dynamics as attempting to solve an ordinary differential equation on a constraint manifold; the constraints are various nonlinear geometric constraints such as "these two points must always coincide", "this surface must not penetrate any other", or "this point must always lie somewhere on this curve". Also, the problem of computing contact forces can be done by solving a linear complementarity problem, which can also be viewed as a QP (quadratic programming problem).

Many design problems can also be expressed as optimization programs. This application is called design optimization. One recent and growing subset of this field is multidisciplinary design optimization, which, while useful in many problems, has in particular been applied to aerospace engineering problems.

Another field that uses optimization techniques extensively is operations research

The available optimization techniques are displayed in Table 1 and can be chosen by the TECH=***name*** option.

Table1: Optimization Techniques

TECH =	**Optimization Technique**
LEVMAR	Levenberg-Marquardt Method
TRUREG	Trust-Region Method
NEWRAP	Newton-Raphson Method with Line Search
NRRIDG	Newton-Raphson Method with Ridging
QUANEW	Quasi-Newton Methods (DBFGS, DDFP, BFGS, DFP)
DBLDOG	Double-Dogleg Method (DBFGS, DDFP)
CONGRA	Conjugate Gradient Methods (PB, FR, PR, CD)

Table shows, for each optimization technique, which derivatives are needed (first-order or second-order) and what kind of constraints (boundary, linear, or nonlinear) can be imposed on the parameters.

The Levenberg-Marquardt, trust-region, and Newton-Raphson techniques are usually the most reliable, work well with boundary and general linear constraints, and generally converge after a few iterations to a precise solution. However, these techniques need to compute a Hessian matrix in each iteration. For HESSALG=1, this means that you need about $4(n(n+1)/2)t$ bytes of work memory (n = the number of manifest variables, t = the number of parameters to estimate) to store the Jacobian and its cross product. With HESSALG=2 or HESSALG=3, you do not need this work memory, but the use of a utility file increases execution time. Computing the approximate Hessian in each iteration can be very time- and memory-consuming, especially for large problems (more than 60 or 100 parameters, depending on the computer used). For large problems, a quasi-Newton technique, especially with the BFGS update, can be far more efficient.

For a poor choice of initial values, the Levenberg-Marquardt method seems to be more reliable.

If memory problems occur, you can use one of the conjugate gradient techniques, but they are generally slower and less reliable than the methods that use second-order information.

There are several options to control the optimization process. First of all, you can specify various termination criteria. You can specify the GCONV= option to specify a relative gradient termination criterion. If there are active boundary constraints, only those gradient components that correspond to inactive constraints contribute to the criterion. When you want very precise parameter estimates, the GCONV= option is useful. Other criteria that use relative changes in function values or parameter estimates in consecutive iterations can lead to early termination when active constraints cause small steps to occur. The small default value for the FCONV= option helps prevent early termination. Using the MAXITER= and MAXFUNC= options enables you to specify the maximum number of iterations and function calls in the optimization process. These limits are especially useful in combination with the INRAM= and OUTRAM= options; you can run a few iterations at a time, inspect the results, and decide whether to continue iterating.

2.3.2.4 Nonlinearly Constrained QN Optimization

The algorithm used for nonlinearly constrained quasi-Newton optimization is an efficient modification of Powell's *Variable Metric Constrained WatchDog* (VMCWD) algorithm. A similar but older algorithm (VF02AD) is part of the Harwell library. Both VMCWD and VF02AD use Fletcher's VE02AD algorithm (also part of the Harwell library) for positive definite quadratic programming. The PROC CALIS QUANEW implementation uses a quadratic programming subroutine that updates and downdates the approximation of the Cholesky factor when the active set changes. The nonlinear QUANEW algorithm is not a feasible point algorithm, and the value of the objective function need not decrease (minimization) or increase (maximization) monotonically. Instead, the algorithm tries to reduce a linear combination of the objective function and constraint violations, called the *merit function*.

The following are similarities and differences between this algorithm and VMCWD:

- A modification of this algorithm can be performed by specifying VERSION=1, which replaces the update of the Lagrange vector μ with the original update of Powell, which is used in VF02AD. This can be helpful for some applications with linearly dependent active constraints.
- If the VERSION= option is not specified or VERSION=2 is specified, the evaluation of the Lagrange vector μ is performed in the same way as Powell describes.
- Instead of updating an approximate Hessian matrix, this algorithm uses the dual BFGS (or DFP) update that updates the Cholesky factor of an approximate Hessian. If the condition of the updated matrix gets too bad, a restart is done with a positive diagonal matrix. At the end of the first iteration after each restart, the Cholesky factor is scaled.
- The Cholesky factor is loaded into the quadratic programming subroutine, automatically ensuring positive definiteness of the problem. During the quadratic programming step, the Cholesky factor of the projected Hessian matrix $\mathbf{Z}'_k\mathbf{G}\mathbf{Z}_k$ and the $\mathbf{QT}$ decomposition are updated simultaneously when the active set changes. Refer to Gill et al. (1984) for more information.
- The line-search strategy is very similar to that of Powell. However, this algorithm does not call for derivatives during the line search; hence, it generally needs fewer derivative calls than function calls. The VMCWD algorithm always requires the same number of derivative and function calls. It was also found in several applications of VMCWD that Powell's line-search method sometimes uses steps that are too long during the first iterations. In those cases, you can use the INSTEP= option specification to restrict the step length α of the first iterations.
- Also the watchdog strategy is similar to that of Powell (1982a, 1982b). However, this algorithm doesn't return automatically after a fixed number of iterations to a former better point. A return here is further delayed if the observed function reduction is close to the expected function reduction of the quadratic model.
- Although Powell's termination criterion still is used (as FCONV2), the QUANEW implementation uses two additional termination criteria (GCONV and ABSGCONV).

This algorithm is automatically invoked when you specify the NLINCON statement. The nonlinear QUANEW algorithm needs the Jacobian matrix of the first-order derivatives (constraints normals) of the constraints

$$(\nabla c_i) = (\frac{\partial c_i}{\partial x_j}), \quad i = 1, \ldots, nc, j = 1, \ldots, n$$

where ***nc*** is the number of nonlinear constraints for a given point ***x***.

You can specify two update formulas with the UPDATE= option:

- UPDATE=DBFGS performs the dual BFGS update of the Cholesky factor of the Hessian matrix. This is the default.
- UPDATE=DDFP performs the dual DFP update of the Cholesky factor of the Hessian matrix.

This algorithm uses its own line-search technique. All options and parameters (except the INSTEP= option) controlling the line search in the other algorithms do not apply here. In several applications, large steps in the first iterations are troublesome. You can specify the INSTEP= option to impose an upper bound for the step size α during the first five iterations. The values of the LCSINGULAR=, LCEPSILON=, and LCDEACT= options, which control the processing of linear and boundary constraints, are valid only for the quadratic programming subroutine used in each iteration of the nonlinear constraints QUANEW algorithm.

2.3.2.5 Optimization and Iteration History

The optimization and iteration histories are displayed by default because it is important to check for possible convergence problems.

The optimization history includes the following summary of information about the initial state of the optimization.

- the number of constraints that are active at the starting point, or more precisely, the number of constraints that are currently members of the working set. If this number is followed by a plus sign, there are more active constraints, of which at least one is temporarily released from the working set due to negative Lagrange multipliers.
- the value of the objective function at the starting point
- if the (projected) gradient is available, the value of the largest absolute (projected) gradient element
- for the TRUREG and LEVMAR subroutines, the initial radius of the trust region around the starting point

The optimization history ends with some information concerning the optimization result:

- the number of constraints that are active at the final point, or more precisely, the number of constraints that are currently members of the working set. If this number is followed by a plus sign, there are more active constraints, of which at least one is temporarily released from the working set due to negative Lagrange multipliers.
- the value of the objective function at the final point

- if the (projected) gradient is available, the value of the largest absolute (projected) gradient element
- other information specific to the optimization technique

The iteration history generally consists of one line of displayed output containing the most important information for each iteration. The _LIST_ variable (see the "SAS Program Statements" section) also enables you to display the parameter estimates and the gradient in some or all iterations.

The iteration history always includes the following (the words in parentheses are the column header output):

- the iteration number (Iter)
- the number of iteration restarts (rest)
- the number of function calls (nfun)
- the number of active constraints (act)
- the value of the optimization criterion (optcrit)
- the difference between adjacent function values (difcrit)
- the maximum of the absolute gradient components corresponding to inactive boundary constraints (maxgrad)

An apostrophe trailing the number of active constraints indicates that at least one of the active constraints is released from the active set due to a significant Lagrange multiplier.

For the Levenberg-Marquardt technique (LEVMAR), the iteration history also includes the following information:

- An asterisk trailing the iteration number means that the computed Hessian approximation is singular and consequently ridged with a positive lambda value. If all or the last several iterations show a singular Hessian approximation, the problem is not sufficiently identified. Thus, there are other locally optimal solutions that lead to the same optimum function value for different parameter values. This implies that standard errors for the parameter estimates are not computable without the addition of further constraints.
- the value of the Lagrange multiplier (lambda); this is 0 if the optimum of the quadratic function approximation is inside the trust region (a trust-region-scaled Newton step can be performed) and is greater than 0 when the optimum of the quadratic function approximation is located at the boundary of the trust region (the scaled Newton step is too long to fit in the trust region and a quadratic constraint optimization is performed). Large values indicate optimization difficulties. For a nonsingular Hessian matrix, the value of lambda should go to 0 during the last iterations, indicating that the objective function can be well approximated by a quadratic function in a small neighborhood of the optimum point. An increasing lambda value often indicates problems in the optimization process.

- the value of the ratio ρ (rho) between the actually achieved difference in function values and the predicted difference in the function values on the basis of the quadratic function approximation. Values much less than 1 indicate optimization difficulties. The value of the ratio ρ indicates the goodness of the quadratic function approximation; in other words, $\rho << 1$ means that the radius of the trust region has to be reduced. A fairly large value of ρ means that the radius of the trust region need not be changed. And a value close to or larger than 1 means that the radius can be increased, indicating a good quadratic function approximation.

For the Newton-Raphson technique (NRRIDG), the iteration history also includes the following information:

- the value of the ridge parameter. This is 0 when a Newton step can be performed, and it is greater than 0 when either the Hessian approximation is singular or a Newton step fails to reduce the optimization criterion. Large values indicate optimization difficulties.

- the value of the ratio ρ (rho) between the actually achieved difference in function values and the predicted difference in the function values on the basis of the quadratic function approximation. Values much less than 1.0 indicate optimization difficulties.

For the Newton-Raphson with line-search technique (NEWRAP), the iteration history also includes

- the step size α (alpha) computed with one of the line-search algorithms
- the slope of the search direction at the current parameter iterate. For minimization, this value should be significantly negative. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

For the Trust-Region technique (TRUREG), the iteration history also includes the following information:

- An asterisk after the iteration number means that the computed Hessian approximation is singular and consequently ridged with a positive lambda value.
- the value of the Lagrange multiplier (lambda). This value is zero when the optimum of the quadratic function approximation is inside the trust region (a trust-region-scaled Newton step can be performed) and is greater than zero when the optimum of the quadratic function approximation is located at the boundary of the trust region (the scaled Newton step is too long to fit in the trust region and a quadratically constrained optimization is performed). Large values indicate optimization difficulties. As in Gay (1983), a negative lambda value indicates the

special case of an indefinite Hessian matrix (the smallest eigenvalue is negative in minimization).

- the value of the radius Δ of the trust region. Small trust region radius values combined with large lambda values in subsequent iterations indicate optimization problems.

For the quasi-Newton (QUANEW) and conjugate gradient (CONGRA) techniques, the iteration history also includes the following information:

- the step size (alpha) computed with one of the line-search algorithms
- the descent of the search direction at the current parameter iterate. This value should be significantly smaller than 0. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

Frequent update restarts (rest) of a quasi-Newton algorithm often indicate numerical problems related to required properties of the approximate Hessian update, and they decrease the speed of convergence. This can happen particularly if the ABSGCONV= termination criterion is too small, that is, when the requested precision cannot be obtained by quasi-Newton optimization. Generally, the number of automatic restarts used by conjugate gradient methods are much higher.

For the nonlinearly constrained quasi-Newton technique, the iteration history also includes the following information:

- the maximum value of all constraint violations,

$$\textbf{conmax} = \max(|c_i(x)| : c_i(x) < 0)$$

- the value of the predicted function reduction used with the GCONV and FCONV2 termination criteria,

$$\textbf{pred} = |g(x^{(k)})s(x^{(k)})| + \sum_{i=1}^{m} |\lambda_i c_i(x^{(k)})|$$

- the step size α of the quasi-Newton step. Note that this algorithm works with a special line-search algorithm.
- the maximum element of the gradient of the Lagrange function,

$$\begin{aligned}\text{lfgmax} &= \nabla_x L(x^{(k)}, \lambda^{(k)}) \\ &= \nabla_x f(x^{(k)}) - \sum_{i=1}^{m} \lambda_i^{(k)} \nabla_x c_i(x^{(k)})\end{aligned}$$

For the double dogleg technique, the iteration history also includes the following information:

- the parameter λ of the double-dogleg step. A value $\lambda = 0$ corresponds to the full (quasi) Newton step.
- the slope of the search direction at the current parameter iterate. For minimization, this value should be significantly negative.

2.3.2.6 Line-Search Methods

In each iteration ***k***, the (dual) quasi-Newton, hybrid quasi-Newton, conjugate gradient, and Newton-Raphson minimization techniques use iterative line-search algorithms that try to optimize a linear, quadratic, or cubic approximation of the nonlinear objective function ***f*** of ***n*** parameters ***x*** along a feasible descent search direction $s^{(k)}$

$$f(x^{(k+1)}) = f(x^{(k)} + \alpha^{(k)} s^{(k)})$$

by computing an approximately optimal scalar $\alpha^{(k)} > 0$ .Since the outside iteration process is based only on the approximation of the objective function, the inside iteration of the line-search algorithm does not have to be perfect. Usually, it is satisfactory that the choice of α significantly reduces (in a minimization) the objective function. Criteria often used for termination of line-search algorithms are the Goldstein conditions (Fletcher 1987).

Various line-search algorithms can be selected by using the LIS= option. The line-search methods LIS=1, LIS=2, and LIS=3 satisfy the left-hand-side and right-hand-side Goldstein conditions (refer to Fletcher 1987). When derivatives are available, the line-search methods LIS=6, LIS=7, and LIS=8 try to satisfy the right-hand-side Goldstein condition; if derivatives are not available, these line-search algorithms use only function calls.

The line-search method LIS=2 seems to be superior when function evaluation consumes significantly less computation time than gradient evaluation. Therefore, LIS=2 is the default value for Newton-Raphson, (dual) quasi-Newton, and conjugate gradient optimizations.

2.3.2.7 Restricting the Step Length

Almost all line-search algorithms use iterative extrapolation techniques that can easily lead to feasible points where the objective function ***f*** is no longer defined (resulting in indefinite matrices for ML estimation) or is difficult to compute (resulting in floating point overflows). Therefore, PROC CALIS provides options that restrict the step length or trust region radius, especially during the first main iterations.

The inner product ***g's*** of the gradient ***g*** and the search direction ***s*** is the slope of $f(\alpha) = f(x + \alpha s)$ along the search direction ***s*** with step length α. The default starting value $\alpha^{(0)} = \alpha^{(k,0)}$ in each line-search algorithm ($\min_{\alpha>0} f(x + \alpha s)$)during the main iteration ***k*** is computed in three steps.

1. Use either the difference ***df*=|*f*(k) - *f*(k-1)|** of the function values during the last two consecutive iterations or the final stepsize value α^- of the previous iteration ***k*-1** to compute a first value $\alpha_1^{(0)}$.
   - Using the DAMPSTEP<=***r***> option:

$$\alpha_1^{(0)} = \min(1, r\alpha^-)$$

The initial value for the new step length can be no larger than ***r*** times the final step length α^- of the previous iteration. The default is ***r*=2**.

   - Not using the DAMPSTEP option:

$$\alpha_1^{(0)} = \begin{cases} step & \text{if } 0.1 \le step \le 10 \\ 10 & \text{if } step > 10 \\ 0.1 & \text{if } step < 0.1 \end{cases}$$

with

$$step = \begin{cases} df/|g's| & \textbf{if } |g's| \geq \epsilon \max(100df, 1) \\ 1 & \textbf{otherwise} \end{cases}$$

This value of $\alpha_1^{(0)}$ can be too large and can lead to a difficult or impossible function evaluation, especially for highly nonlinear functions such as the EXP function.

2. During the first five iterations, the second step enables you to reduce $\alpha_1^{(0)}$ to a smaller starting value $\alpha_2^{(0)}$ using the INSTEP=***r*** option:

$$\alpha_2^{(0)} = \min(\alpha_1^{(0)}, r)$$

After more than five iterations, $\alpha_2^{(0)}$ is set to $\alpha_1^{(0)}$.

3. The third step can further reduce the step length by

$$\alpha_3^{(0)} = \min(\alpha_2^{(0)}, \min(10, u))$$

where ***u*** is the maximum length of a step inside the feasible region.

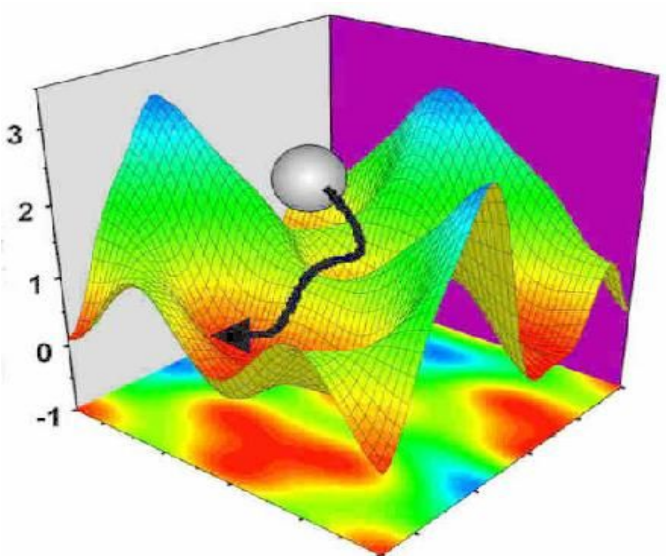
The INSTEP=***r*** option lets you specify a smaller or larger radius of the trust region used in the first iteration by the trust-region, double-dogleg, and Levenberg-Marquardt algorithm. The default initial trust region radius is the length of the scaled gradient (Moré 1978). This step corresponds to the default radius factor of ***r*=1**. This choice is successful in most practical applications of the TRUREG, DBLDOG, and LEVMAR algorithms. However, for bad initial values used in the analysis of a covariance matrix with high variances, or for highly nonlinear constraints (such as using the EXP function) in your programming code, the default start radius can result in arithmetic

overflows. If this happens, you can try decreasing values of INSTEP=***r***, **0 < *r* < 1**, until the iteration starts successfully. A small factor ***r*** also affects the trust region radius of the next steps because the radius is changed in each iteration by a factor $0 < c \leq 4$ depending on the ρ ratio. Reducing the radius corresponds to increasing the ridge parameter λ that produces smaller steps directed closer toward the gradient direction.

2.3.2.8 Techniques of global optimization

Global Optimisation (**GO**) problems involving a given cost function arise in many areas of technology, industry, as well as in natural, economical and social sciences, having received enormous attention in recent years, mostly due to the enormous increase in computational power which has taken place in the last decade.

The figure below provides a trivial two-dimensional example of the optimization paradigm, where one wants to find the value of the coordinates (x,y) for which C(x,y) has its absolute minimum - notice that the figure shows only a portion of the total domain of C(x,y); the sphere symbolizes a molecule the energy of which changes as the coordinates (x,y) change.



As the number of independent variables of the cost function increases, the number of minima tends to increase **exponentially**. As such, global optimization problems are typical examples of NP-complete problems, that is, problems considered generally intractable.

For a large number of independent variables, and for the most general forms of the cost-function, Simulated Annealing (**SA**) constitutes one of the simplest yet powerful methods to perform **GO**. However, it is well known that **SA** requires many evaluations of

the cost-function in order to provide reliable results, a feature which becomes prohibitively expensive whenever the computation of the cost-function is itself expensive, as happens too frequently.

In this project new approaches are studied and developed along two main lines of research:

- methods that do not require computation of the derivatives of the cost-function, such as Jump Annealing (**JA**) - which constitutes a very efficient improvement of the original **SA** - as well as more involved techniques, such as Genetic Algorithms, Memetic Algoithms, Differential Evolution, etc.
- methods which make explicit use of the multi-dimensional gradients of the cost-function, and which ***i)*** either do not involve modifications of the cost function, such as modified conjugate-gradient techniques, ***ii)*** or which map the cost-function in more convenient topologically derived functions, such as TRUST.

Summary:

In typical engineering processes, often there are complicated constraints and the objective function is not easy to calculate. This makes the selection and systematic application of suitable optimization techniques a critical task, which is the thrust of our research in this area.

Sometimes the evaluation of the objective function in practical engineering problems may require a significant amount of computational time and resources. Building an approximate response surface for the objective function and using this response surface for optimization is an efficient alternative.

Model Questions:

1. What is optimization and its uses?
2. Briefly explain the various techniques of optimization?

References:

1. Basic Optimization Techniques By Jim Hedger
2. Homotopy Optimization Methods and Protein Structure Prediction by Daniel M. Dunlavy

AUTHOR:

B.M.REDDY M.Tech. (HBTI, Kanpur)

Lecturer, Centre for Biotechnology

Acharya Nagarjuna University.

Lessons 2.3.3

CONSTRUCTION OF MODELS FOR THE REAL PHYSICAL PROCESSES

Objective

2.3.3.1 Introduction

2.3.3.2 Typical Tasks in the Development Process Life Cycle

2.3.3.3 Ad-hoc Development

2.3.3.4 The Waterfall Model

2.2.3.5 Iterative Development

2.3.3.6 Prototyping

2.2.3.7 The Exploratory Model

2.3.3.8 The Spiral Model

2.3.3.9 The Reuse Model

2.3.3.10 Creating and Combining Models

Summary
Model Questions
References

Objective:

This topic provides an overview of the more common system development *Process Model*s, used to guide the analysis, design, development, maintenance of information systems, and errors (problems) involved in them.

2.3.3.1 Introduction

This topic provides an overview of the more common system development *Process Model*s, used to guide the analysis, design, development, and maintenance of information systems. There are many different methods and techniques used to direct the life cycle of a software development project and most real-world models are customized adaptations of the generic models. While each is designed for a specific purpose or reason, most have similar goals and share many common tasks.

2.3.3.2 Typical Tasks in the Development Process Life Cycle

Professional system developers and the customers they serve share a common goal of building information systems that effectively support business process objectives. In order to ensure that cost-effective, quality systems are developed which address an organization's business needs, developers employ some kind of system development

Process Model to direct the project's life cycle. Typical activities performed include the following:

- System conceptualization
- System requirements and benefits analysis
- Project adoption and project scoping
- System design
- Specification of software requirements
- Architectural design
- Detailed design
- Unit development
- Software integration & testing
- System integration & testing
- Installation at site
- Site testing and acceptance
- Training and documentation
- Implementation
- Maintenance

Process Model/Life-Cycle Variations

While nearly all system development efforts engage in some combination of the above tasks, they can be differentiated by the *feedback* and *control methods* employed during development and the *timing of activities*. Most system development *Process Models* in use today have evolved from three primary approaches: *Ad-hoc Development*, *Waterfall Model*, and the *Iterative* process.

2.3.3.3 Ad-hoc Development

Early systems development often took place in a rather chaotic and haphazard manner, relying entirely on the skills and experience of the individual staff members performing the work. Today, many organizations still practice *Ad-hoc Development* either entirely or for a certain subset of their development (e.g. small projects). The Software Engineering Institute at Carnegie Mellon University2 points out that with *Ad-hoc*

Process Models, "process capability is unpredictable because the software process is constantly changed or modified as the work progresses. Schedules, budgets, functionality, and product quality are generally (inconsistent). Performance depends on the capabilities of individuals and varies with their innate skills, knowledge, and motivations. There are few stable software processes in evidence, and performance can be predicted only by individual rather than organizational capability." 3

Figure 1. Ad-hoc Development

"Even in undisciplined organizations, however, some individual software projects produce excellent results. When such projects succeed, it is generally through the heroic efforts of a dedicated team, rather than through repeating the proven methods of an organization with a mature software process. In the absence of an organization-wide software process, repeating results depends entirely on having the same individuals available for the next project. Success that rests solely on the availability of specific individuals provides no basis for long-term productivity and quality improvement throughout an organization."4

2.3.3.4 The Waterfall Model

The *Waterfall Model* is the earliest method of structured system development. Although it has come under attack in recent years for being too rigid and unrealistic when it comes to quickly meeting customer's needs, the *Waterfall Model* is still widely used. It is attributed with providing the theoretical basis for other *Process Models*, because it most closely resembles a "generic" model for software development.

Feedback & Error Correction

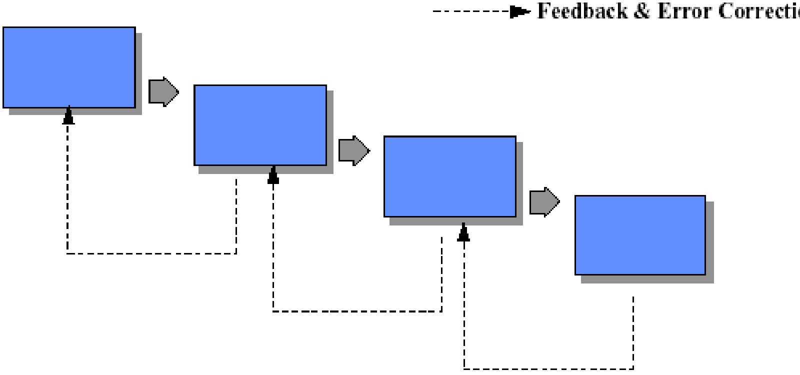


Figure 2. Waterfall Model

The *Waterfall Model* consists of the following steps:

System Conceptualization. System Conceptualization refers to the consideration of all aspects of the targeted business function or process, with the goals of determining

how each of those aspects relates with one another, and which aspects will be incorporated into the system.

Systems Analysis. This step refers to the gathering of system requirements, with the goal of determining how these requirements will be accommodated in the system. Extensive communication between the customer and the developer is essential.

System Design. Once the requirements have been collected and analyzed, it is necessary to identify in detail how the system will be constructed to perform necessary tasks. More specifically, the System Design phase is focused on the data requirements (what information will be processed in the system?), the software construction (how will the application be constructed?), and the interface construction (what will the system look like? What standards will be followed?).

Coding. Also known as programming, this step involves the creation of the system software. Requirements and systems specifications from the System Design step are translated into machine readable computer code.

Testing. As the software is created and add d to the developing system, testing is performed to ensure that it is working correctly and efficiently. Testing is generally focused on two areas: internal efficiency and external effectiveness. The goal of external effectiveness testing is to verify that the software is functioning according to system design, and that it is performing all necessary functions or sub-functions. The goal of internal testing is to make sure that the computer code is efficient, standardized, and well documented. Testing can be a labor-intensive process, due to its iterative nature.

2.2.3.5 Iterative Development

The problems with the *Waterfall Model* created a demand for a new method of developing systems which could provide faster results, require less up-front information, and offer greater flexibility. With *Iterative Development*, the project is divided into small parts. This allows the development team to demonstrate results earlier on in the process and obtain valuable feedback from system users. Often, each iteration is actually a mini-*Waterfall* process with the feedback from one phase providing vital information for the design of the next phase. In a variation of this model, the software products which are produced at the end of each step (or series of steps) can go into production immediately as incremental releases.

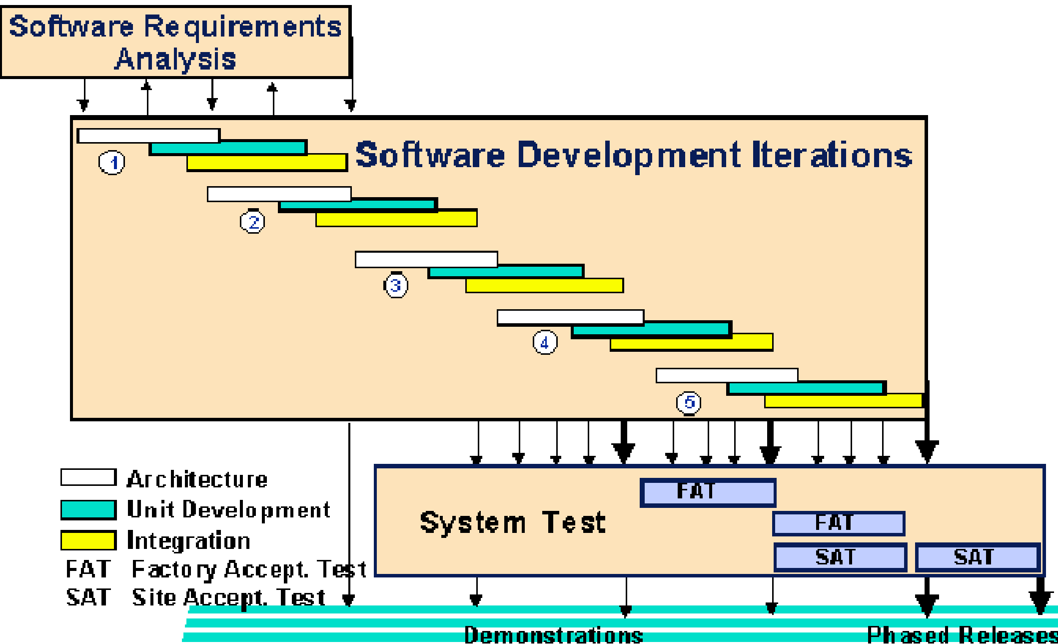


Figure 3. Iterative Development

Variations on Iterative Development

A number of *Process Models* have evolved from the *Iterative* approach. All of these methods produce some demonstrable software product early on in the process in order to obtain valuable feedback from system users or other members of the project team. Several of these methods are described below.

2.3.3.6 Prototyping

The *Prototyping Model* was developed on the assumption that it is often difficult to know all of your requirements at the beginning of a project. Typically, users know many of the objectives that they wish to address with a system, but they do not know all the nuances of the data, nor do they know the details of the system features and capabilities. The *Prototyping Model* allows for these conditions, and offers a development approach that yields results without first requiring all information up-front . When using the *Prototyping Model*, the developer builds a simplified version of the proposed system and presents it to the customer for consideration as part of the development process. The customer in turn provides feedback to the developer, who goes back to refine the system requirements to incorporate the additional information. Often, the prototype code is thrown away and entirely new programs are developed once requirements are identified. There are a few different approaches that may be followed when using the *Prototyping Model*:

creation of the major user interfaces without any substantive coding in the background in order to give the users a “feel” for what the system will look like,

- development of an abbreviated version of the system that performs a limited subset of functions; development of a paper system (depicting proposed screens, reports, relationships etc.), or

- use of an existing system or system components to demonstrate some functions that will be included in the developed system.

Prototyping is comprised of the following steps:

- **Requirements Definition/Collection.** Similar to the Conceptualization phase of the *Waterfall Model*, but not as comprehensive. The information collected is usually limited to a subset of the complete system requirements.

- **Design.** Once the initial layer of requirements information is collected, or new information is gathered, it is rapidly integrated into a new or existing design so that it may be folded into the prototype.

- **Prototype Creation/Modification.** The information from the design is rapidly rolled into a prototype. This may mean the creation/modification of paper information, new coding, or modifications to existing coding.

- **Assessment.** The prototype is presented to the customer for review. Comments and suggestions are collected from the customer.

- **Prototype Refinement.** Information collected from the customer is digested and the prototype is refined. The developer revises the prototype to make it more effective and efficient.

- **System Implementation.** In most cases, the system is rewritten once requirements are understood. Sometimes, the *Iterative* process eventually produces a working system that can be the cornserstone for the fully functional system.

Variation of the Prototyping Model

A popular variation of the *Prototyping Model* is called **Rapid Application Development (RAD)**.

RAD introduces strict time limits on each development phase and relies heavily on rapid application tools which allow for quick development.

2.2.3.7 The Exploratory Model

In some situations it is very difficult, if not impossible, to identify any of the requirements for a system at the beginning of the project. Theoretical areas such as Artificial Intelligence are candidates for using the *Exploratory Model*, because much of the research in these areas is based on guess-work, estimation, and hypothesis. In these cases, an assumption is made as to how the system might work and then rapid iterations are used to quickly incorporate suggested changes and build a usable system. A distinguishing characteristic of the *Exploratory Model* is the absence of precise specifications. Validation is based on adequacy of the end result and not on its adherence to pre-conceived requirements.The *Exploratory Model* is extremely simple in its construction; it is composed of the following steps:

Initial Specification Development. Using whatever information is immediately available, a brief System Specification is created to provide a rudimentary starting point.

System Construction/Modification. A system is created and/or modified according to whatever information is available.

System Test. The system is tested to see what it does, what can be learned from it, and how it may be improved.

System Implementation. After many iterations of the previous two steps produce satisfactory results, the system is dubbed as "finished" and implemented.

2.3.3.8 The Spiral Model

The *Spiral Model* was designed to include the best features from the *Waterfall* and *Prototyping Models*, and introduces a new component - risk-assessment. The term "spiral" is used to describe the process that is followed as the development of the system takes place. Similar to the *Prototyping Model*, an initial version of the system is developed, and then repetitively modified based on input received from customer evaluations. Unlike the *Prototyping Model*, however, the development of each version of the system is carefully designed using the steps involved in the

Waterfall Model. With each iteration around the spiral (beginning at the center and working outward), progressively more complete versions of the system are built.

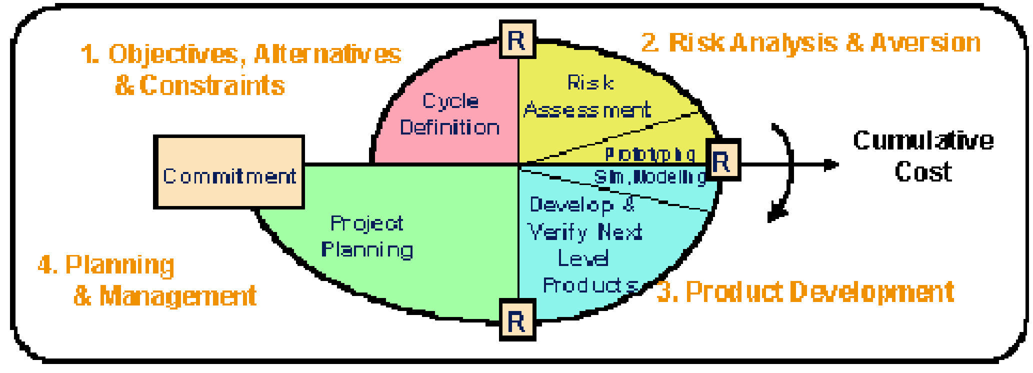


Figure 4. Spiral Model

Risk assessment is included as a step in the development process as a means of evaluating each version of the system to determine whether or not development should continue. If the customer decides that any identified risks are too great, the project may be halted. For example, if a substantial increase in cost or project completion time is identified during one phase of risk assessment, the customer or the developer may decide that it does not make sense to continue with the project, since the increased cost or lengthened timeframe may make continuation of the project impractical or unfeasible.

The *Spiral Model* is made up of the following steps:

- **Project Objectives.** Similar to the system conception phase of the *Waterfall Model*. Objectives are determined, possible obstacles are identified and alternative approaches are weighed.
- **Risk Assessment.** Possible alternatives are examined by the developer, and associated risks/problems are identified. Resolutions of the risks are evaluated and weighed in the consideration of project continuation. Sometimes prototyping is used to clarify needs.
- **Engineering & Production.** Detailed requirements are determined and the software piece is developed.
- **Planning and Management.** The customer is given an opportunity to analyze the results of the version created in the Engineering step and to offer feedback to the developer.

2.3.3.9 The Reuse Model

The basic premise behind the *Reuse Model* is that systems should be built using existing components, as opposed to custom-building new components. The *Reuse Model* is clearly suited to Object-Oriented computing environments, which have become one of the premiere technologies in today's system development industry. Within the *Reuse Model*, libraries of software modules are maintained that can be copied for use in any system. These components are of two types: procedural modules and database modules. When building a new system, the developer will "borrow" a copy of a module from the systemlibrary and then plug it into a function or procedure. If the needed module is not available, the developer will build it, and store a copy in the system library for future usage. If the modules are well engineered, the developer with minimal changes can implement them.

The *Reuse Model* consists of the following steps:

- **Definition of Requirements.** Initial system requirements are collected. These requirements are usually a subset of complete system requirements.
- **Definition of Objects.** The objects, which can support the necessary system components, are identified.
- **Collection of Objects.** The system libraries are scanned to determine whether or not the needed objects are available. Copies of the needed objects are downloaded from the system.
- **Creation of Customized Objects.** Objects that have been identified as needed, but that are not available in the library are created.
- **Prototype Assembly.** A prototype version of the system is created and/or modified using the necessary objects.
- **Prototype Evaluation.** The prototype is evaluated to determine if it adequately addresses customer needs and requirements.
- **Requirements Refinement.** Requirements are further refined as a more detailed version of the prototype is created.

- **Objects Refinement.** Objects are refined to reflect the changes in the requirements.

2.3.3.10 Creating and Combining Models

In many cases, parts and procedures from various *Process Models* are integrated to support system development. This occurs because most models were designed to provide a framework for achieving success only under a certain set of circumstances. When the circumstances change beyond the limits of the model, the results from using it are no longer predictable. When this situation occurs it is sometimes necessary to alter the existing model to accommodate the change in circumstances, or adopt or combine different models to accommodate the new circumstances.

The selection of an appropriate *Process Model* hinges primarily on two factors: organizational environment and the nature of the application. Frank Land, from the London School of Economics, suggests that suitable approaches to system analysis, design, development, and implementation be based on the relationship between the information system and its organizational environment.8 Four categories of relationships are identified:

- **The Unchanging Environment.** Information requirements are unchanging for the lifetime of the system (e.g. those depending on scientific algorithms). Requirements can be stated unambiguously and comprehensively. A high degree of accuracy is essential. In this environment, formal methods (such as the *Waterfall* or *Spiral Models*) would provide the completeness and precision required by the system.

- **The Turbulent Environment.** The organization is undergoing constant change and system requirements are always changing. A system developed on the basis of the conventional *Waterfall Model* would be, in part; already obsolete by the time it is implemented. Many business systems fall into this category. Successful methods would include those, which incorporate rapid development, some throwaway code (such as in *Prototyping*), the maximum use of reusable code, and a highly modular design.

- **The Uncertain Environment.** The requirements of the system are unknown or uncertain. It is not possible to define requirements accurately ahead of time because the situation is new or the system being employed is highly innovative. Here, the development methods must emphasize learning. Experimental *Process Model*s, which take advantage of prototyping and rapid development, are most appropriate.

- **The Adaptive Environment.** The environment may change in reaction to the system being developed, thus initiating a changed set of requirements. Teaching systems and expert systems fall into this category. For these systems, adaptation is key, and the methodology must allow for a straightforward introduction of new rules.

Summary

The evolution of system development *Process Model*s has reflected the changing needs of computer customers. As customers demanded faster results, more involvement in the development process, and the inclusion of measures to determine risks and effectiveness, the methods for developing systems changed. In addition, the software and hardware tools used in the industry changed (and continue to change) substantially. Faster networks and hardware supported the use of smarter and faster

operating systems that paved the way for new languages and databases, and applications that were far more powerful than any predecessors. These rapid and numerous changes in the system development environment simultaneously spawned the development of more practical new *Process Model*s and the demise of older models that were no longer useful.

Model Questions:

1. Explain various physical construction models and the problems facing in them?

References:

1. G Walsham "Interpreting Information Systems in Organizations", John Wiley, Chichester.
2. Automatic Construction of Accurate Models of Physical Systems, by Elizabeth Bradley.

AUTHOR:

B. M. REDDY **M.Tech. (HBTI, Kanpur)**

Lecturer, Centre for Biotechnology

Acharya Nagarjuna University.

Lesson 2.3.4

ERRORS INVOLVED IN THE CONSTRUCTION OF MODELS FOR REAL PHYSICAL PROCESSES

Objective
2.3.4.1 The Waterfall Model
2.3.4.2 Iterative Development
2.3.4.3 Prototyping
2.3.4.4 The Exploratory Model
2.3.4.5 The Spiral Model
2.3.4.6 The Reuse Model
2.3.4.7 Automatic Construction of Accurate Models of Physical Systems

Summary
Model Questions
References

Objective:

In the previous lesson we have learned about the construction of models. The present lesson discusses the problems (errors) associated with those construction models.

2.3.4.1 The Waterfall Model

Problems/Challenges associated with the Waterfall Model

Although the *Waterfall Model* has been used extensively over the years in the production of many quality systems, it is not without its problems. In recent years it has come under attack, due to its rigid design and inflexible procedure. Criticisms fall into the following categories:

- Real projects rarely follow the sequential flow that the model proposes.

- At the beginning of most projects there is often a great deal of uncertainty about requirements and goals, and it is therefore difficult for customers to identify these criteria on a detailed level. The model does not accommodate this natural uncertainty very well.

- Developing a system using the *Waterfall Model* can be a long, painstaking process that does not yield a working version of the system until late in the process.

2.3.4.2 Iterative Development

Problems/Challenges associated with the Iterative Model

While the *Iterative Model* addresses many of the problems associated with the *Waterfall Model*, it does present new challenges.

The user community needs to be actively involved throughout the project. While this involvement is a positive for the project, it is demanding on the time of the staff nd can add project delay.

Communication and coordination skills take center stage in project development.

Informal requests for improvement after each phase may lead to confusion -- a controlled mechanism for handling substantive requests needs to be developed.

The *Iterative Model* can lead to “scope creep,” since user feedback following each phase may lead to increased customer demands. As users see the system develop, they may realize the potential of other system capabilities which would enhance their work.

2.3.4.3 Prototyping

Problems/Challenges associated with the Prototyping Model

Criticisms of the *Prototyping Model* generally fall into the following categories:

Prototyping can lead to false expectations. *Prototyping* often creates a situation where the customer mistakenly believes that the system is “finished” when in fact it is not. More specifically, when using the *Prototyping Model*, the pre-implementation versions of a system are really nothing more than one-dimensional structures. The necessary, behindthe- scenes work such as database normalization, documentation, testing, and reviews for efficiency have not been done. Thus the necessary underpinnings for the system are not in place.

Prototyping can lead to poorly designed systems. Because the primary goal of *Prototyping* is rapid development, the design of the system can sometimes suffer because the system is built in a series of “layers” without a global consideration of the integration of all other components. While initial software development is often built to be a “throwaway, ” attempting to retroactively produce a solid system design can sometimes be problematic.

2.3.4.4 The Exploratory Model

Problems/Challenges associated with the Exploratory Model

There are numerous criticisms of the *Exploratory Model*:

- It is limited to use with very high-level languages that allow for rapid development, such as LISP.

- It is difficult to measure or predict its cost-effectiveness.

- As with the *Prototyping Model*, the use of the *Exploratory Model* often yields inefficient or crudely designed systems, since no forethought is given as to how to produce a streamlined system.

2.3.4.5 The Spiral Model

Problems/Challenges associated with the Spiral Model

Due to the relative newness of the *Spiral Model*, it is difficult to assess its strengths and weaknesses. However, the risk assessment component of the *Spiral Model* provides both developers and customers with a measuring tool that earlier *Process Model*s do not have. The measurement of risk is a feature that occurs everyday in real-life situations, but (unfortunately) not as often in the system development industry. The practical nature of this tool helps to make the *Spiral Model* a more realistic *Process Model* than some of its predecessors.

2.3.4.6 The Reuse Model

Problems/Challenges Associated with the Reuse Model

A general criticism of the *Reuse Model* is that it is limited for use in object-oriented development environments. Although this environment is rapidly growing in popularity, it is currently used in only a minority of system development applications.

2.3.4.7 Automatic Construction of Accurate Models of Physical Systems

Traditional system identification addresses the task of inferring a mathematical model of a system from observations of that system. A controls engineer might perform this task, in its most basic form, by choosing a power series and matching its coefficients against the numerical observations via some sort of regression. This topic describes a computer program called pret that automates the system identi_cation process, at several levels, by building an artificial intelligence (AI) layer on top of a set of traditional system identification techniques. This AI layer automates the high-level stages of the identification process that are normally performed by a human expert. Qualitative, symbolic, and geometric reasoning are used to perform structural identification in the choice of the power series made by the engineer in the example above. This layer also automates another subtle and difficult part of the process: the choice and application of the appropriate lower-level method for each stage of the process. pret works with ordinary differential equation (ODE) models, linear or nonlinear, in one variable or many. Its implementation is a hybridization of traditional numerical analysis methods, such as simulation and nonlinear

regression, with logic programming, computer vision techniques, and qualitative reasoning. The input consists of specific information about an individual system, in three forms:

- the user's hypotheses about the physics involved
- observations, interpreted and described by the user, symbolically or graphically, in varying formats and degrees of precision
- physical measurements made directly and automatically on the system

To construct an ODE model from this information, pret combines powerful mathematical formalisms, such as the link between the divergence of an ODE and the friction of the system that it describes, with domain-specific notions such as force balances in mechanical systems to allow the types of custom-generated approximations" that are lacking in existing AI modeling programs. Two sets of rules, both of which may easily be changed or augmented by the user, play very different roles in the model-building task. Domain-specific rules are used to combine hypotheses into models a nontrivial task in a system with more than one degree of freedom, or a system in which physical effects couple to one another while general rules about ODE properties are used by a custom deduction engine to infer facts from models and from observations. Both model- and observation-based inferences are governed by specifications, which prescribe the resolution for quantities of interest. Any contradictions between the set of facts inferred from the observations and the set of facts inferred from a candidate model cause that model to be ruled out, in which case prêt tries a new combination of hypotheses. The first noncontradictory model in this sequence is returned as the answer.

Acting upon simple mechanical examples like the parametrically driven pendulum, the current version of the program can efficiently perform these tasks and construct accurate ODE models. Of course, a model of a driven pendulum is not the research goal here physicists and engineers have spent centuries constructing and refining such models. This is simply an example one that was chosen because it is instantly recognizable and intuitively obvious to the reader. In spite of its simplicity, this example is interesting from an AI standpoint, as it demonstrates effective automated reasoning, even if only on the level performed by a 17th-century physicist. One of the ultimate goals of this research is to produce a tool that can construct a model of a black-box system using only information from its ports. Textbook examples like the pendulum are critical to such an endeavor, as one must, for obvious reasons, verify the tool's performance on such exercises before trusting the results that it produces when presented with difficult open problems. The following subsection is a brief review of AI and AI-specific perspectives on modeling research, coupled with a description of where this work fits within that context. The next section presents an overview of pret's function, illustrates its input syntax using a simple example, and discusses some of the more important implications of that syntax. Section 3 outlines how the program uses that information, together with its encoded knowledge, to build an ODE model and closes with some discussion of related work, both in AI and other

fields. The final section wraps up the example, gives a status report, discusses future directions, and summarizes some of the most important issues of the research.

Artificial Intelligence and Modeling

Research in AI has two major goals: the understanding of the mechanisms that make human intelligence possible and the construction of intelligent artifacts. Both ends of this spectrum analytic and synthetic AI depend on each other: a theory of human intelligence may be verified by the intelligent behavior of an artifact that instantiates the theory. Conversely, an engineer who builds an intelligent system may obtain useful ideas by observing human experts. Intelligent behavior requires that the world knowledge that is relevant to the task at hand be available, along with the means to reason about it. The construction of an intelligent computer program requires a framework in which both knowledge and reasoning can be formalized. This formal system should be small and neat enough to be understandable and easy to maintain, and yet powerful enough to allow its users to think and formulate in the language and concepts of the application domain. An adequate representation formalism allows a natural formulation of the problem that is to be solved. Therefore, AI programmers typically try to represent knowledge declaratively rather than operationally: one formulates facts rather than instructions. Ideally, the declarative representation of the problem is executable. That means, it is not just part of the intelligent artifact but it is the artifact.

Modeling physical systems is an ideal application for these ideas and techniques. One of the most powerful analysis tools in existence| and often one of the most difficult to create is a good model. Expert model-builders typically construct hierarchies of successively subtler representations that capture the salient features of a physical system, each incorporating more physics than the last. At each level in the hierarchy, the modeler assesses what properties and perspectives are important and uses approximations and abstractions to focus the model accordingly. The subtlety of the reasoning skills involved in this process, together with the intricacy of the interplay between them, has led many of its practitioners to classify modeling as \intuitive" and \an art." Any tool that effectively automated a coherent and useful part of this art would be of obvious practical importance in science and engineering: as a corroborator of existing models and designs, as a medium within which to instruct newcomers, and as an intelligent assistant, whose aid allows more time and creative thought to be devoted to other demanding tasks. The computer program pret described in this paper is exactly such an automatic modeler. This work falls on the \intelligent artifact" end of the AI spectrum | its focus is not to construct a cognitive model of the thought process of a physicist or an engineer when he or she builds a model of a physical system, but rather to build a useful tool that obtains the same result as a human expert would. However, as outlined above, learning from a physicist's techniques is a fruitful approach. pret's techniques fall mostly in the category of qualitative physics (QP) or qualitative reasoning (QR) Like AI in general, qualitative reasoning about physics spans a whole spectrum, from modeling how humans reason about their physical environment to engineering artifacts that can reason about physics. Its main goals are the prediction of

behavior, analysis, design, control, monitoring, and fault diagnosis. Many QR programs, particularly the ones that perform monitoring or diagnosis, infer the behavior of a physical system from its structure or vice versa. What distinguishes QP from other formalisms that represent physics knowledge, such as differential equations, is the abstraction to a qualitative level. For example, so-called landmarks divide the continuum of real numbers into a finite number of intervals. Typically, landmarks are critical values of quantities that describe the physical system. The behavior of the physical system | the progression of the values of relevant quantities | is described as a discrete sequence of states and state transitions. States describe situations, such as \x = 0" or \y is positive" or \z = l1" where l1 is a landmark. State transitions describe changing values, e.g., \x is monotonically increasing." Many well-developed formalisms to represent and reason about mathematical, quantitative, and numerical knowledge exist. The goal of QP, however, is to formalize and automate conceptual, abstract, and qualitative reasoning in the physics domain. This qualitative kind of knowledge and reasoning requires a completely different set of primitives, such as the states and state transitions described above, or the facts about ordinary differential equations that are pret's primitives. Adequate combinations of well-chosen primitives are a primary research goal in problems like this; as they enable a computer program to handle unforeseen situations. The ability to rearrange thought primitives in order to solve new problems is a crucial part of what makes a good scientist or engineer | or an intelligent artifact that performs the same tasks. In general, modeling underlies most of the approaches to reasoning about physical systems. Strictly speaking, every formalization of the properties of a physical system constitutes a model of the physical system. The spectrum ranges from models that use a language that is very close to the physics of the system to models that use a language that is well-suited to describe the system mathematically. An example of the physics end of this spectrum might be formal instructions how to build a pendulum. These instructions would use terms like rod, bob, and bearing. In any case, a modeler human or not builds the model out of simple components, assuming that the overall behavior follows from the behavior of the components and their interaction. Examples of QR modeling systems include the ENVISION system, which reasons about the components of the system and their interaction, as well as QPT (Qualitative Process Theory) and its successor QPE (Qualitative Process Engine), which emphasize the notion of causality. This is a very useful approach if the goal of the program is to explain certain phenomena in physical terms. QSIM (Qualitative SIMulation) simulates the behavior of a physical system qualitatively. The description of the physical system, called a qualitative differential equation (QDE), uses mathematical language rather than terms from physics, namely the progression of relevant quantities (functions) and constraints on relations between these functions. The following example, drawn directly from a recent and thorough text on this topic, is a QDE fragment that relates the amounts of fluids in two connected containers, A and B, and the flow rates in the pipe between them, while constraining the total amount of fluid to remain constant:

((minus flowAB -flowAB))

((d/dt amtB flowAB))

((d/dt amtA -flowAB))

((add amtA amtB total))

((constant total))

From this information and some initial conditions2, QSIM generates qualitative descriptions of every possible outcome, which it presents on graphs whose breakpoints are the landmarks described above. One current thrust of research by this group targets the integration of quantitative and qualitative information. Some other useful QR references are: on varying resolution, on order-of-magnitude reasoning, and on mathematical aspects. A few useful general AI references are . On the spectrum from physics language to mathematics language, QPT resides on the physics end, QSIM on the mathematics end, and pret somewhere in between. Its inputs primarily observations and hypotheses about the physical system are partially in the terms of physics. This approach allows the user to state the problem in his or her domain language. Also, pret's reasoning uses concepts from physics, allowing it to rule out bad candidate models by high-level abstract reasoning. The rules about how hypotheses are combined into ODEs reflect laws of physics, such as F = ma. The decision about what to try next if some candidate model fails will also use physics concepts, e.g, \try quadratic friction instead of linear friction." However, pret's output | the model of the physical system that it constructs | is purely mathematical: an ODE.

In summary, pret makes heavy use of QR's notions of landmarks, qualitative vocabulary, and qualitative behavior. As do most QR systems, it exploits symbolic reasoning and reason-maintenance techniques. The structure of the physical system is described by notions like point, loop, etc., but there is no elaborate scenario description (e.g., a description that uses language that is highly specific to, say, fluids in containers). The program does not reason about causal relationships between behaviors of physical system components, nor does it try to explain observed phenomena. pret takes a minimalist approach: find a simple model that is consistent with the observed behavior of the physical system.

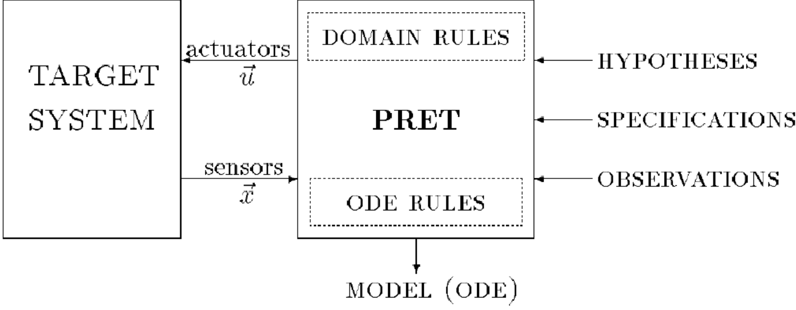


Fig1: Structure and application.

Summary

The evolution of system development *Process Model*s has reflected the changing needs of computer customers. As customers demanded faster results, more involvement in the development process, and the inclusion of measures to determine risks and effectiveness, the methods for developing systems changed. In addition, the software and hardware tools used in the industry changed (and continue to change) substantially. Faster networks and hardware supported the use of smarter and faster operating systems that paved the way for new languages and databases, and applications that were far more powerful than any predecessors. These rapid and numerous changes in the system development environment simultaneously spawned the development of more practical new *Process Model*s and the demise of older models that were no longer useful.

Model Questions:

1. Explain various physical construction models and the problems facing in them?

References:

1. G Walsham "Interpreting Information Systems in Organizations", John Wiley, Chichester.

2. Automatic Construction of Accurate Models of Physical Systems, by Elizabeth Bradley.

AUTHOR:

B.M.REDDY **M.Tech. (HBTI, Kanpur)**

Lecturer, Centre for Biotechnology

Acharya Nagarjuna University.

Lesson 2.4.1

MINIMISATION AND MAXIMISATION OF FUNCTIONS

Objective

2.4.1.1 Introduction

2.4.1.2 Maxima and Minima

2.4.1.3 Maxima and Minima In brief

2.4.1.4 Examples

2.4.1.5 Maxima and Minima in a Bounded Region

2.4.1.6 Maxima and Minima in a Disk

2.4.1.7 Functions of More than 2 Variables

Summary
Model Questions
References

Objective:

This lesson tells about the minimization and maximization of functions in different fields along with examples.

2.4.1.1 Introduction

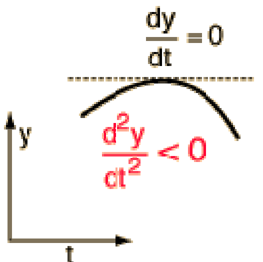
The problem of determining the maximum or minimum of function is encountered in geometry, mechanics, physics, and other fields, and was one of the motivating factors in the development of the calculus in the seventeenth century. One of the great powers of calculus is in the determination of the maximum or minimum value of a function. Take f(x) to be a function of x. Then the value of x for which the derivative of f(x) with respect to x is equal to zero corresponds to a maximum, a minimum or an inflexion point of the function f(x).

2.4.1.2 Maxima and Minima

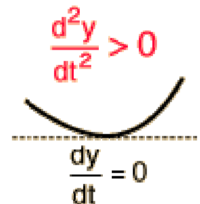
For example, the height of a projectile that is fired straight up is given by the motion equation:

The derivative of a function can be geometrically interpreted as the slope of the curve of the mathematical function y(t) plotted as a function of t. The derivative is positive when a function is increasing toward a maximum, zero (horizontal) at the maximum, and negative just after the maximum. The second derivative is the rate of change of the derivative, and it is negative for the process described above since the first derivative (slope) is always getting smaller. The second derivative is always negative for a "hump" in the function, corresponding to a maximum

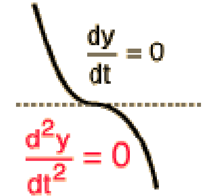
The second derivative demonstrates whether a point with zero first derivative is a maximum, a minimum, or an inflexion point.



For a maximum, the second derivative is negative. The slope of the curve (first derivative) is at first positive, then goes through zero to become negative.

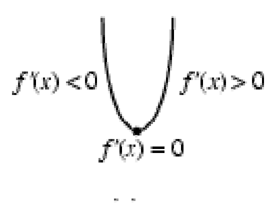


For a minimum, the second derivative is positive. The slope of the curve = first derivative is at first negative, then goes through zero to become positive.



For an inflexion point, the second derivative is zero at the same time the first derivative is zero. It represents a point where the curvature is changing its sense. Inflexion points are relatively rare in nature.

2.4.1.3 Maxima and Minima In brief

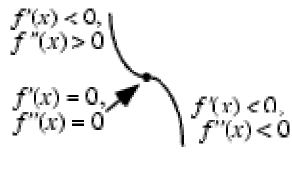


minimum

$f'(x) = 0$

$f'(x) > 0$ $f'(x) < 0$

maximum



stationary point

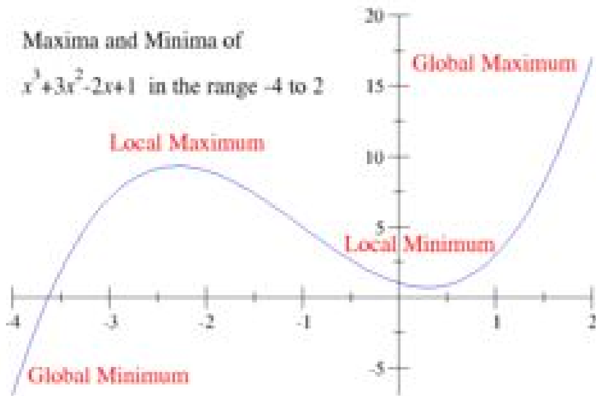
A continuous function may assume a minimum at a single point or may have minima at a number of points. A global minimum of a function is the smallest value in the entire range of the function, while a local minimum is the smallest value in some local neighborhood.

For a function $f(x)$which is continuous at a point x_0, a necessary but not sufficient condition for $f(x)$to have a relative minimum at $x = x_0$ is that x_0 be a critical point (i.e., $f(x)$is either not differentiable at x_0 or x_0 is a stationary point, in which case $f'(x_0) = 0$).

The first derivative test can be applied to continuous functions to distinguish minima from maxima. For twice differentiable functions of one variable, $f(x)$, or of two variables, $f(x, y)$, the second derivative test can sometimes also identify the nature of an extremum. For a function $f(x)$, the extremum test succeeds under more general conditions than the second derivative test.

As in one variable calculations, one use for derivatives in several variables is in calculating maxima and minima. Again as for one variable, we shall rely on the theorem that if f is continuous on a closed bounded subset of $\mathbb{R}^2$, then it has a global maximum and a global minimum. And again as before, we note that these must occur *either* at a local maximum or minimum, or else on the boundary of the region. Of course in $\mathbb{R}$, the boundary of the region usually consisted of a pair of end points, while in $\mathbb{R}^2$, the

situation is more complicated. However, the principle remains the same. And we can test for local maxima and minima in the same way as for one variable.



A graph illustrating local min/max and global min/max points

In mathematics, a point x^* is a **local maximum** of a function f if there exists some $\varepsilon > 0$ such that $f(x^*) \geq f(x)$ for all x with $|x-x^*| < \varepsilon$. Stated less formally, a local maximum is a point where the function takes on its largest value among all points in the immediate vicinity. On a graph of a function, its local maxima will look like the tops of hills.

A **local minimum** is a point x^* for which $f(x^*) \leq f(x)$ for all x with $|x-x^*| < \varepsilon$. On a graph of a function, its local minima will look like the bottoms of valleys.

A **global maximum** is a point x^* for which $f(x^*) \geq f(x)$ for all x. Similarly, a **global minimum** is a point x^* for which $f(x^*) \leq f(x)$ for all x. Any global maximum (minimum) is also a local maximum (minimum); however, a local maximum or minimum need not also be a global maximum or minimum.

The concepts of maxima and minima are not restricted to functions whose domain is the real numbers. One can talk about global maxima and global minima for real-valued functions whose domain is any set. In order to be able to define local maxima and local minima, the function needs to take real values, and the concept of neighborhood must be defined on the domain of the function. A neighborhood then plays the role of the set of x such that $|x - x^*| < \varepsilon$.

One refers to a local maximum/minimum as to a local extremum (or local optimum), and to a global maximum/minimum as to a global extremum (or global optimum).

Finding global maxima and minima is the goal of optimization. For twice-differentiable functions in one variable, a simple technique for finding local maxima and minima is to look for stationary points, which are points where the first derivative is zero. If the second derivative at a stationary point is positive, the point is a local minimum; if it is negative, the point is a local maximum; if it is zero, further investigation is required.

If the function is defined over a bounded segment, one also need to check the end points of the segment.

2.4.1.4 Examples

- The function x^2 has a unique global minimum at $x = 0$.
- The function x^3 has no global or local minima or maxima. Although the first derivative ($3x^2$) is 0 at $x = 0$, the second derivative ($6x$) is also 0.
- The function $x^3/3 - x$ has first derivative $x^2 - 1$ and second derivative $2x$. Setting the first derivative to 0 and solving for x gives stationary points at -1 and +1. From the sign of the second derivative we can see that -1 is a local maximum and +1 is a local minimum. Note that this function has no global maxima or minima.
- The function $|x|$ has a global minimum at $x = 0$ that cannot be found by taking derivatives, because the derivative does not exist at $x = 0$.
- The function $\cos(x)$ has infinitely many global maxima at 0, ±2π, ±π, ..., and infinitely many global minima at ±π, ±3π, ...
- The function $2\cos(x) - x$ has infinitely many local maxima and minima, but no global maxima or minima.
- The function $x^3 + 3x^2 - 2x + 1$ defined over the closed interval (segment) [-4,2] (see graph) has two extrema: one local maximum in $x = (-1 - \sqrt{15})/3$, one local minimum in $x = (-1 + \sqrt{15})/3$, a global maximum on $x=2$ and a global minimum on $x=-4$.

Definition Say that $f(x, y)$ has a **critical point** at (a, b) if and only if

$$\frac{\partial f}{\partial x}(a, b) = \frac{\partial f}{\partial y}(a, b) = 0.$$

It is clear by comparison with the single variable result, that a necessary condition that f have a local extremum at (a, b) is that it have a critical point there, although that is not a sufficient condition. We refer to this as the **first derivative test**.

We can get more information by looking at the second derivative. Recall that we gave a number of different notations for partial derivatives, and in what follows we use f_x rather than the more cumbersome $\frac{\delta f}{\delta x}$ etc. This idea extends to higher derivatives; we shall use

f_{xx} instead of $\frac{\partial^2 f}{\partial x^2}$, and f_{xy} instead of $\frac{\partial^2 f}{\partial x \partial y}$ etc.

Theorem (Second Derivative Test) Assume that (a, b) is a critical point for f. Then

- If, at (a, b), we have $f_{xx} < 0$ and $f_{xx}f_{yy} - f^2_{xy} > 0$, then f has a **local maximum** at (a, b).
- If, at (a, b), we have $f_{xx} > 0$ and $f_{xx}f_{yy} - f^2_{xy} > 0$, then f has a **local minimum** at (a, b).
- If, at (a, b), we have $f_{xx}f_{yy} - f^2_{xy} < 0$, then f has a **saddle point** at (a, b).

The test is **inconclusive** at (a, b) if $f_{xx}f_{yy} - f^2_{xy} = 0$, and the investigation has to be continued some other way.

Note that the discriminant is easily remembered as

$$\Delta = \begin{vmatrix} f_{xx} & f_{xy} \\ f_{yx} & f_{yy} \end{vmatrix} = f_{xx}f_{yy} - f^2_{xy}$$

A number of very simple examples can help to remember this. After all, the result of the test should work on things where we can do the calculation anyway!

Example 1 Show that $f(x, y) = x^2 + y^2$ has a minimum at (0, 0).

Of course we know it has a global minimum there, but here goes with the test:

Solution. We have $f_x = 2x$; $f_y = 2y$, so $f_x = f_y$ precisely when $x = y = 0$, and this is the only critical point. We have $f_{xx} = f_{yy} = 2$; $f_{xy} = 0$, so $\Delta = f_{xx}f_{yy} - f^2_{xy} = 4 > 0$ and there is a local minimum at (0, 0).

Exercise 2 Let $f(x, y) = xy$. Show there is a unique critical point, which is a saddle point

Proof. We give an indication of how the theorem can be derived -- or if necessary how it can be remembered. We start with the two dimensional version of Taylor's theorem, see section 5.6. We have

$$f(a + h, b + k) \sim f(a, b) + h\frac{\partial f}{\partial x}(a, b) + k\frac{\partial f}{\partial y}(a, b) + \frac{1}{2}\left(h^2\frac{\partial^2 f}{\partial x^2} + 2kh\frac{\partial^2 f}{\partial x \partial y} + k^2\frac{\partial^2 f}{\partial y^2}\right)$$

where we have actually taken an expansion to second order and assumed the corresponding remainder is small.

We are looking at a critical point, so for any pair (h, k), we have $h\frac{\partial f}{\partial x}(a, b) + k\frac{\partial f}{\partial y}(a, b) = 0$ and everything hinges on the behaviour of the second order terms. It is thus

enough to study the behaviour of the quadratic $Ah^2 + 2Bhk + Ck^2$, where we have written

$$A = \frac{\partial^2 f}{\partial x^2}, \quad B = \frac{\partial^2 f}{\partial x \partial y}, \quad \text{and} \quad C = \frac{\partial^2 f}{\partial y^2}.$$

Assuming that $A \neq 0$ we can write

$$Ah^2 + 2Bhk + Ck^2 = A\left(h + \frac{Bk}{A}\right)^2 + \left(C - \frac{B^2}{A}\right)k^2$$

$$= A\left(h + \frac{Bk}{A}\right)^2 + \left(\frac{\Delta}{A}\right)k^2$$

where we write $\Delta = CA - B^2$ for the discriminant. We have thus expressed the quadratic as the sum of two squares. It is thus clear that

- if $A < 0$ and $\Delta > 0$ we have a local maximum;
- if $A > 0$ and $\Delta > 0$ we have a local minimum; and
- if $\Delta < 0$ then the coefficients of the two squared terms have opposite signs, so by going out in two different directions, the quadratic may be made either to increase or to decrease.

Note also that we could have completed the square in the same way, but starting from the k term, rather than the h term; so the result could just as easily be stated in terms of C instead of A □

Example 3 Let $f(x, y) = 2x^3 - 6x^2 - 3y^2 - 6xy$. Find and classify the critical points of f. By considering $f(x, 0)$, or otherwise, show that f does not achieve a global maximum.

Solution. We have $f_x = 6x^2 - 12x - 6y$ and $f_y = -6y - 6x$. Thus critical points occur when $y = -x$ and $x^2 - x = 0$, and so at (0, 0) and (1, - 1). Differentiating again, $f_{xx} = 12x - 12$, $f_{yy} = -6$ and $f_{xy} = -6$. Thus the discriminant is $\Delta = -6.(12x - 12) - 36$. When $x = 0$, $\Delta = 36 > 0$ and since $f_{xx} = -12$, we have a local maximum at (0, 0). When $x = 1$, $\Delta = -36 < 0$, so there is a saddle at (1, - 1).

To see there is no global maximum, note that $f(x, 0) = 2x^3(1 - 3/x) \to \infty$ as $x \to \infty$, since $x^3 \to \infty$ as $x \to \infty$.

Exercise 4 Find the extrema of $f(x, y) = xy - x^2 - y^2 - 2x - 2y + 4$.

Example 5 An open-topped rectangular tank is to be constructed so that the sum of the height and the perimeter of the base is 30 metres. Find the dimensions which maximise the surface area of the tank. What is the maximum value of the surface area?

[You may assume that the maximum exists, and that the corresponding dimensions of the tank are strictly positive.]

Solution. Let the dimensions of the box be as shown.

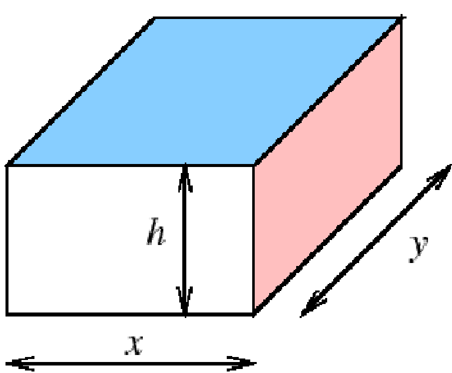


Figure: A dimensioned box

Let the area of the surface of the material be S. Then

$S = 2xh + 2yh + xy$,

and since, from our restriction on the base and height,

$30 = 2(x + y) + h$, we have $h = 30 - 2(x + y)$.

Substituting, we have

$$S = 2(x + y)\left(30 - 2(x + y)\right) + xy = 60(x + y) - 4(x + y)^2 + xy,$$

and for physical reasons, S is defined for $x \geq 0$, $y \geq 0$ and $x + y \leq 15$.

A global maximum (which we are given exists) can only occur on the boundary of the domain of definition of S, or at a critical point, when

$\frac{\partial S}{\partial x} = \frac{\delta S}{\partial y} = 0$. On the boundary of the domain of definition of S, we have $x = 0$ or $y = 0$ or $x + y = 15$, in which case $h = 0$. We are given that we may ignore these cases. Now

$S \quad -4x^2 - 4y^2 - 7xy + 60x + 60y$, so

$\frac{\partial S}{\partial x} \quad -8x - 7y + 60 = 0$,

$\frac{\partial S}{\partial y} \quad -8y - 7x + 60 = 0$.

Subtracting gives $x = y$ and so $15x = 60$, or $x = y = 4$. Thus $h = 14$ and the surface area is $S = 16(-4 - 4 - 7 + 15 + 15) = 240$ square metres. Since we are given that a maximum exists, this must be it. [If both sides of the surface are counted, the area is doubled, but the critical proportions are still the same.]

Sometimes a function necessarily has an absolute maximum and absolute minimum -- in the following case because we have a continuous function defined on a closed bounded subset of $\mathbb{R}^2$, and so the analogue of 4.35 holds. In this case exactly as in the one variable case, we need only search the boundary (using ad - hoc methods, which in fact reduce to 1-variable methods) and the critical points in the interior, using our ability to find local maxima.

Example 6 Find the absolute maximum and minimum values of

$f(x, y) = 2 + 2x + 2y - x^2 - y^2$

on the triangular plate in the first quadrant bounded by the lines $x = 0$, $y = 0$ and $y = 9 - x$

Solution. We know there is a global maximum, because the function is continuous on a closed bounded subset of $\mathbb{R}^2$. Thus the absolute max will occur either in the interior, at a critical point, or on the boundary. If $y = 0$, investigate $f(x, 0) = 2 + 2x - x^2$, while if $x = 0$, investigate $f(0, y) = 2 + 2y - y^2$. If $y = 9 - x$, investigate

$f(x, 9 - x) = 2 + 2x + 2(9 - x) - x^2 - (9 - x)^2$

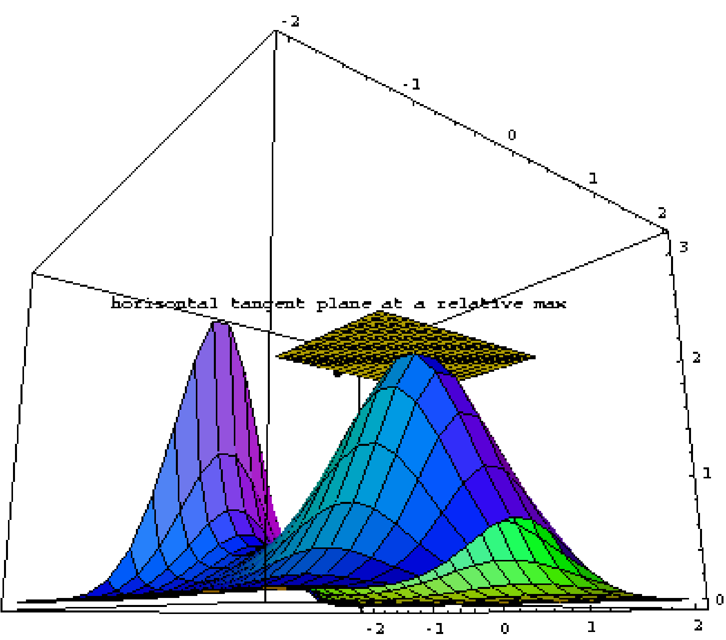
for an absolute maximum. In fact extreme may occur when $(x, y) = (0, 1)$ or $(1, 0)$ or $(0, 0)$ or $(9, 0)$ or $(0, 9)$, or $(9/2, 9/2)$. At these points, f takes the values -41/2, 2, 3, -61.

Next we seek critical points in the interior of the plate,

$f_x = 2 - 2x = 0$ and $f_y = 2 - 2y = 0$.

so $(x, y) = (1, 1)$ and $f(1, 1) = 4$, so this must be the global maximum. Can check also using the second derivative test, that it is a *local* maximum.

The notions of critical points and the second derivative test carry over to functions of two variables. Let z=f(x,y). Critical points are points in the xy-plane where the tangent plane is horizontal.



Since the normal vector of the tangent plane at (x,y) is given by

$$f_x(x,y)\mathbf{i} + f_y(x,y)\mathbf{j} - \mathbf{k}$$

The tangent plane is horizontal if its normal vector points in the z direction. Hence, *critical points are solutions of the equations*:

$$f_x(x,y) = 0 \quad \textit{and} \quad f_y(x,y) = 0$$

because horizontal planes have normal vector parallel to z-axis. The two equations above must be solved *simultaneously*.

Example 7

Let us find the critical points of

$$z = f(x,y) = \exp\left(-\frac{1}{3}x^3 + x - y^2\right)$$

The partial derivatives are

$$f_x(x,y) = (-x^2+1)\exp\left(-\frac{1}{3}x^3 + x - y^2\right)$$

$$f_y(x,y) = -2y\exp\left(-\frac{1}{3}x^3 + x - y^2\right)$$

f_x=0 if 1-x^2=0 or the exponential term is 0. f_y=0 if -2y=0 or the exponential term is 0. The exponential term is not 0 except in the degenerate case. Hence we require 1-x^2=0 and -2y=0, implying x=1 or x=-1 and y=0. There are two critical points (-1,0) and (1,0).

The Second Derivative Test for Functions of Two Variables:

How can we determine if the critical points found above are relative maxima or minima? We apply a second derivative test for functions of two variables.

Let (x_c,y_c) be a critical point and define

$$D(x_c, y_c) = f_{xx}(x_c, y_c) f_{yy}(x_c, y_c) - [f_{xy}(x_c, y_c)]^2.$$

We have the following cases:

- If D>0 and f_xx(x_c,y_c)<0, then f(x,y) has a relative maximum at (x_c,y_c).
- If D>0 and f_xx(x_c,y_c)>0, then f(x,y) has a relative minimum at (x_c,y_c).
- If D<0, then f(x,y) has a saddle point at (x_c,y_c).
- If D=0, the second derivative test is inconclusive.

An example of a saddle point is shown in the example below.

Example: Continued

For the example above, we have

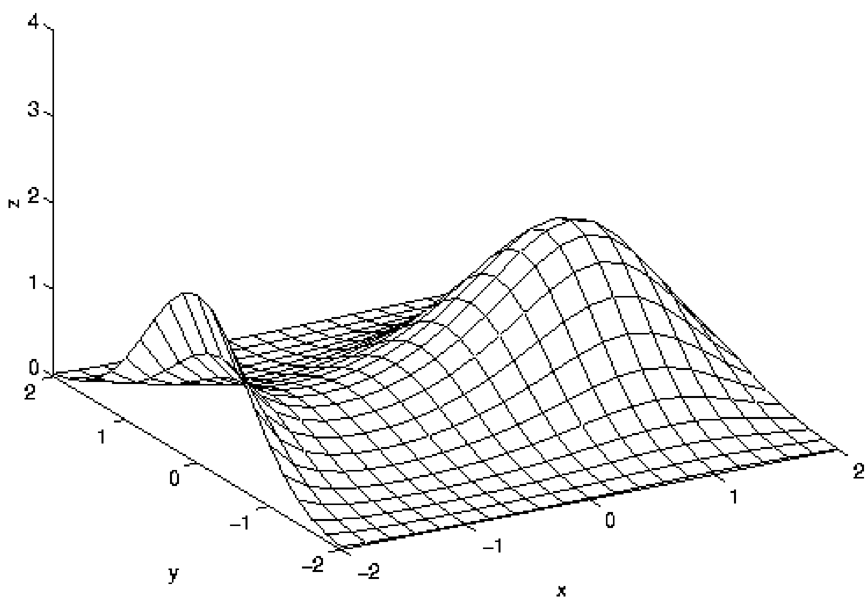
$$f_{xx}(x, y) = (-2x + (1 - x^2)^2) exp\left(-\frac{1}{3}x^3 + x - y^2\right),$$

$$f_{yy}(x, y) = (-2 + 4y^2) exp\left(-\frac{1}{3}x^3 + x - y^2\right),$$

$$f_{xy}(x, y) = -2y(1 - x^2) exp\left(-\frac{1}{3}x^3 - x - y^2\right),$$

For x=1 and y=0, we have D(1,0)=4exp(4/3)>0 with f_xx(1,0)=-2exp(2/3)<0. Hence, (1,0) is a relative maximum. For x=-1 and y=0, we have D(-1,0)=-4exp(-4/3)<0. Hence, (-1,0) is a saddle point.

The figure below plots the surface z=f(x,y).



Notice the relative maximum at (x=1,y=0). (x=-1,y=0) is a relative maximum if one travels in the y direction and a relative minimum if one travels in the x-direction. Near (-1,0) the surface looks like a saddle, hence the name.

2.4.1.5 Maxima and Minima in a Bounded Region

Suppose that our goal is to find the global maximum and minimum of our model function above in the square -2<=x<=2 and -2<=y<=2? There are three types of points that can potentially be global maxima or minima:

1. Relative extrema in the interior of the square.
2. Relative extrema on the boundary of the square.
3. Corner Points.

We have already done step 1. There are extrema at (1,0) and (-1,0). The boundary of square consists of 4 parts. Side 1 is y=-2 and -2<=x<=2. On this side, we have

$$z = f(x,-2) = exp\left(-\frac{1}{3}x^3 + x - (-2)^2\right) = g(x)$$

The original function of 2 variables is now a function of x only. We set g'(x)=0 to determine relative extrema on Side 1. It can be shown that x=1 and x=-1 are the relative extrema. Since y=-2, the relative extrema on Side 1 are at (1,-2) and (-1,-2).

On Side 2 (x=-2 and -2<=y<=2)

$$z = f(-2,y) = exp\left(-\frac{1}{3}(-2)^3 - 2 - y^2\right) = h(y).$$

We set h'(y)=0 to determine the relative extrema. It can be shown that y=0 is the only critical point, corresponding to (-2,0).

We play the same game to determine the relative extrema on the other 2 sides. It can be shown that they are (2,0), (1,2), and (-1,2).

Finally, we must include the 4 corners (-2,-2), (-2,2), (2,-2), and (2,2). In summary, the candidates for global maximum and minimum are (-1,0), (1,0), (1,-2), (-1,-2), (-2,0), (2,0), (1,2), (-1,2), (-2,-2), (-2,2), (2,-2), and (2,2). We evaluate f(x,y) at each of these points to determine the global max and min in the square. The global maximum occurs (-2,0) and (1,0). This can be seen in the figure above. The global minimum occurs at 4 points: (-1,2), (-1,-2), (2,2), and (2,-2).

2.4.1.6 Maxima and Minima in a Disk

Another example of a bounded region is the disk of radius 2 centered at the origin. We proceed as in the previous example, determining in the 3 classes above. (1,0) and (-1,0) lie in the interior of the disk.

The boundary of the disk is the circle x^2+y^2=4. To find extreme points on the disk we parameterize the circle. A natural parameterization is x=2cos(t) and y=2sin(t) for 0<=t<=2*pi. We substitute these expressions into z=f(x,y) and obtain

$$z = f(x,y) = f(\cos(t), \sin(t)) = exp\left(-\frac{8}{3}\cos^3 t + 2\cos t - 4\sin^2 t\right) = g(t)$$

On the circle, the original functions of 2 variables is reduced to a function of 1 variable. We can determine the extrema on the circle using techniques from calculus of on variable.

In this problem there are not any corners. Hence, we determine the global max and min by considering points in the interior of the disk and on the circle. An alternative method for finding the maximum and minimum on the circle is the method of Lagrange multipliers.

2.4.1.7 Functions of More than 2 Variables

The notion of extreme points can be extended to functions of more than 2 variables. Suppose z=f(x_1,x_2,...,x_n). (a_1,a_2,...,a_n) is extreme point if it satisfies the n equations

$$\frac{\partial f}{\partial x_i}(x_1, x_2, \ldots, x_n) = 0 \qquad i = 1, 2, \ldots, n.$$

There is not a general second derivative test to determine if a point is a relative maximum or minimum for functions of more than two variables.

Functional minimization is an important area of study in many fields, and a variety of numerical methods have been developed to address particular problems. We will look at just one such method, which is particularly effective at minimizing functions of many variables.

Let's define the problem. Let's say you want to minimize a function, *f(x)*, of a single variable, *x*. For a simple function, you follow the standard practice of solving for those values, x_1, x_2, x_3 ... such that: $f'(x_i)=0$, and $f''(x_i) > 0$. The value of x_i so obtained that gives the smallest $f(x_i)$ is the *global minimum* of the function. All other $f(x_i)$ give the *local minima* of the function.

It could happen that the function to be minimized is too unwieldy to evaluate the necessary derivatives and/or to solve for the roots of $f'(x_i)=0$. Furthermore, there could be constraints on the variable that must be factored into the minimization. Finally, we might need to minimize a function of many variables: $f(x_1,x_2,x_3...)$. In such cases, efficient numerical techniques to achieve minimization are essential.

It should be pointed out that the techniques used to minimize functions can immediately be used to maximize them: just take f to be $-f$. The field of minimization or maximization of functions is called *OPTIMIZATION*.

Many optimization techniques are described in the *Numerical Recipes* book used in PH430. In this module, we will focus on a particular application of the so-called "annealing method".

The system that we will be studying is called a "quantum dot". It consists of a group of electrons that are quantum-confined in all three spatial dimensions, and so they are analogous to atoms. Quantum dots can be fabricated in a variety of shapes, and the sizes are typically in the range of ~10's to 100's of nanometers--billionths of a meter. A great deal of research is currently ongoing with the goal of designing quantum dot systems for particular applications such as high-efficiency lasers, biological tracers, and quantum computers.

In the quantum dot system that we will model, a "puddle" of electrons is confined to a plane and restricted to lie near the center of a two-dimensional parabolic potential energy profile (such systems are actually made!). The parabolic potential tries to keep the electrons near the origin--they have to climb uphill to move away from the bottom of the potential well. This is counter-acted by the repulsion between electrons that keeps them apart. As a rough approximation, we will treat the electrons as classical point particles instead of quantum particles. We want to answer the following question:

We will be guided to the answer by noting that the electrons will want to reside in the configuration that minimizes their total energy:

$$E = K + V$$

For classical electrons we can immediately eliminate the Kinetic energy, K, by taking the electrons to be stationary. Then, our problem amounts to minimizing the potential energy, V.

Let's consider the simple cases first: for the case of one electron, there is no Coulomb energy and the single electron will sit at the bottom of the potential well with E=0. For two electrons the potential energy is:

$$V = \frac{1}{2} m\omega_0^2 (\rho_1^2 + \rho_2^2) + \frac{e^2}{|\rho_1 - \rho_2|}$$

where ρ_1 and ρ_2 are the 2D coordinates of the 2 electrons, and ω_0 defines the curvature of the potential energy parabola. Note that since $\rho_1 = x_1\hat{x} + y_1\hat{y}$ and

$\rho_2 = x_2\hat{x} + y_2\hat{y}$, V is now a function of the 4 the variables, x_1, y_1, x_2, y_2. We can reduce this down to a single variable as follows.

Let's make a transformation to center-of-mass (CM) and relative coordinates:

$$\mathbf{R} = \frac{1}{2}(\rho_1 + \rho_2)$$

$$\rho = \rho_1 - \rho_2$$

with the inverse transformation:

$$\rho_1 = \mathbf{R} + \frac{1}{2}\rho$$

$$\rho_2 = \mathbf{R} - \frac{1}{2}\rho$$

Plugging in to V gives:

$$V(\rho_1, \rho_2) = V(\mathbf{R}, \rho) = \frac{1}{2}M\omega_0^2 R^2 + (\frac{1}{2}\mu\omega_0^2\rho^2 + \frac{e^2}{\rho})$$

with $M = m_1 + m_2 = 2m$ and $\mu = m_1 m_2/(m_1 + m_2) = m/2$.

Notice that the relative and CM parts of V now separate. It is clear that we can lower the energy by choosing the location of the CM at the origin, ***R***=*0*. This means that $\rho_1 = -\rho_2$.

Then,

$$V(\rho) = \frac{1}{2}\mu\omega_0^2\rho^2 + \frac{e^2}{\rho}$$

Now, *V* is a simple function of a single variable. The minimum is located from:

$$\frac{\partial V}{\partial \rho} = \frac{1}{2}m\omega_0^2\rho_0 - \frac{e^2}{\rho_0^2} = 0$$

$$\rho_0 = \left(\frac{2e^2}{m\omega^2}\right)^{1/3}$$

$$V(\rho_0) = \frac{3}{2}E_0 \qquad E_0 = (\frac{m\omega_0^2 e^4}{2})^{1/3}$$

Of course, there is a much easier way of extracting this result. At the minimum of the total potential energy, the net force on each electron must be zero.

Calculate the location of each electron in the two-electron dot by requiring that the sum of the forces on each electron vanish.

If we now consider higher numbers of electrons, N=3,4,..., the minimization procedure used above becomes intractable. For N electrons, the potential energy is:

$$V(\rho_1, \rho_2, \ldots \rho_N) = \frac{1}{2} m\omega_0^2 \sum_{i=1}^{N} \rho_i^2 + e^2 \sum_{i>j}^{N} \frac{1}{|\rho_i - \rho_j|}$$

For simplicity, we can switch to new variables, r_i, measured in units of ρ_0 and measure energy in units of E_0. Then, the potential energy is simplified to:

$$V(\mathbf{r}_1, \mathbf{r}_2, \ldots \mathbf{r}_N) = \sum_{i=1}^{N} r_i^2 + \sum_{i>j}^{N} \frac{1}{|\mathbf{r}_i - \mathbf{r}_j|}$$

If N=100, we have 200 independent variables. Transformation of coordinates as above will separate out the CM part. Requiring the CM to be at the origin introduces only 2 constraints and so leaves us with 198 variables! We need a minimization technique that can accommodate the large number of variables arising in this problem.

Summary:

maximum or minimum of function is encountered in geometry, mechanics, physics, and other fields, and was one of the motivating factors in the development of the calculus in the seventeenth century. One of the great powers of calculus is in the determination of the maximum or minimum value of a function. The field of minimization or maximization of functions is called *OPTIMIZATION*.

Model Questions:

1. Briefly explain the maxima and minima of functions?
2. Outline the procedure to find out the maxima & minima of functions?

References:

1. Linear Algebra and its Applications, 3rd ed. by Strang, G
2. Calculus (Early Transcendentals) 3rd ed.by James Stewart.

AUTHOR:

B. M.REDDY M.Tech. (HBTI, Kanpur)

Lecturer, Centre for Biotechnology

Acharya Nagarjuna University.

Lesson 2.4.2

RANDOMIZED MINIMIZATION TECHNIQUES

Objective
2.4.2.1 Introduction
2.4.2.2 Randomized Unit Testing
2.4.2.3 Three robust design paradigms
2.4.2.4 Properties of the scenario design
Summary
Model Questions
References

Objective

Engineering design problems can often be cast as numerical optimization programs where the designer goal is to minimize a cost index (or maximize a utility index), subject to a set of constraints on the decision variables. In particular, a class of design problems that have both a theoretical and a practical relevance are those where the minimization objective and the constraints are convex functions of the variables.

However, in practice, the problem data are often *uncertain,* and hence the design needs not only be optimal, but also *guaranteed,* or *robust,* against the uncertainty. Unfortunately, it has been proven that convex problems in which uncertainty is present are very hard to solve. In this note, we discuss a novel technique based on uncertainty randomization that permits to overcome this difficulty.

2.4.2.1 Introduction

The main motivation for studying robustness problems in engineering comes from the fact that the actual system (a "plant," or in general a problem involving physical data) upon which the engineer should act, is realistically not fixed but rather it entails some level of uncertainty. For instance, the data that characterizes an engineering problem typically depends on the value of physical parameters. If measurements of these parameters are performed, say, on different days or under different operating conditions, it is likely that we will end up not with a single (nominal) problem representation D, but rather with a family $D(\delta)$ of possible problems, where $\delta\ E\ \Delta$ represents the vector of uncertain parameters that affect the data, and Δ is the admissible set of variation of the parameters. Any sensible design should in some way take into account the variability in the problem data, i.e. it should be *robust* with respect to the uncertainty.

To formalize our setup, we consider specifically design problems that may be expressed in the form of minimization of a linear objective subject to convex constraints:

$$\min_{x \in \mathcal{X}} \quad c^T x \quad \text{subject to:}$$
$$f_i(x, \delta) \leq 0, \quad i = 1, \ldots, m$$

where $\mathcal{X} \subseteq \mathbb{R}^n$ is a convex set, and the functions *fi(x,δ)* that define the constraints are convex in the decision variable *x,* for any given value of the uncertainty δ G Δ. Without loss of generality, we can actually consider problems with a single constraint function

$$f(x, \delta) \doteq \max_{1,\ldots,m} f_i(x, \delta)$$

(the maximum of convex functions is still convex), which yields a prototype problem in the form

$$\min_{x \in \mathcal{X}} \quad c^T x \quad \text{subject to:}$$
$$f(x, \delta) \leq 0.$$

Notice that in this problem statement we remained voluntarily vague as to the meaning of robustness. We shall next define and briefly discuss three different ways in which this robustness can be intended.

2.4.2.2. RANDOMIZED UNIT TESTING

A. Controlflow and Dataflow Views of Drivers

In a flowchart view of the operation of a unit test driver (Figure 1), the driver performs a sequence of test cases, checks the results, and judges the success or failure of each test case. For example, test cases for the Course class may include one that tries to register a student having the prerequisites, one that tries to register a student not having the prerequisites, and one that tries to register students when the size limit has been reached. The driver could be programmed from scratch or built using a framework, in which each test case would be represented by one test method.

For the purpose of generalizing a test driver to a randomized test driver, it is more useful to view the driver not in terms of its control flow, but rather in terms of its data flow, because the randomized operations will randomly change the values of the data used. In this dataflow-oriented view (Figure 1b), the driver executes a series of *test fragments,* each of which might either perform a call to a UUT method, set up parameters for future method calls, or extract information from the results of past method calls.

The driver will typically have one or more local variables, such as instances of a class under test or variables to be used as parameters or results of UUT methods. Each local variable whose value is set by one test fragment and whose value is then used by a future test fragment is here called a *persistent variable* (PV in Figure 1b). The driver can therefore be seen as a program that sets initial values for the persistent variables, and then executes test fragments, each of which may change the value of one or more of the persistent variables. For example, a Course driver may have one Course persistent variable and one Student persistent variable; the test fragments may initialize or reinitialize the Student to a new object instance, may add more information to the Student about past courses taken, and/or may call the register method, giving the Student as a parameter.

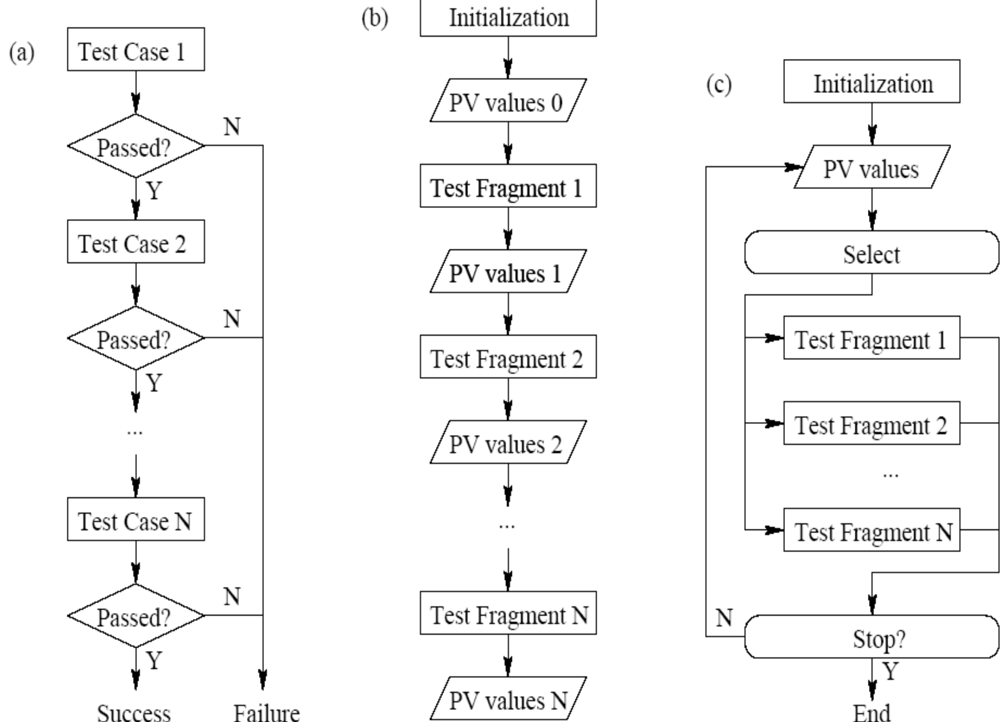


Fig. 1. (a) Controlflow-oriented view of a unit driver. (b) Dataflow-oriented view of a unit driver. (c) A randomized unit driver based on the dataflow view.

B. Randomized Test Drivers

The kind of randomized test drivers that we consider are generalizations of the dataflow view of unit test drivers. The randomized driver initializes the persistent variables, and then randomly selects and executes a test fragment. It keeps randomly selecting and executing test fragments until some stopping condition is met, such as when a certain number of test fragments has been executed. Each of these test fragments may evaluate preconditions and skip further processing of its target method calls if the preconditions are not met - for instance, if the persistent variables are not

yet in a form that can be passed as a parameter. Each fragment may itself contain a random element, such as in the selection of scalar arguments for method calls. Each fragment may evaluate results, or just output information about the results for later evaluation, as with a test oracle.

For example, a randomized driver for Course might contain a test fragment that reinitializes the Student, another that randomly selects a "past course taken" from a fixed list of courses and adds it to the Student, and another that calls the register method. A random sequence of test fragment executions is very likely to eventually make both valid and invalid calls to register, and may execute test sequences not accounted for by the writer of a non-randomized test driver, such as calling register twice with the same Student.

Such randomized drivers are a strict generalization of the dataflow view of conventional non-randomized drivers, such as JUnit drivers. Since conventional drivers are commonly written for units (for instance, JUnit is widely used in industry), it is reasonable to expect test engineers to be able to write randomized test drivers from scratch. However, there is clearly potential for automating parts of the process of writing the randomized test drivers.

C. Test Case Minimization

Although randomized unit testing can accurately find failing test cases, the test inputs that cause the failure can contain both commands relevant to the failure and random commands that are irrelevant, in the sense of not contributing to forcing the failure. Separating the relevant parts from the irrelevant parts is a difficult task for the human debugger; for example, Andrews estimated that half the time spent in randomized testing of some units consisted of tracking down and fixing bugs.

It is therefore natural to ask how much benefit could be obtained by automatically reducing the size of the failing test cases. The state of the art in this area is Zeller and Hilde-brandt's Delta Debugging minimization algorithm, *ddmin*. We briefly summarize this algorithm here.

First some definitions. A *failing test case* is a test case that causes the SUT to perform incorrectly. Since we assume software with text inputs, and since the unit of input to our test driver programs is a one-line command, the *length* of a test case is the number of lines in its input file. A *global minimized* failing test case is one whose length is less than or equal to the length of all other failing test cases. A *local minimized* failing test case is one such that deleting any one line in the input file does not result in a failing test case. The *ddmin* algorithm does not attempt to find global minimized failing test cases; instead it finds local minimized failing test cases based on a given failing test case.

2.4.2.3 Three robust design paradigms

1 Worst-case design

A first paradigm is a worst-case one, in which we seek a design x that satisfies the constraints *for all possible realizations* of the uncertain parameter δ. In formal terms, the design problem becomes

$$\min_{x \in \mathcal{X}} \quad c^T x \quad \text{subject to:} \tag{1}$$
$$f(x, \delta) \leq 0, \quad \forall \delta \in \Delta.$$

A worst-case design may be necessary in cases when the violation of a constraint is associated to an unacceptable cost or it has disastrous consequences. It is a "pessimistic" paradigm in which the designer tries to be guaranteed against all possible odds. For the same reason, this kind of design tends to be conservative, since it takes into account also very rare events that may never realize in practice. Also, from a computation point of view, worst-case design problems are, in general, computationally hard. Even under the convexity assumptions the convex optimization problem (1) entails a usually infinite number of constraints. This class of problem goes under the name of robust convex programs, which are known to be NP-hard, see for instance.

2 Probabilistic design

A second paradigm for robustness is a probabilistic one. In this setup, we add further structure on the problem, assuming that δ is a random variable with assigned probability distribution over Δ. Then, the probabilistic design objective is to determine a parameter x that satisfies the constraints up to a given high level of probability, p G $(0,1)$. Formally, the optimization problem takes the form:

$$\min_{x \in \mathcal{X}} \quad c^T x \quad \text{subject to:} \tag{2}$$
$$\text{Prob}\{f(x, \delta) \leq 0\} \geq p. \tag{3}$$

This design formulation alleviates the pessimism inherent in the worst-case design, but still gives rise to a numerically untractable problem. In fact, even under the convexity assumption, problem (2) can be extremely hard to solve exactly in general. This is due to the fact that the probability in the so-called "chance constraint" (3) can be hard to compute explicitly and, more fundamentally, to the fact that the restriction imposed on x by (3) is in general *nonconvex,* even though $f(x, \delta)$ is convex in x, see for instance.

3 Sampled scenarios design

Finally, we define a third approach to robustness, which is the scenario approach: let $\delta^{(1)}, \ldots, \delta^{(N)}$ be N independent and identically distributed random samples of the uncertainty $\delta \in \Delta$, extracted according to some assigned probability distribution. Each sample corresponds to a different realization (scenario) of the uncertain parameters upon which the problem data depend. If the problem solver task is to devise a once and for all fixed policy that performs well on the actual (unknown) problem, a sensible strategy would be to design this policy such that it performs well on all the collected scenarios. This is of course a well-established technique which is widely used in practical problems, and it is for instance the standard way in which uncertainty is dealt with in difficult financial planning problems, such as multi-stage stochastic portfolio optimization. We hence define the *scenario design problem* as:

$$\min_{x \in \mathcal{X}} \quad c^T x \quad \text{subject to:} \tag{4}$$

$$f(x, \delta^{(i)}) \leq 0, \quad i = 1, \ldots, N. \tag{5}$$

We readily notice that the scenario problem is a standard convex problem with a finite number of constraints, and therefore it is generally solvable efficiently by numerical techniques, such as interior point methods.

While simple and effective in practice, the scenario approach also raises interesting theoretical questions. First, it is clear that a design that is robust for given scenarios is not robust in the worst-case sense, unless the considered scenarios actually contain all possible realizations of the uncertain parameters. Also, satisfaction of the constraints for the considered scenarios does not a-priori enforce the probabilistic constraint (3). Then, it becomes natural to ask what is the relation between robustness in the scenario sense and the probabilistic robustness. It turns out that a design based on scenarios actually guarantees a specified level of probabilistic robustness, provided that the number N of scenarios is chosen properly.

2.4.2.4 Properties of the scenario design

In this section we analyze in further detail the properties of the solution of the scenario-robust design problem (4). The first results on sampling-based convex optimization appeared recently in the paper [3], whereas an important refinement of these results is given in [2]. This section is based on the results contained in these references.

Denote with *XN* the optimal solution of (4), assuming that the problem is feasible and the solution is attained. Notice that problem (4) is certainly feasible whenever the worst-case problem (1) is feasible, since the former involves a subset of the constraints of the latter. Notice further that the optimal solution *XN* is a random variable, since it depends on the sampled random scenarios $\delta^{(1)}, \ldots, \delta^{(N)}$.

The following key result establishes the connection between the scenario approach and the probabilistic approach to robust design.

Theorem 1 (Scenario optimization) *Let* p, β G $(0,1)$ *be given probability levels, and let* x_N *denote the optimal solution*[1] *of problem (4), where the number N of scenarios has been selected so that*

$$N \geq \frac{2}{1-p}\ln\frac{1}{\beta} + 2n + \frac{2n}{1-p}\ln\frac{2}{1-p}. \qquad (6)$$

Then, it holds with probability at least $1 - \beta$ *that*

$$\text{Prob}\{f(x,\delta) \leq 0\} \geq p.$$

What it is claimed in the above theorem is that if the number of scenarios is selected according to the bound (6), then the optimal solution returned by the scenario-robust design has, with high probability $1 - \beta$, a guaranteed level p of probabilistic robustness. An important feature of bound (6) is that the probability level β enters it under a logarithm, and therefore β may be chosen very small without substantially increasing the required number of samples. For instance, setting $\beta = 10^{-9}$, we have the bound

$$N \geq \frac{41.5}{1-p} + 2n + \frac{2n}{1-p}\ln\frac{2}{1-p}.$$

Summary:

Seeking robustness in design by considering different scenarios has long been a common practice in engineering. However, the application of the sampling technique was mainly driven by heuristics, and no general result was previously available to answer the key question: "how many scenarios are needed to guarantee some given level of robustness" ?

The methodology that we briefly illustrated here actually answers in a rigorous way this fundamental question. The result from Theorem 1 states that the number of required scenarios scales gracefully with the problem dimension n and with the probabilistic levels. Convex optimization based on sampled scenarios is thus a simple, rigorous and efficient way to achieve robustness in design.

In a broader perspective, methods based on sampling and randomization have proved to be effective in solving in a relaxed sense problems that are otherwise hard to attack by means of classical deterministic techniques. Specific applications of

randomized techniques in the domain of dynamic systems and robust control are extensively discussed in the recent monograph [8]. A comprehensive and up-to-date account of general probabilistic optimization methods, along with many pointers to the literature, is instead available in the edited monograph [4].

Model Questions:

1. Draw a flow chart for randomized unit testing and explain it?
2. Explain the randomized minimization techniques and how they can help in bioinformatics?

References:

1. Randomized Techniques for Design Under Uncertainty *by* Giuseppe Carlo Calafiore

2. Minimization of Randomized Unit Test Cases by Yong Lei and James H. Andrews

AUTHOR:

B.M.REDDY M.Tech. (HBTI, Kanpur)

Lecturer, Centre for Biotechnology

Acharya Nagarjuna University.

Lesson 2.4.3

FOURIER TRANSFORM FOR DISCRETELY SAMPLED DATA

CONTENTS

2.4.3.1 Introduction
2.4.3.2 Fourier Transform
2.4.3.3 DFT Definition
2.4.3.4 A derivation of the discrete Fourier transform
2.4.3.5 Properties
2.4.3.6 Generalized DFT
2.4.3.7 Applications
2.4.3.8 Some discrete Fourier transform pairs
2.4.3.9 Variants of the Fourier transform
2.4.3.10 Applications

2.4.3.1 Introduction

Jean Baptiste Fourier showed that any signal or waveform could be made up just by adding together a series of pure tones (sine waves) with appropriate amplitude and phase. This is a rather startling theory, if you think about it. It means, for instance, that by simply turning on a number of sine wave generators we could sit back and enjoy a Beethoven symphony. Of course we would have to use a very large number of sine wave generators, and we would have to turn them on at the time of the Big Bang and leave them on until the heat death of the universe. Fourier's theorem assumes we add sine waves of infinite duration.

2.4.3.2 Fourier Transform

Before we get started on the DFT, let's look for a moment at the *Fourier transform* (FT) and explain why we are not talking about it instead. The Fourier transform of a continuous-time signal $x(t)$ may be defined as

$$X(\omega) = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} dt, \qquad \omega \in (-\infty, \infty).$$

Thus, right off the bat, we need *calculus*. The DFT, on the other hand, replaces the infinite integral with a finite sum:

$$X(\omega_k) \triangleq \sum_{n=0}^{N-1} x(t_n) e^{-j\omega_k t_n}, \qquad k = 0, 1, 2, \ldots, N-1,$$

where the various quantities in this formula are defined on the next page. Calculus is not needed to define the DFT (or its inverse), and with finite summation limits, we cannot

encounter difficulties with infinities (provided $x(t_n)$ is finite, which is always true in practice). Moreover, in the field of digital signal processing, signals and spectra are processed only in *sampled* form, so that the DFT is what we really need anyway (implemented using the FFT when possible). In summary, the DFT is *simpler mathematically*, and *more relevant computationally* than the Fourier transform. At the same time, the basic concepts are the same.

In mathematics, the discrete Fourier transform (DFT), sometimes called the finite Fourier transform, is a Fourier transform widely employed in signal processing and related fields to analyze the frequencies contained in a sampled signal, solve partial differential equations, and to perform other operations such as convolutions. The DFT can be computed efficiently in practice using a fast Fourier transform (FFT) algorithm.

2.4.3.3 DFT Definition

The *Discrete Fourier Transform (DFT)* of a signal x may be defined by

$$X(\omega_k) \triangleq \sum_{n=0}^{N-1} x(t_n) e^{-j\omega_k t_n}, \qquad k = 0, 1, 2, \ldots, N-1,$$

where

$$\begin{aligned}
& \triangleq && \text{means "is defined as" or "equals by definition"} \\
\sum_{n=0}^{N-1} f(n) & \triangleq && f(0) + f(1) + \cdots + f(N-1) \\
x(t_n) & \triangleq && \text{input signal } \textit{amplitude} \text{ (real or complex) at time } t_n \text{ (sec)} \\
t_n & \triangleq && nT = n\text{th sampling instant (sec)}, n \text{ an integer} \geq 0 \\
T & \triangleq && \text{sampling interval (sec)} \\
X(\omega_k) & \triangleq && \textit{spectrum} \text{ of } x \text{ (complex valued), at frequency } \omega_k \\
\omega_k & \triangleq && k\Omega = k\text{th frequency sample (radians per second)} \\
\Omega & \triangleq && \frac{2\pi}{NT} = \text{radian-frequency sampling interval (rad/sec)} \\
f_s & \triangleq && 1/T = \textit{sampling rate} \text{ (samples/sec, or Hertz (Hz))} \\
N & = && \text{number of time samples} = \text{no. frequency samples (integer).}
\end{aligned}$$

The sampling interval T is also called the sampling *period*. For a tutorial on sampling continuous-time signals to obtain non-aliased discrete-time signals.

When all N signal samples $x(t_n)$ are real, we say $x \in \mathbf{R}^N$. If they may be complex, we write $x \in \mathbf{C}^N$. Finally, $n \in \mathbf{Z}$ means n is any integer.

The sequence of n complex numbers x_0, ..., x_{n-1} are transformed into the sequence of n complex numbers f_0, ..., f_{n-1} by the DFT according to the formula:

$$f_j = \sum_{k=0}^{n-1} x_k e^{-\frac{2\pi i}{n} jk} \qquad j = 0, \dots, n-1$$

Note that the normalization factor multiplying the DFT and IDFT (here 1 and $1/n$) and the signs of the exponents are merely conventions, and differ in some treatments.

2.4.3.4 A derivation of the discrete Fourier transform

The discrete Fourier transform article defines the transform as:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i \frac{2\pi}{N} kn} \qquad k = 0, \dots, N-1$$

and notes that it can be derived as the continuous Fourier transform of infinite periodic sequences of impulses. That is an instructive exercise:

When the sequence $\{x_n\}$ represents a subset of the samples of a waveform $x(t)$, we can model the process that created $\{x_n\}$ as applying a window function to $x(t)$, followed by sampling (or vice versa). It is instructive to envision what those operations do to the Fourier transform, $X(f)$. The window function widens every frequency component of $X(f)$ in a way that depends on the type of window used. That effect is called spectral leakage. We can think of it as causing $X(f)$ to blur... thus a loss of resolution. The sampling operation causes the Fourier transform to become periodic. Copies of the blurred $X(f)$ are repeated at regular multiples of the sampling frequency, F_s, and summed together where they overlap. The copies are aliases of the original frequency components. In particular, due to the overlap, aliases can significantly distort the region containing the original $X(f)$ (if F_s is not sufficiently large enough to prevent it). But if the windowing and sampling are done with sufficient care, the Fourier transform still contains a reasonable semblance of $X(f)$. The transform is defined as:

$$
\begin{aligned}
S(f) &= \int_{-\infty}^{\infty} e^{-i2\pi ft} \cdot \left[\sum_{n=0}^{N-1} x_n \cdot \delta(t - \frac{n}{F_s})\right] dt \\
&= \sum_{n=0}^{N-1} x_n \cdot \left[\int_{-\infty}^{\infty} e^{-i2\pi ft} \cdot \delta(t - \frac{n}{F_s}) dt\right] \\
&= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi \frac{f}{F_s} n}
\end{aligned}
$$

This continuous Fourier transform is valid for all frequencies, including the discrete subset**:**

$$f_k = \frac{k}{N} F_s \qquad k = 0, \ldots, N-1$$

One thing to note about this subset is that the width of the region it spans is F_s, which is the periodicity of $S(f)$. So there is no need for more frequencies at this spacing $(\frac{F_s}{N})$.

Another thing to notice is that**:** $S(f_k) = X_k$ .So the DFT coefficients are a subset of the actual (continuous) Fourier transform of the windowed and sampled waveform.

Periodicity of the time-samples has not been assumed.

In fact up to this point, it is specifically in contradiction with the assumed window function.

Without it, the continuous Fourier transform, $S(f)$, is indeed continuous. So a different function, can be imagined by sampling it**:**

$$X_{dft}(f) = S(f) \cdot \sum_{m=-\infty}^{\infty} \delta(f - \frac{F_s}{N} m)$$

This transform also has the property: $X_{dft}(f_k) = X_k$. But of course it is zero in between the non-zero samples.

Its inverse transform is the original $\{x_n\}$ sequence, repeated at intervals of $\frac{N}{F_s}$.

So in summary, the DFT can be viewed in a couple of different ways:

1. as points on the spectrum of the windowed time-samples (i.e., the blurred and possibly aliased $X(f)$), or
2. as the non-zero coefficients of the spectrum of the windowed time-samples that have been periodically extended (to infinity)

Discrete Time Fourier Transform

For completeness, we note that with this definition:

$$\omega = \frac{2\pi f}{F_s}$$

we can also write $S(f)$(now a function of ω) as:

$$\begin{aligned} X_{dtft}(\omega) &= \sum_{n=0}^{N-1} x_n \cdot e^{-i\omega n} \\ &= \sum_{n=-\infty}^{\infty} x_n \cdot e^{-i\omega n} \end{aligned}$$

, (because of the window function)

which is now recognizable as the discrete-time Fourier transform.

In conclusion, rather than simply presenting the DFT and the DTFT as definitions, we have shown how they can be viewed as the logical result of applying the standard Fourier transform to discrete data. From that perspective, we have the satisfying result that it's not the transform that varies, it's just the form of the input:

- If it is discrete, the Fourier transform becomes a DTFT.
- If it is periodic, the Fourier transform becomes a Fourier series.
- If it is both, the Fourier transform becomes a DFT.

2.4.3.5 Properties

1. **Completeness**

The discrete Fourier transform is an invertible, linear transformation

$$\mathcal{F} : \mathbf{C}^n \to \mathbf{C}^n$$

with C denoting the set of complex numbers. In other words, for any $n \geq 0$, any n-dimensional complex vector has a DFT and an IDFT which are in turn n-dimensional complex vectors.

2. **Orthogonality**

The vectors $\exp(2\pi i\, jk/n)$ form an orthogonal basis over the set of n-dimensional complex vectors:

$$\sum_{k=0}^{n-1} \left(e^{2\pi ijk/n}\right) \left(e^{-2\pi ij'k/n}\right) = n\ \delta_{jj'}$$

where δ_{jk} is the Kronecker delta.

3. **The Plancherel theorem and Parseval's theorem**

If X_j and Y_j are the DFTs of x_k and y_k respectively then we have the Plancherel theorem:

$$\sum_{k=0}^{n-1} x_k y_k^* = \frac{1}{n} \sum_{j=0}^{n-1} X_j Y_j^*$$

where the star denotes complex conjugation. Parseval's theorem is a special case of the Plancherel theorem and states:

$$\sum_{k=0}^{n-1} |x_k|^2 = \frac{1}{n} \sum_{j=0}^{n-1} |X_j|^2.$$

4. **The shift theorem**

Multiplying x_n by a *linear phase* $\exp(2\pi inm / N)$ for some integer m corresponds to a *circular shift* of the output X_k: X_k is replaced by X_{k-m}, where the subscript is interpreted modulo N (i.e. periodically). Similarly, a circular shift of the input x_n corresponds to multiplying the output X_k by a linear phase. Mathematically, if $\{x_n\}$ represents the vector **x** then

if $\mathcal{F}(\{x_n\})_k = X_k$

then $\mathcal{F}(\{x_n e^{\frac{2\pi i}{N}nm}\})_k = X_{k-m}$

and $\mathcal{F}(\{x_{n-m}\})_k = X_k e^{-\frac{2\pi i}{N}km}$

5. Periodicity and aliasing

Although the DFT transforms N numbers to N numbers, in many ways it can be thought of as implicitly operating on infinite *periodic* sequences of both inputs and outputs. (Indeed, the DFT can be derived as the continuous Fourier transform of infinite periodic sequences of impulses.)

If one simply evaluates the DFT formula at $k + N$ then one finds that $X_{k+N} = X_k$, because $\exp[-2\pi i(k + N)n / N]$ is equal to $\exp[-2\pi ikn / N]\exp[-2\pi iNn / N]$ which equals $\exp[-2\pi ikn / N]$. This phenomenon is called aliasing, and it means that in a discrete signal one cannot distinguish between frequencies k that differ by N.

For the same reason, if one evaluates the inverse DFT formula at $n + N$ then one finds that $x_{n+N} = x_n$. Thus, the *inputs* also have implicitly periodic boundaries.

The shift theorem, above, is also an expression of this implicit periodicity, because it shows that the DFT amplitudes $| X_k |$ are unaffected by a circular (periodic) shift of the inputs, which is simply a choice of origin and therefore only affects the phase.

These periodic boundary conditions play an important role in many applications of the DFT. When solving differential equations they allow periodic boundary conditions to be automatically satisfied, and thus can be a useful property. For digital signal processing on the other hand, the periodicity is usually an obstacle—not only does it alias high frequencies with low frequencies, as noted above, but it also tends to introduce artifacts because natural signals are often non-periodic (resulting in implied discontinuities between the ends of the input).

6. Circular Convolution theorem and cross-correlation theorem

The cyclic convolution x*y of the two vectors x = x_j and y = y_k is the vector x*y with components

$$(\mathbf{x} * \mathbf{y})_k = \sum_{j=0}^{n-1} x_j y_{k-j} \qquad k = 0, \ldots, n-1$$

where we continue y cyclically so that

$$y_{-j} = y_{n-j} \qquad j = 0, \ldots, n-1$$

The discrete Fourier transform turns cyclic convolutions into component-wise multiplication. That is, if $z_k = (\mathbf{x} * \mathbf{y})_k$ then

$$Z_j = X_j Y_j \qquad j = 0, \ldots, n-1$$

where capital letters (X, Y, Z) represent the DFTs of sequences represented by small letters (x, y, z). Note that if a different normalization convention is adopted for the DFT (e.g., the unitary normalization), then there will in general be a constant factor multiplying the above relation.

The direct evaluation of the convolution summation, above, would require $O(n^2)$ operations, but the DFT (via an FFT) thus provides an $O(n\log n)$ method to compute the same thing. Conversely, convolutions can be used to efficiently compute DFTs via Rader's FFT algorithm and

Bluestein's FFT algorithm.

In an analogous manner, it can be shown that if z_k is the cross-correlation of x_j and y_j:

$$z_k = (\mathbf{x} \star \mathbf{y})_k = \sum_{j=0}^{n-1} x_j^* \, y_{j+k}$$

where the sum is again cyclic in j, then the discrete Fourier transform of z_k is:

$$Z_k = X_k^* Y_k$$

where capital letters are again used to signify the discrete Fourier transform.

7. Relationship to the trigonometric interpolation polynomial

The function

$$p(t) = \frac{f_0}{n} + \frac{f_1}{n}e^{it} + \frac{f_2}{n}e^{2it} + \cdots + \frac{f_{n-1}}{n}e^{(n-1)it}$$

whose coefficients f_j / n are given by the DFT of x_k, above, is called the trigonometric interpolation polynomial of degree $n - 1$. It is the unique function of this form that satisfies the property: $p(2\pi k/n) = x_k$ for $k = 0, \ldots, n - 1$.

8. The DFT as a unitary transformation

With unitary normalization constants, the DFT becomes a unitary transformation. In a real vector space, a unitary transformation can be thought of as simply a rigid rotation of the coordinate system, and all of the properties of a rigid rotation can be found in the unitary DFT. Using unitary normalization, we can express the DFT as:

$$x_{k'} = \sum_{j=0}^{n-1} \mathcal{F}_{k'j} x_j$$

where primed coordinates indicate the components of the vector x in the transformed space and

$$\mathcal{F}_{k'j} = \frac{e^{2\pi i j k'/n}}{\sqrt{n}}$$

The orthogonality of the DFT is now expressed as an orthonormality condition:

$$\sum_{m=0}^{n-1} \mathcal{F}_{j'm} \mathcal{F}_{k'm}^* = \delta_{j'k'}$$

The Plancherel theorem is expressed as:

$$\sum_{j=0}^{n-1} x_j y_j^* = \sum_{j'=0}^{n-1} x_{j'} y_{j'}^*$$

which is the statement that the dot product of two vectors is preserved under a unitary DFT transformation. This means that the angle between two vectors is preserved as well, as is the length of a vector. The fact that the length of a vector is preserved is just Parseval's theorem:

$$\sum_{j=0}^{n-1} |x_j|^2 = \sum_{j'=0}^{n-1} |x_{j'}|^2$$

Another way of looking at the unitary DFT is to note that in the above discussion, the DFT has been expressed as a Vandermonde matrix:

$$\mathcal{F} = \frac{1}{\sqrt{n}} \begin{bmatrix} \omega_n^{0\cdot 0} & \omega_n^{0\cdot 1} & \ldots & \omega_n^{0\cdot(n-1)} \\ \omega_n^{1\cdot 0} & \omega_n^{1\cdot 1} & \ldots & \omega_n^{1\cdot(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_n^{(n-1)\cdot 0} & \omega_n^{(n-1)\cdot 1} & \ldots & \omega_n^{(n-1)\cdot(n-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

where

$\omega_n = e^{-2\pi i/n}$

is a primitive nth root of unity. This matrix is a unitary matrix with

$$\det(\mathcal{F}) = 1$$

where det() is the determinant function and

$$\mathcal{F}^* = \mathcal{F}^{-1}$$

9. The real DFT

If $x_0, \ldots, x_{n-1}$ are real numbers, as they often are in the applications, then $f_j = f_{n-j}^*$, where the star denotes complex conjugation. Hence, the full information in this case is already contained in the first half (roughly) of the sequence $f_0, \ldots, f_{n-1}$. In this case, the "DC" element f_0 is purely real, and for even n the "Nyquist" element $f_{n/2}$ is also real, so there are exactly n non-redundant real numbers in the first half + Nyquist element of the complex output f. Using Euler's formula, the interpolating trigonometric polynomial can then be interpreted as a sum of sine and cosine functions.

2.4.3.6 Generalized DFT

It is possible to shift the transform sampling in time and/or frequency domain by some real shifts a and b, respectively. This is sometimes known as a generalized DFT (or GDFT) and has analogous properties to the ordinary DFT:

$$f_j = \sum_{k=0}^{n-1} x_k e^{-\frac{2\pi i}{n}(j+b)(k+a)} \qquad j = 0, \ldots, n-1$$

Most often, shifts of $\frac{1}{2}$ (half a sample) are used. While the ordinary DFT corresponds to a periodic signal in both time and frequency domains, $a = \frac{1}{2}$ produces a signal that is anti-periodic in frequency domain ($f_{j+n} = -f_j$) and vice-versa for $b = \frac{1}{2}$. Thus, the specific case of $a = b = \frac{1}{2}$ is known as an *odd-time odd-frequency* discrete Fourier transform (or O^2 DFT). Such shifted transforms are most often used for symmetric data, to represent different boundary symmetries, and for real-symmetric data they correspond to different forms of the discrete cosine and sine transforms.
The discrete Fourier transform can be viewed as a special case of the *z*-transform, evaluated on the unit circle in the complex plane.

2-D transform
For digital image processing, the 2-D transform is used to find the frequency content of an image. The transform is defined as:

$$F(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-2\pi i\left(\frac{ux}{M}+\frac{vy}{N}\right)} \qquad u = 0,\ldots,M-1; v = 0,\ldots,N-1$$

and the inverse transform is defined as:

$$f(x,y) = \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)e^{2\pi i\left(\frac{ux}{M}+\frac{vy}{N}\right)} \qquad x = 0,\ldots,M-1; y = 0,\ldots,N-1$$

where
$f(x,y)$ is a 2-D signal (e.g., an image) with x as the xth column of f; and y as the yth row of f
$F(u,v)$ is the 2-D frequency spectrum of $f(x,y)$
The form of the 2-D DFT can be simplified by using matrices
$F = Wf^T W$
where

$$F = \begin{bmatrix} F(0,0) & F(1,0) & \ldots & F(M-1,0) \\ F(0,1) & F(1,1) & \ldots & F(M-1,1) \\ \vdots & \vdots & \ddots & \vdots \\ F(0,N-1) & F(1,N-1) & \ldots & F(M-1,N-1) \end{bmatrix}$$

$$W = \begin{bmatrix} \omega_n^{0\cdot 0} & \omega_n^{0\cdot 1} & \dots & \omega_n^{0\cdot (n-1)} \\ \omega_n^{1\cdot 0} & \omega_n^{1\cdot 1} & \dots & \omega_n^{1\cdot (n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_n^{(n-1)\cdot 0} & \omega_n^{(n-1)\cdot 1} & \dots & \omega_n^{(n-1)\cdot (n-1)} \end{bmatrix}$$ (this is the unitary matrix shown above)

$$f = \begin{bmatrix} f(0,0) & f(1,0) & \dots & f(M-1,0) \\ f(0,1) & f(1,1) & \dots & f(M-1,1) \\ \vdots & \vdots & \ddots & \vdots \\ f(0,N-1) & f(1,N-1) & \dots & F(M-1,N-1) \end{bmatrix}$$

Matrix form derivation

The forward transform can be reformulated into matrix notation

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-2\pi i \left(\frac{ux}{M} \frac{vy}{N}\right)}$$

$$F(u,v) = \sum_{x=0}^{M-1} \left[\sum_{y=0}^{N-1} f(x,y) e^{-2\pi i \frac{vy}{N}} \right] e^{-2\pi i \frac{ux}{M}}$$

$$F(u,v) = \sum_{x=0}^{M-1} \underbrace{\left[\sum_{y=0}^{N-1} f(x,y) e^{-2\pi i \frac{vy}{N}} \right]}_{\mathcal{F}_y\{f(x,y)\}} e^{-2\pi i \frac{ux}{M}}$$

$F_v = Wf$

$$F(u,v) = \sum_{x=0}^{M-1} F_v(x,v) e^{-2\pi i \frac{ux}{M}}$$

$$F = W F_v^T$$

$F = W f^T W$

Equation 1 is the 2-D DFT definition from above.

Equation 2 is equation 1 with $e^{-2\pi i \frac{ux}{M}}$ pulled out of the inner summation.

Equation 3 shows that the inner summation is a 1-D DFT taken with respect to the y-dimension (the columns and denoted $\mathcal{F}_y\{f(x,y)\}$) of the signal $f(x,y)$.

Equation 4 is the matrix form of a 1-D DFT of the underbraced portion in equation 3.

Equation 5 is with $F_v(x,v)$ substituted for the underbraced portion of equation 3. $F_v(x,v)$ is the vth frequency of the xth column of $f(x,y)$. Collecting the column 1-D DFTs of $f(x,y)$ forms the matrix F_v.

Equation 6 is derived after realizing that equation 4 is a 1-D DFT of F_v^T .
Equation 7 is simply equation 4 substituted into equation 6 and noting that $W = W^T$ because W is symmetric.

In summation:
A column of F_v represents the 1-D DFT of the xth column of the signal $f(x,y)$. So, F_v can be created by combining the 1-D DFT column vectors taken with respect to each column of $f(x,y)$. Consequently, the transposition of F_v in $F = WF_v^T$ and pre-multiplication of W is the same as taking the 1-D DFT of the *rows* of F_v.

2.4.3.7 Applications

The DFT has seen wide usage across a large number of fields; we only sketch a few examples below (see also the references at the end). All applications of the DFT depend crucially on the availability of a fast algorithm to compute discrete Fourier transforms and their inverses, a fast Fourier transform.

Signal analysis
Suppose a signal $x(t)$ is to be analyzed. Here, t stands for time, which varies over the interval $[0,T]$, and, in the case of a sound signal, $x(t)$ is the air pressure at time t.The signal is *sampled* at n equidistant points to get the n real numbers $x_0 = x(0)$, $x_1 = x(h)$, $x_2 = x(2h)$, ..., $x_{n-1} = x((n-1)h)$, where $h = T/n$ and n is even.Then the discrete Fourier transform f_0, ..., f_{n-1} is computed and the numbers $f_{n/2+1}$, ..., f_{n-1} are discarded (they are redundant for real signals).Then f_0/n approximates the average value of the signal over the interval, and for every $j = 1, ..., n/2$, the argument $\arg(f_j)$ represents the phase and the modulus $|f_j|/n$ represents one half of the amplitude of the component of the signal having frequency j/T
The reason behind this interpretation is that the f_j are approximations to the coefficients occurring in the Fourier series expansion of $x(t)$. In general, the problem of using the DFT of discrete samples to approximate the Fourier transform of an infinite, continuous signal is called *spectral estimation*, and involves many more details than are described here. (For example, one often wants to window the data in order to reduce the distortion caused by the periodic boundary conditions implicit in the DFT.)

Data compression
The field of digital signal processing relies heavily on operations in the frequency domain (i.e. on the Fourier transform). For example, several lossy image and sound compression methods employ the discrete Fourier transform: the signal is cut into short segments, each is transformed, and then the Fourier coefficients of high frequencies, which are assumed to be unnoticeable, are discarded. The decompressor computes the inverse transform based on this reduced number of Fourier coefficients. (Compression applications often use a specialized form of the DFT, the discrete cosine transform.)

Partial differential equations

Discrete Fourier transforms, especially in more than one dimension, are often used to solve partial differential equations. The reason is that it expands the signal in complex exponentials e^{ikx}, which are eigenfunctions of differentiation: $d/dx\, e^{ikx} = ik\, e^{ikx}$. Thus, in the Fourier representation, a linear differential equation with constant coefficients is transformed into an easily solvable algebraic equation. One then uses the inverse DFT to transform the result back into the ordinary spatial representation. Such an approach is called a spectral method.

Multiplication of large integers

The fastest known algorithms for the multiplication of large integers or polynomials are based on the discrete Fourier transform: the sequences of digits or coefficients are interpreted as vectors whose convolution needs to be computed; in order to do this, they are first Fourier-transformed, then multiplied component-wise, then transformed back.

2.4.3.8 Some discrete Fourier transform pairs

In the following table ω_n stands for exp(- 2πi / n)

$x_j = \frac{1}{n}\sum_{k=0}^{n-1} f_k \omega_n^{-jk}$	$f_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk}$
a^j	$\frac{1 - a^n \omega_n^{kn}}{1 - a\omega_n^k}$
$\binom{n-1}{j}$	$(1 + \omega_n^k)^{n-1}$

2.4.3.9 Variants of the Fourier transform

Continuous Fourier transform

Most often, the unqualified term "Fourier transform" refers to the continuous Fourier transform, representing any square-integrable function $f(t)$ as a sum of complex exponentials with angular frequencies ω and complex amplitudes $F(\omega)$:

$$f(t) = \mathcal{F}^{-1}(F)(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} F(\omega) e^{i\omega t}\, d\omega.$$

This is actually the *inverse* continuous Fourier transform, whereas the Fourier transform expresses $F(\omega)$ in terms of $f(t)$; the original function and its transform are sometimes called a *transform pair*. A generalization of this transform is the fractional Fourier transform, by which the transform can be raised to any real "power".

When $f(t)$ is an even or odd function, the sine or cosine terms disappear and one is left with the cosine transform or sine transform, respectively. Another important case is where $f(t)$ is purely real, where it follows that $F(-\omega) = F(\omega)^*$. (Similar special cases appear for all other variants of the Fourier transform as well.)

Fourier series

The continuous transform is itself actually a generalization of an earlier concept, a Fourier series, which was specific to periodic (or finite-domain) functions $f(x)$ (with period 2π), and represents these functions as a series of sinusoids:

$$f(x) = \sum_{n=-\infty}^{\infty} F_n \, e^{inx},$$

where F_n is the (complex) amplitude. Or, for real-valued functions, the Fourier series is often written:

$$f(x) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} \left[a_n \cos(nx) + b_n \sin(nx)\right],$$

where a_n and b_n are the (real) Fourier series amplitudes.

Discrete Fourier transform

For use on computers, both for scientific computation and digital signal processing, one must have functions x_k that are defined over *discrete* instead of continuous domains, again finite or periodic. In this case, one uses the discrete Fourier transform (DFT), which represents x_k as the sum of sinusoids:

$$x_k = \frac{1}{n}\sum_{j=0}^{n-1} f_j e^{2\pi ijk/n} \qquad k = 0, \ldots, n-1$$

where f_j are the Fourier amplitudes. Although applying this formula directly would require $O(n^2)$ operations, it can be computed in only $O(n \log n)$ operations using a fast Fourier transform (FFT) algorithm, which makes FFT a practical and important operation on computers.

Other variants

The DFT is a special case of (and is sometimes used as an approximation for) the discrete-time Fourier transform (DTFT), in which the x_k are defined over discrete but infinite domains, and thus the spectrum is continuous and periodic. The DTFT is essentially the inverse of the

Fourier series.

These Fourier variants can also be generalized to Fourier transforms on arbitrary locally compact abelian topological groups, which are studied in harmonic analysis; there, one transforms from a group to its dual group. This treatment also allows a general formulation of the convolution theorem, which relates Fourier transforms and convolutions. Time-frequency transforms such as the short-time Fourier transform, wavelet transforms, chirplet transforms, and the fractional Fourier transform try to obtain

frequency information from a signal as a function of time (or whatever the independent variable is), although the ability to simultaneously resolve frequency and time is limited by a (mathematical) uncertainty principle

2.4.3.10 Applications

Fourier transforms have many scientific applications — in physics, number theory, combinatorics, signal processing, probability theory, statistics, cryptography, acoustics, oceanography, optics, geometry, and other areas. (In signal processing and related fields, the Fourier transform is typically thought of as decomposing a signal into its component frequencies and their amplitudes.) This wide applicability stems from several useful properties of the transforms:

The transforms are linear operators and, with proper normalization, are unitary as well (a property known as Parseval's theorem or, more generally, as the Plancherel theorem, and most generally via Pontryagin duality). The transforms are invertible, and in fact the inverse transform has almost the same form as the forward transform. The sinusoidal basis functions are eigenfunctions of differentiation, which means that this representation transforms linear differential equations with constant coefficients into ordinary algebraic ones. (For example, in a linear time-invariant physical system, frequency is a conserved quantity, so the behavior at each frequency can be solved independently.)

By the convolution theorem, Fourier transforms turn the complicated convolution operation into simple multiplication, which means that they provide an efficient way to compute convolution-based operations such as polynomial multiplication and multiplying large numbers. The discrete version of the Fourier transform can be evaluated quickly on computers using fast Fourier transform (FFT) algorithms.

Summary

The Discrete *Fourier Transform* (DFT) is used to produce frequency analysis of discrete non-periodic signals. The FFT is another method of achieving the same result, but with less overhead involved in the calculations. The various properties of DFT are also described and the applications of DFT. The variants of DFT are dealt.

Model Questions

1. Define Discrete Fourier Trnasform? Give its derivation.
2. What are the variants of fourier transform?
3. What are the properties of DFT ?

References:

1. http://www.imb-jena.de/ImgLibDoc/ftir/IMAGE_FTIR.html
2. http://topex.ucsd.edu/geodynamics/01fourier.pdf
3. http://en.wikipedia.org/wiki/Fourier_transform#Applications
4. http://en.wikipedia.org/wiki/Discrete_Fourier_transform
5. http://en.wikipedia.org/wiki/Discrete_Fourier_transform#Generalized_DFT

Asha Smitha. B, Centre for Biotechnology

Lesson 2.4.4

FAST FOURIER TRANSFORM

CONTENTS

Objective
2.4.4.1 Introduction
2.4.4.2 Why FFT
2.4.4.3 Background
2.4.4.4 Fast Fourier transform
2.4.4.5 The Cooley-Tukey algorithm
2.4.4.6 Other FFT algorithms
2.4.4.7 FFT algorithms specialized for real and/or symmetric data
2.4.4.8 Multidimensional FFT algorithms
2.4.4.9 Applications of the FFT
Summary
Model Questions
References

Objective

- To know what fast fourier transform is
- To understand how it is applied

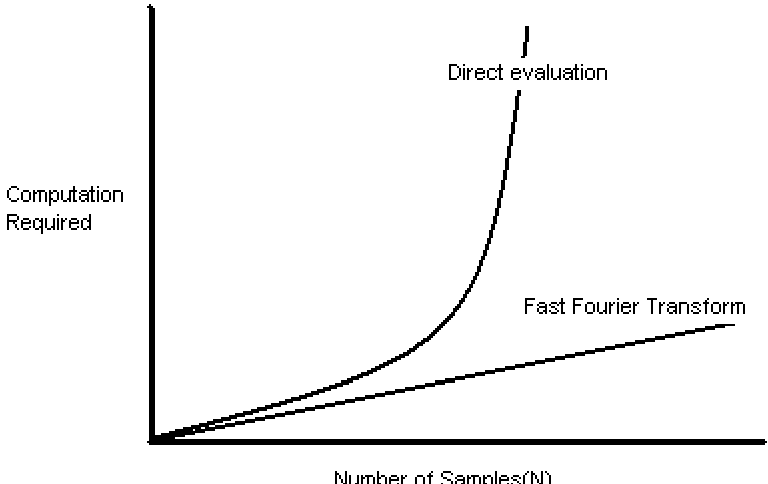
2.4.4.1 Introduction

In 1807, the French mathematician Joseph Fourier (1768-1830) submitted a paper to the Academy of Sciences in Paris. In it he presented a mathematical description of problems involving heat conduction. Although the paper was at first rejected, it contained ideas that would develop into an important area of mathematics named in honor, Fourier analysis. One surprising ramification of Fourier's work was that many familiar functions can be expanded in infinite series and integrals involving trigonometric functions. The idea today is important in modeling many phenomena in physics and engineering.

The mathematical operation that resolves a time series (for example, a recording of ground motion) into a series of numbers that characterize the relative amplitude and phase components of the signal as a function of frequency. Frequency. Number of cycles occurring in unit time.

2.4.4.2 Why FFT ?

If you look at the equation for the *Discrete Fourier Transform* you will see that it is complicated to work out as it involves many additions and multiplications involving complex numbers. Even a simple eight sample signal would require 49 complex multiplications and 56 complex additions to work out the DFT. At this level it is still manageable, however a realistic signal could have 1024 samples which requires over 20,000,000 complex multiplications and additions. As you can see the number of calculations required soon mounts up to unmanageable proportions.



The Fast Fourier Transform is a simply a method of laying out the computation, which is much faster for large values of *N*, where *N* is the number of samples in the sequence. It is an ingenious way of achieving rather than the DFT's clumsy P^2 timing.

The idea behind the FFT is the *divide and conquer* approach, to break up the original *N* point sample into two *(N / 2)* sequences. This is because a series of smaller problems is easier to solve than one large one. The DFT requires $(N-1)^2$ complex multiplications and *N(N-1)* complex additions as opposed to the FFT's approach of breaking it down into a series of 2 point samples which only require 1 multiplication and 2 additions and the recombination of the points which is minimal.

2.4.4.3 Background

The Fast Fourier Transform (FFT) is one of the most important family of algorithms in applied and computational mathematics. These are the algorithms that make most of signal processing, and hence modern telecommunications possible. The most basic divide and conquer approach was originally discovered by Gauss for the efficient interpolation of asteroidal orbits. Since then, various versions of the algorithm have been discovered and

rediscovered many times, culminating with the publishing of Cooley and Tukey's landmark paper, "An algorithm for machine calculation of complex Fourier series",

A Fourier series can sometimes be used to represent a function over an interval. If a function is defined over the entire real line, it may still have a Fourier series representation if it is periodic. If it is not periodic, then it cannot be represented by a Fourier series for all x. In such case we may still be able to represent the function in terms of sines and cosines, except that now the Fourier series becomes a Fourier integral.

The motivation comes from formally considering Fourier series for functions of period $2T$ and letting T tend to infinity.

Suppose

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{in\frac{\pi}{T}x}$$

and

$$c_n = \frac{1}{2T}\int_{-T}^{T} e^{-in\frac{\pi}{T}t} f(t)dt$$

Now, set

$$\omega_n = \frac{n\pi}{T} \quad \text{and} \quad \Delta\omega = \omega_n - \omega_{n-1} = \frac{\pi}{T}$$

and insert the integral formula for the Fourier coefficients:

$$\frac{1}{2\pi}\sum_{n=-\infty}^{\infty}\left(e^{i\omega_n x}\int_{-T}^{T} e^{-i\omega_n t} f(t)dt\right)\Delta\omega$$

The summation resembles a Riemann sum for a definite integral, and in the limit $T \to \infty$ ($\Delta\omega \to 0$) we might get

$$\frac{1}{2\pi}\int_{-\infty}^{\infty}\left(e^{i\omega x}\int_{-T}^{T}e^{-i\omega t}f(t)dt\right)d\omega \qquad x \in \mathbb{R}$$

This informal reasoning suggest the following definition

2.4.4.4 Fast Fourier transform

A **fast Fourier transform** (**FFT**) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. FFTs are of great importance to a wide variety of applications, from digital signal processing to solving partial differential equations to algorithms for quickly multiplying large integers. This article describes the algorithms, of which there are many; see discrete Fourier transform for properties and applications of the transform.

Let x_0, ..., x_{n-1} be complex numbers. The DFT is defined by the formula

$$f_j = \sum_{k=0}^{n-1} x_k e^{-\frac{2\pi i}{n}jk} \qquad j = 0, \dots, n-1.$$

Evaluating these sums directly would take $O(n^2)$ arithmetical operations. An FFT is an algorithm to compute the same result in only $O(n \log n)$ operations. In general, such algorithms depend upon the factorization of n, but (contrary to popular misconception) there are $O(n \log n)$ FFTs for all n, even prime n.

Since the inverse DFT is the same as the DFT, but with the opposite sign in the exponent and a $1/n$ factor, any FFT algorithm can easily be adapted for it as well.

2.4.4.5 The Cooley-Tukey algorithm

By far the most common FFT is the Cooley-Tukey algorithm. This is a divide and conquer algorithm that recursively breaks down a DFT of any composite size $n = n_1 n_2$ into many smaller DFTs of sizes n_1 and n_2, along with $O(n)$ multiplications by complex roots of unity traditionally called **twiddle factors**.

This method (and the general idea of an FFT) was popularized by a publication of J. W. Cooley and J. W. Tukey in 1965, but it was later discovered that those two authors had independently re-invented an algorithm known to Carl Friedrich Gauss around 1805 (and subsequently rediscovered several times in limited forms).

The most well-known use of the Cooley-Tukey algorithm is to divide the transform into two pieces of size $n / 2$ at each step, and is therefore limited to power-of-two sizes, but any factorization can be used in general (as was known to both Gauss and Cooley/Tukey). These are called the **radix-2** and **mixed-radix** cases, respectively (and other variants have their own names as well). Although the basic idea is recursive, most traditional implementations rearrange the algorithm to avoid explicit recursion. Also, because the Cooley-Tukey algorithm breaks the DFT into smaller DFTs, it can be combined arbitrarily with any other algorithm for the DFT.

2.4.4.6 Other FFT algorithms

There are other FFT algorithms distinct from Cooley-Tukey. For $n = n_1n_2$ with coprime n_1 and n_2, one can use the Prime-Factor (Good-Thomas) algorithm (PFA), based on the Chinese Remainder Theorem, to factorize the DFT similarly to Cooley-Tukey but without the twiddle factors. The Rader-Brenner algorithm (1976) is a Cooley-Tukey-like factorization but with purely imaginary twiddle factors, reducing multiplications at the cost of increased additions and reduced numerical stability. Algorithms that recursively factorize the DFT into smaller operations other than DFTs include the Bruun and QFT algorithms. (The Rader-Brenner and QFT algorithms were proposed for power-of-two sizes, but it is possible that they could be adapted to general composite n. Bruun's algorithm applies to arbitrary even composite sizes.) Bruun's algorithm, in particular, is based on interpreting the FFT as a recursive factorization of the polynomial $z^n - 1$, here into real-coefficient polynomials of the form $z^m - 1$ and $z^{2m} + az^m + 1$. Another polynomial viewpoint is exploited by the Winograd algorithm, which factorizes $z^n - 1$ into cyclotomic polynomials—these often have coefficients of 1, 0, or −1, and therefore require few (if any) multiplications, so Winograd can be used to obtain minimal-multiplication FFTs and is often used to find efficient algorithms for small factors. Indeed, Winograd showed that the DFT can be computed with only $O(n)$ multiplications, leading to a proven achievable lower bound on the number of irrational multiplications for power-of-two sizes; unfortunately, this comes at the cost of many more additions, a tradeoff no longer favorable on modern processors with hardware multipliers. In particular, Winograd also makes use of the PFA as well as an algorithm by Rader for FFTs of *prime* sizes. Rader's algorithm, exploiting the existence of a generator for the multiplicative group modulo prime n, expresses a DFT of prime size n as a cyclic convolution of (composite) size $n - 1$, which can then be computed by a pair of ordinary FFTs via the convolution theorem (although Winograd uses other convolution methods). Another prime-size FFT is due to L. I. Bluestein, and is sometimes called the chirp-z algorithm; it also re-expresses a DFT as a convolution, but this time of the *same* size (which can be zero-padded to a power of two and evaluated by radix-2 Cooley-Tukey FFTs, for example), via the identity $jk = -(j - k)^2 / 2 + j^2 / 2 + k^2 / 2$.

2.4.4.7 FFT algorithms specialized for real and/or symmetric data

In many applications, the input data for the DFT are purely real, in which case the outputs satisfy the symmetry

$$f_{n-j} = f_j^*,$$

and efficient FFT algorithms have been designed for this situation (see e.g. Sorensen, 1987). One approach consists of taking an ordinary algorithm (e.g. Cooley-Tukey) and removing the redundant parts of the computation, saving roughly a factor of two in time and memory. Alternatively, it is possible to express an *even*-length real-input DFT as a complex DFT of half the length (whose real and imaginary parts are the even/odd elements of the original real data), followed by O(n) post-processing operations.

It was once believed that real-input DFTs could be more efficiently computed by means of the Discrete Hartley transform (DHT), but it was subsequently argued that a specialized real-input DFT algorithm (FFT) can typically be found that requires fewer operations than the corresponding DHT algorithm (FHT) for the same number of inputs. Bruun's algorithm (above) is another method that was initially proposed to take advantage of real inputs, but it has not proved popular.

There are further FFT specializations for the cases of real data that have even/odd symmetry, in which case one can gain another factor of (roughly) two in time and memory and the DFT becomes the discrete cosine/sine transform(s) (DCT/DST). Instead of directly modifying an FFT algorithm for these cases, DCTs/DSTs can also be computed via FFTs of real data combined with O(n) pre/post processing.

Accuracy and approximations

All of the FFT algorithms discussed so far compute the DFT exactly (in exact arithmetic, i.e. neglecting floating-point errors). A few "FFT" algorithms have been proposed, however, that compute the DFT *approximately*, with an error that can be made arbitrarily small at the expense of increased computations. Such algorithms trade the approximation error for increased speed or other properties. For example, an approximate FFT algorithm by Edelman et al. (1999) achieves lower communication requirements for parallel computing with the help of a fast-multipole method. A wavelet-based approximate FFT by Guo and Burrus (1996) takes sparse inputs/outputs (time/frequency localization) into account more efficiently than is possible with an exact FFT. Another algorithm for approximate computation of a subset of the DFT outputs is due to Shentov et al. (1995). Only the Edelman algorithm works equally well for sparse and non-sparse data, however, since it is based on the compressibility (rank deficiency) of the Fourier matrix itself rather than the compressibility (sparsity) of the data.

Even the "exact" FFT algorithms have errors when finite-precision floating-point arithmetic is used, but these errors are typically quite small; most FFT algorithms, e.g. Cooley-Tukey, have excellent numerical properties. The upper bound on the relative error for the Cooley-Tukey algorithm is O($\varepsilon \log n$), compared to O($\varepsilon n^{3/2}$) for the naïve DFT formula (Gentleman and Sande, 1966), where ε is the machine floating-point relative precision. In fact, the root

mean square (rms) errors are much better than these upper bounds, being only O(ε √log *n*) for Cooley-Tukey and O(ε √*n*) for the naïve DFT (Schatzman, 1996). These results, however, are very sensitive to the accuracy of the twiddle factors used in the FFT (i.e. the trigonometric function values), and it is not unusual for incautious FFT implementations to have much worse accuracy, e.g. if they use inaccurate trigonometric recurrence formulas. Some FFTs other than Cooley-Tukey, such as the Rader-Brenner algorithm, are intrinsically less stable.

In fixed-point arithmetic, the finite-precision errors accumulated by FFT algorithms are worse, with rms errors growing as O(√*n*) for the Cooley-Tukey algorithm (Welch, 1969). Moreover, even achieving this accuracy requires careful attention to scaling in order to minimize the loss of precision, and fixed-point FFT algorithms involve rescaling at each intermediate stage of decompositions like Cooley-Tukey.

To verify the correctness of an FFT implementation, rigorous guarantees can be obtained in O(*n* log *n*) time by a simple procedure checking the linearity, impulse-response, and time-shift properties of the transform on random inputs (Ergün, 1995).

2.4.4.8 Multidimensional FFT algorithms

As defined in the multidimensional DFT article, the multidimensional DFT

$$f_{\mathbf{j}} = \sum_{\mathbf{k}=0}^{\mathbf{n}-1} e^{-2\pi i \mathbf{j}\cdot(\mathbf{k}/\mathbf{n})} x_{\mathbf{k}}$$

transforms an array $x_{\mathbf{k}}$ with a d-dimensional vector of indices $\mathbf{k} = (k_1 = 0 \ldots n_1 - 1, \cdots, k_d = 0 \ldots n_d - 1)$ by a set of d nested summations. Equivalently, it is simply the composition of a sequence of d one-dimensional DFTs, performed along one dimension at a time (in any order).

This compositional viewpoint immediately provides the simplest and most common multidimensional DFT algorithm, known as the **row-column** algorithm (after the two-dimensional case, below). That is, one simply performs a sequence of d one-dimensional FFTs (by any of the above algorithms): first you transform along the k_1 dimension, then along the k_2 dimension, and so on (or actually, any ordering will work). This method is easily shown to have the usual $O(N \log N)$ complexity, where $N = n_1 n_2 \cdots n_d$ is the total number of data points transformed. In particular, there are N / n_1 transforms of size n_1, etcetera, so the complexity of the sequence of FFTs is:

$$N/n_1 O(n_1 \log n_1) + \cdots + N/n_d O(n_d \log n_d) = O(N[\log n_1 + \cdots + \log n_d]) = O(N \log N).$$

In two dimensions, the $x_{\mathbf{k}}$ can be viewed as an $n_1 \times n_2$ matrix, and this algorithm corresponds to first performing the FFT of all the rows and then of all the columns (or vice versa), hence the name.

In more than two dimensions, it is often advantageous for cache locality to group the dimensions recursively. For example, a three-dimensional FFT might first perform two-dimensional FFTs of each planar "slice" for each fixed k_1, and then perform the one-dimensional FFTs along the k_1 direction. More generally, an (asymptotically) optimal cache-oblivious algorithm consists of recursively dividing the dimensions into two groups $(k_1, \cdots, k_{d/2})$ and $(k_{d/2+1}, \cdots, k_d)$ that are transformed recursively (rounding if d is not even) (see Frigo and Johnson, 2005). Still, this remains a straightforward variation of the row-column algorithm that ultimately requires only a one-dimensional FFT algorithm as the base case, and still has $O(N \log N)$ complexity. Yet another variation is to perform matrix transpositions in between transforming subsequent dimensions, so that the transforms operate on contiguous data; this is especially important for out-of-core and distributed memory situations where accessing non-contiguous data is extremely time-consuming.

There are other multidimensional FFT algorithms that are distinct from the row-column algorithm, although all of them have $O(N \log N)$ complexity. Perhaps the simplest non-row-column FFT is the vector-radix FFT algorithm, which is a generalization of the ordinary Cooley-Tukey algorithm where one divides the transform dimensions by a vector $\mathbf{r} = (r_1, r_2, \cdots, r_d)$ of radices at each step. (This may also have cache benefits.) The simplest case of vector-radix is where all of the radices are equal (e.g. vector-radix-2 divides *all* of the dimensions by two), but this is not necessary. Vector radix with only a single non-unit radix at a time, i.e. $\mathbf{r} = (1, \cdots, 1, r, 1, \cdots, 1)$, is essentially a row-column algorithm. Other, more complicated, methods include polynomial transform algorithms due to Nussbaumer (1977), which view the transform in terms of convolutions and polynomials products.

2.4.4.9 Applications of the FFT

The FFT algorithm tends to be better suited to analyzing digital audio recordings than for filtering or synthesizing sounds. A common example is when you want to do the software equivalent of a device known as a *spectrum analyzer*, which electrical engineers use for displaying a graph of the frequency content of an electrical signal. You can use the FFT to determine the frequency of a note played in recorded music, to try to recognize different kinds of birds or insects, etc. The FFT is also useful for things which have nothing to do with audio, such as image processing (using a two-dimensional version of the FFT). The FFT also has scientific/statistical applications, like trying to detect periodic fluctuations in stock prices, animal populations, etc. FFTs are also used in analyzing seismographic

information to take "sonograms" of the inside of the Earth. It is even used to read about Fourier methods used to analyze DNA sequences!

The main problem with using the FFT for processing sounds is that the digital recordings must be broken up into chunks of n samples, where n always has to be an integer power of 2. One would first take the FFT of a block, process the FFT output array (i.e. zero out all frequency samples outside a certain range of frequencies), then perform the inverse FFT to get a filtered time-domain signal back. When the audio is broken up into chunks like this and processed with the FFT, the filtered result will have discontinuities which cause a clicking sound in the output at each chunk boundary. For example, if the recording has a sampling rate of 44,100 Hz, and the blocks have a size n = 1024, then there will be an audible click every 1024 / (44,100 Hz) = 0.0232 seconds, which is extremely annoying to say the least.

Assume the buffer size is $n = 2^N$. On the first iteration, read n samples from the input audio, do the FFT, processing, and IFFT, and keep the resulting time data in a second buffer. Then, shift the second half of the original buffer to the first half (remember that n is a power of 2, so it is divisible by 2), and read $n/2$ samples into the second half of the buffer. Do the same FFT, processing, IFFT. Now, do a linear fade out on the second half of the old buffer that was saved from the first (FFT,processing,IFFT) triplet by multiplying each sample by a value that varies from 1 (for sample number $n/2$) to 0 (for sample number n - 1). Do a linear fade in on the first half of the new output buffer (going linearly from 0 to 1), and add the two halves together to get output which is a smooth transition. Note that the areas surrounding each discontinuity are virtually erased from the output, though a consistent volume level is maintained. This technique works best when the processing does not disturb the phase information of the frequency spectrum. For example, a bandpass filter will work very well, but you may encounter distortion with pitch shifting.

An open question in computational molecular biology is whether long-range correlations are present in both coding and noncoding DNA or only in the latter. To answer this question, we consider all 33 301 coding and all 29 453 noncoding eukaryotic sequences—each of length larger than 512 base pairs (bp—in the present release of the GenBank to determine whether there is any statistically significant distinction in their long-range correlation properties. Standard fast Fourier transform (FFT) analysis indicates that *coding* sequences have practically no correlations in the range from 10 bp to 100 bp (spectral exponent $\beta=0.00\pm0.04$, where the uncertainty is two standard deviations). In contrast, for *noncoding* sequences, the average value of the spectral exponent β is positive (0.16 ± 0.05), which unambiguously shows the presence of long-range correlations. We also separately analyze the 874 coding and the 1157 noncoding sequences that have more than 4096 bp and find a larger region of power-law behavior. We calculate the probability that these two data sets (coding and noncoding) were drawn from the same distribution and we find that it is less than 10^{-10}. We obtain independent confirmation of these findings using the method of detrended fluctuation analysis (DFA), which is designed to treat sequences with

statistical heterogeneity, such as DNA's known mosaic structure ("patchiness") arising from the nonstationarity of nucleotide concentration. The near-perfect agreement between the two independent analysis methods, FFT and DFA, increases the confidence in the reliability of our conclusion.

Summary

Fourier analysis of a periodic function refers to the extraction of the series of sines and cosines which when superimposed will reproduce the function. This analysis can be expressed as a Fourier series. The fast Fourier transform is a mathematical method for transforming a function of time into a function of frequency. Sometimes it is described as transforming from the time domain to the frequency domain. It is very useful for analysis of time-dependent phenomena.. Fast fourier transform can also be used for the DNA sequence analysis. The FFT also has scientific/statistical applications, like trying to detect periodic fluctuations in stock prices, animal populations, etc

Model Questions

1. What is meant Fast Fourier transform?Why do we use it ?
2. Wrire about FFT algorithms.
3. How is fast fourier transform applied?

References

1. http://mathworld.wolfram.com/FastFourierTransform.html
2. http://en.wikipedia.org/wiki/Fast_Fourier_transform
3. http://astron.berkeley.edu/~jrg/ngst/fft/fft.html

Author :-
Asha Smitha. B
Center for Biotechnology
Acharaya Nagarjuna University

Lesson 2.5.1

PROGRAMMING WITH C

Objective

2.5.1.1. Introduction
2.5.1.2. The origin of C
2.5.1.3. The character set
2.5.1.4. Constants, variables & keywords
2.5.1.5. Type declaration
2.5.1.6. Hierarchy of operations
2.5.1.7 Basic Structure of C Program
2.5.1.8. The first C program
2.5.1.9. Control instructions in C
2.5.1.10. Loops
2.5.1.11. The for loop
2.5.1.12. The goto statement
2.5.1.13. Functions
2.5.1.14. Datatypes in C
2.5.1.15. Arrays
Summary
Self-assessment questions
Reference books

Objectives

- → To know what are programming languages and their various generations.
- → The origin and fundamentals of C on hand.
- → To program using C programming language features like.
  - → Operators
  - → Control structures
  - → Loops
  - → Functions
  - → Arrays

2.5.1.1. Introduction

Computers are man made machines. They do only what they are told to do. Most computer systems perform their operations on a very primitive level. The basic operations of a computer system form what is known as the computer's `instruction set'. In order to solve a problem using a computer, it is must to express the solution to the problem in terms of the instructions of the particular computer. Man instructs a computer to perform given tasks or applications through computer programming languages.

A programming language or computer language is a standardized communication technique for expressing instructions to a computer. It is a set of syntactic and semantic rules used to define computer programs. A language enables a programmer to

precisely specify what data a computer will act open, how these data will be stored / transmitted, and precisely what actions to take under various circumstances.

A primary purpose of programming languages is to enable programmers to express their intent for a computation more easily than they could with a lower-level language or machine code. For this reason, programming languages are generally designed to use a higher-level syntax, which can be easily communicated and understood by human programmers. Programming languages are important tools for helping software engineers write better programs faster. During the last few decades, a large number of computer languages have been introduced have replaced each other, and have been modified / combined. The need for a significant range of computer languages is caused by the fact that the purpose of programming languages varies from commercial software development to scientific to hobby use; There are many special purpose languages, for use in special situations: PHP is a scripting language that is especially suited for web development. Perl is suitable for text manipulation. The C language has been widely used for developing of the operating system and compilers (so-called system programming). Programming languages make computer programs less dependent on particular machines or environments. This is because programming languages are converted into specific machine code for a particular machine rather than being executed directly by the machine.

Most languages can be either compiled or interpreted, but most are better suited for one than the other. In some programming systems, programs are compiled in multiple stages, into a variety of intermediate representations. Typically, later stages of compilation are closer to machine code than earlier stages.

If the program code is translated at runtime, with each translated step being executed immediately, the translation mechanism is spoken of as an interpreter. Interpreted programs run usually more slowly than compiled programs, but have more flexibility because they are able to interact with the execution environment. If the translation mechanism used is one that translates the program which takes the human-readable program text (called source code) as data input and supplies object code as output. The resulting object code may be machine code which will be executed directly by the computer's CPU, or it may be code matching the specification of virtual machine.

Computers communicate and are communicated using their own languages. The `natural' language of the computer is far less exotic than the human languages and has comparatively restricted vocabulary. Programming languages are divided into several categories-there is the machine level, ie., the binary level which the computer actually executes; there is the assembly level; ie., a machine-oriented version which allows the use of mnemonics. Both of these are termed low level because they mirror directly the machine's architecture. There are high level languages, also known as problem-oriented languages because they allow the programmer to write instructions more closely related to a class of problem rather than to the constraints of a given computer's architecture.

One such language is the C programming language that which possesses the powerful features of programming and supports conditional constructs, loop constructs, a rich list of operators and a variety of data structures as in third generation languages. There is a higher level yet which includes packages, such as word processing, database management and spreadsheets. These languages allow users to specify the information they want, in non-programming terms, and the package will produce the required results. There are closely related to Fourth Generation Languages (4 GLs).

Many hundreds of programming languages exist and it is commonly accepted to attach a descriptive label to each, eg: `scientific', `commercial'. However attempts to classify the various languages are rigid and should be used with caution.

2.5.1.2. The origin of C

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. Basic Combined Programming Language (BCPL), developed by Martin Richards at Cambridge University aimed to solve this problem by bringing CPL down to its basic good features. Around same time a language called B was written by Ken Thompson at AT & T's Bell Labs, as further simplification of CPL. But like BCPL, B too turned out to be very specific. C programming language was designed and written by a man named Dennis Ritchie. Ritchie inherited the features of B and BCPL, added some of his own and developed C. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL etc. Ritchie's main achievement is the restoration of the lost generality in BCPL and B, and still keeping it powerful. Possibly why C seems so popular is because it is reliable, simple and easy to use.

C is often called a middle-level computer language. This does not mean that C is less powerful, harder to use, or less developed than a high-level language, nor does it imply that C has the cumbersome nature of assembly language. As a middle-level language, C allows the manipulation of bits, bytes, and addresses the basic elements with which the computer functions.

Getting started with C

Communicatinng with a computer involves speaking the language the computers understands, which immediately rules out English as the language of communication with computer. However, there is close analogy between learning English language and learning C language.

2.5.1.3. The character set

A character denotes any alphabet, digit or special symbol used to represent information. Following table shows the valid alphabets, numbers and special symbols allowed in C:

Alphabets	A,B,……Y, Z
	A, b, ……., y, z

Digits 0,1,2,3,4,5,6,7,8,9

Special ~ ‘ ! @ # % ^ & * ( ) _ - + =

1 / { } [ ] : ; “ ‘ <> , . ? /

2.5.1.4. Constants, variables & keywords

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords. A constant is a quantity that doesn’t change. This quantity can be stored at a location in the memory of the computer. A variable can be considered as a name given to the location in memory where this constant is stored. The contents of a variable can change.

1. Types of constants

C constants can be divided into two major categories:

a. Primary constants
   - Integer constant
   - Real constant
   - Character constant

b. Secondary constant
   - Array
   - Pointer
   - Structure
   - Union
   - Enum etc.

For constructing these different types of constants certain rules have been laid down. These rules are as under:

a. Rules for constructing integer constants

a. An integer must have atleast one digit.

b. It must not have a decimal point.

c. It could be either positive or negative.

d. If no sign precedes an integer constant is assumed to be positive.

e. No commas or blanks are allowed within an integer constant

f. The allowable range for integer constants is –32767 to +32767.

Integer constants must fall within this range because the 1BM compatible microcomputers are usually 16 bit computers which cannot support a number falling outside the above mentioned range. For a 32 bit computer ofcourse the range would be much larger.

Eg: 426

+ 782

- 8000

b. Rubles for constructing real constants

Real constants are often called floating point constants. The real constants could be written in two forms, Fractional form and exponential form.

a. A real constant must have atleast one digit.
b. It must have a decimal point.
c. It could be either positive or negative.
d. Default sign is positive.
e. No commas or blanks are allowed within a real constant.

Eg: 426.0

- 32.76

In exponential form of representation, the real constant is represented in two parts. The part appearing before `e' is called mantissa, whereas the part following `e' is called exponent.

a. The mantissa part and the exponential part should be separated by a letter C.
b. The mantissa part may have a positive or negative sign.
c. Default sign of mantissa part is positive.
d. The exponent must have atleast one digit which must be a positive or negative integer. Default sign is positive.
e. Range of real constants expressed inn exponential form is –3.4e38 to 3.4e38.

Eg: + 3.2 e-5

4.1e8

-0.2e+3

c. Rules for constructing character constants

a. A character constant is either a single alphabet, a single digit or a single special symbol enclosed within single inverted commas both the inverted commas should point to the left. For example, `A' is a valid character constant whereas `A' is not.
b. The maximum length of a character constant can be 1 character.

Eg : `A’ `1’

`5’ `=’

2. Types of C variables

In C, a quantity which may vary during program is called a variable. Variable names are the names given to locations in the memory of computer where different constants are stored. These locations can contain integer, real or character constants. A constant stored in a location with a particular type of variable name can hold only that type of constant.

a. A variable name is any combination of 1 to 8 alphabets, digits or underscores.

b. The first character in variable name must be an alphabet.

c. No commas or blanks are allowed within a variable name.

d. No special symbol other than an underscore (as in gross-sal) can be used in a variable name.

Eg: si-int

Pop-e-89

These rules remain same for all the types of primary and secondary variables. C compiler is able to distinguish between the variable names, by making it compulsory to declare the type of any variable name to be used in a program. This type declaration is done at the beginning of the program.

3. C Key words

Keywords are the words whose meaning has already been explained to the C compiler (or in broad sense to the computer). The keywords cannot be used as variable names because doing so implies to assign a new meaning to the keyword, which is not allowed by the computer. The keywords are also called `Reserved words’. There are only 32 keywords available in C. Following is the list of keywords in C.

Auto	Double	If	Statie	Break	Else	Int	Struct	Case
Enum	Long	Switch	Char	Extern	Near	Typedy	Const	Float
Register	Union	Continue	Far	Return	Unsigned	Default	For	Short
Void	Do	Goto	Signed	While				

2.5.1.5. Type declaration

Type declaration is used to declare the type of variables being used in the program. Any variable used in the program must be declared before using it in any statement. The type declaration statement is usually written at the beginning of the C program.

Eg : int bas;

Float sal;

Char name;

It is very important to understand how the execution of an arithmetic statement takes place. First, the right hand side is evaluated using constants and the numerical values stored in the variable names. This value is assigned to the variable on the left hand side.

The following points has to be noted carefully.

a. C allows only one variable on left hand side of = that is, Z = k * l is legal, where as k * l = Z is illegal.
b. No operator is assumed to be present. It must be written explicitly. In the following example, the manipulation operator after b must be explicitly written.

   A = c,b,d (xy) has to be written as

   A = c*b*d* (x*y).
c. Arithematic operations can be performed on ints, floats is chars.

   Thus the statements,

   Char x, y;

   Int z;

   X = `a’,

   Y = `b’,

   Z = x+y; are perfectly valid,

Since the addition is performed on the ASC_{11} values of the characters and not on characters themselves. The ASC11 values of `a’ eg `b’ are 97 and 98 (refer the ASC11 values) and hence can definetly be added.

d. Unlike other high level languages, there is no operator for performing exponentiation operation.

   A = 3 **2;

   B = 3” 2; are valid
e. An arithmetic operation between an integer and integer always yields an integer result.

f. Operation between a real and real always yields a real result.

g. Operation between an integer and real always yields a real result.

Eg : 5/2 = 2 ; 2/5 = 0

5.0/2 = 2.5 ; 2.0/5 = 0.4

5.0/2.0 = 2.5 ; 2.0/5.0 = 0.4

2.5.1.6. Hierarchy of operations

All operators in C are ranked according to their precedence.

Priority	Operators	Description
1st	* 1%	Multiplication, division, modular division
2nd	+ -	Addition, subtraction
3rd	=	Assignment

a. In case of a tie between operations of same priority preference is given to the operator which occurs first.

b. Within a parantheses, if more than one set of parantheses are present, the operations within the innermost parantheses will be performed first, followed by the operations within the second innermost pair and so on.

A careless imbalance of the right and left parantheses is a common error.

2.5.1.7 Basic Structure of C Program

A C program may contain one or more sections shown.
The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later. The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants.
There are some variables that are used in more than one function. Such variables are called *global* variables and are declared in the *global* declaration section that is outside of all the functions. This section also declares all the user-defined functions.

Every C program must have one **main()** function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The

closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon.

The subprogram section contains all the user-defined functions that are called in the **main** function. User-defined functions are generally placed immediately after the **main** function, although they may appear in any order.
All sections, except the **main** function section may be absent when they are not required.

Documentation Section
Link Section
Definition Section
Global Declaration Section
main ( ) Function Section
{
Declaration part
Executable part
}
Subprogram section
Function 1
Function 2
-
-
Function n
(User-defined functions)

2.5.1.8. The first C program

Each instruction in a C program is written as a separate statement. Therefore a complete C program is nothing but statements in the same order in which they are to be executed; unless ofcourse the logic of the problem demands a deliberate `jump' or transfer of control to a statement which is out of sequence. The following rules are applicable to all C statements:

a. Blank spaces may be inserted between two words to improve the reliability of the statement. However, no blank spaces are allowed within a variable, constant or keyword.

b. C has no specific rules for the position at which a statement is to be written. That's why it is often called a free-form language.

c. Any C statement always ends with a ; (semicolon).

A Sample C Program

```
/* calculation of simple interest */
/* Author gekay Date: 25/12/1994 */
{
    int p, n;
    float r, si;
    P = 1000;
    N = 3;
    R = 8.5;
    Si = p * n * r / 100;
    Printf (" % f " , si);
}
```

→ Comment about the program should be enclosed within /* */. For example, the first two statements in the above program are comments. Any number of comments anywhere in the program can be written. Comments cannot be nested. A comment can be split over more than one line as in,

```
/* This is
   just a
   comment */
```

→ Any C program is nothing but a combination of functions. Main ( ) is one such function. Empty parantheses after `main' are necessary.

→ The set of statements belonging to a function are enclosed within a pair of braces. For eg:

```
Main ( )
{
  statement 1;
  statement 2;
}
```

→ Any variable used in the program must be declared before using it.

→ Print ( ) is a function which is used to print on the screen the value contained in a variable.

The general form of printf statement is,

Printf ("< format string >", <list of variables>);

< format string > could be,

% f for printing real values

% d for printing integer values

% c for printing character values

```
Eg: Printf ("% f", si);
      Printf ("% d %d%f%f", p, n, r, si);
      Printf ( " simple interest = Rs. % f", si) ;
```

The output of the last statement would be

Simple interest = Rs. 25

```
Printf ("Prin = % d in Rate = % f", p, r);
```

"In" is called new line and it takes the cursor to the next line. Therefore the output is split over two lines like

Prin = 1000

Rate = 8.5

In the above sample program, the values of p, n & r are supplied through keyboard, rather, a statement called scant should be used:

```
Main ( )
{
  int p, n;
  float r, si;
  print f ("Enter values of p, n, r");
  scanf ("%d %d %f", & p, &n, &r);
   si = p * n * r/100;
   printf ("%f", si);
}
```

The first printf ( ) statement outputs the message `enter the values of p,n,r' on the screen. The ambersand (&) before the variables in the scanf statement is a must, & is a pointer operator. The meaning and working of this pointer operator is the `address of' operator, that which returns the address of the variables. Here, in the above program, the p, n & r values are to be supplied through keyboard. The three values can be separted, while entering through keyboard, using a blank, a tab or even by entering each value on a new line.

2.5.1.9. Control instructions in C

C has three major decision making instructions, otherwise called conditional statements:

a. the if statement, (b) the if-else statement, and (c) the switch statement.

1. The if statement

C uses the keyword `if' to implement the decision control instruction. The general form is :

```
If (this condition is true)

   Execute this statement;
```

The keyword `if' tells the compiler that what follows, is a decision control instruction. The condition following the keyword if is always enclosed within a pair of paratheses. If the condition, whatever it is, is true, then the statement is executed. If the condition is not true then the statement is not executed; instead the program skips past it. As a general rule, the condition is expressed using C's `relational' operators. They allow to compare the values.

OPERATOR	MEANING
==	Equals
! =	Not equals
<	Less than
>	Greater than
<=	Lessthan or equals
> =	Greater than or equals

The requirement of more than one statement are to be executed if a given if condition is satisfied, then they must be placed within a pair of braces.

2. The if-else statement

The if statement will execute a single or multiple statement (s) when the condition following if is true, it doesn't do anythis if the condition is not satisfied.

This is what the purpose of the else statement. As the if statement, the default scope of else is also the statement immediately after the else. To override, a pair of braas are used. The format would be:

```
If (condition)
{
  statement 1;
  statement 2;
}
else
{
  statement 3;
  statement 4;
}
```

3. Nested if-elses

It is perfectly alright to write an entire if-else construct within either the body of the if statement or the body of an else statement. This is called `nesting' of ifs. Note that the second if-else construct is nested in the first else statement. If the condition in

the first if statement is false then the condition in the second if statement is checked. If it is false as well, the final else statement is executed.

4. Logical operators

C allows usage of three logical operators.

a. & & (read as AND)
b. 11 (OR)
c. ! (NOT)

They are composed of and are used as two symbols together. The first two operators, & & and 11, allow two or more conditions to be combined in an if statement. The & & operator is used to combine two conditions in which if one of the conditions evaluate to false then the whole condition of the if statement is treated as false. And when 11 is used, any one of conditions evaluates to true then the whole condition is treated as true. The third operator is the NOT operator written as !. It reverses the value of the expression it operates on; it makes a true expression false and a false expression true.

Eg : (1) if (a>b & & b>c)

That implies the condition is true only when a is greater than be and b is greater than c.

Eg : if (a>10 11 a>5)

A is greater than 10 and it can be true when the a is just greater than 5.

Eg : ! (y<10)

Implies "not y less than". In other words

The NOT operator is often used to reverse the logical value of a single variable.

5. The conditional operators

The conditional operators ? and : are sometimes called ternary operators since they take three arguments.

The general form is;

Expression 1 ? expression 2 : expression 3

This implies that : "if expression 1 is true (that is, if its value is non-zero), then the value returned will be expression 2 otherwise the value returned will be expression 3".

Eg : int x, y;

Scanf ("%d", & x);

Y = (x>5 ? 3:4);

This statement will store 3 in y if x is greater than 5, otherwise it will store 4 in y.

2.5.1.10. Loops

The versatility of the computer lies in its ability to perform a set of instructions repeatedly. Repeating some portion of the program either a specific number of times or until a particular condition is being satisfied. This repetitive operation is done through a loop control structure. There are three methods by way of which looping is done.

a. Using a for statement

b. Using a while statement
c. Using a do-while statement

1. The while loop

In programming, it is often a case of doing something a fixed number of times. The while loop is ideally suited for such cases. In a while block of statements, the condition is evaluated first and, if it is true, the loop body is executed. The parantheses after the while contains a condition. So long as this condition remains true all statements within the body of the while loop keep getting executed repeatedly. After execution of the loop body once / everytime, the condition in the while statement is evaluated again. This repeats until the condition becomes false. The format would be :

```
While (condition)
{
  statement 1;
  statement 2;
}
```

The statements within the while loop's condition may use relational or logical operators.

```
While (I<=10)
While (j>10 & & (b<15 \\ c < 20) )
```

As a rule the while must test a condition that will eventually become false, otherwise the loop would be executed forever, indefinitely.

Eg:

```
Int I = 1;
While (I<=10)
Printf (“%d”, I);
```

This is an indefinite loop, since ; remains equal to 1 forever.

→ No semicolon or anytermination are not to be used after the while condition and before the while block begins.

2. More operators

→ The increment operator(++) increments the value of preceeding / succeeding variable to which it is used.

Eg: I++ → I is incremented by 1.

→ The decrement operator (--) decrements the value of the variable by 1.

Eg: 1 -- → I is decreased by 1.

→ += is a compound assignment operator.

Eg : I = I+1;

Can be written as

I + = 1;

Similarly other compound assignment operators are -=, *=, /= and %=.

2.5.1.11. The for loop

The for loops allows to specify three conditions about the loop in a single line.

a. setting a loop counter to an initial value.
b. Testing the loop counter to determine whether its value has reached the number of repetitions desired.
c. Increasing the value of loop counter each time the program segment within the loop has been executed.

The general form of for statement is as under:

```
For (intialise counter; test counter; increment counter)
{
   do this;
    statements;
   }
```

Eg: int count;

```
For (count =1 ; count < = 3; count = count + 1)
{
   statement 1;
   statement 2;
    }
```

→ When the for statement is executed for the first time, the value of count is set to an initial value 1.

→ Now the condition count <=3 is tested. Since count is 1 the condition is satisfied and the body of the loop is executed for once. And after that, upon reaching the closing brace of for, the control gets back to the for statements, where the third statement of the for condition is evaluated and incremented, by 1.

→ Again the test is performed to check whether the value is <3 or not and the cycle goes on.

→ When count reaches the value 4 the control exists from the loop and is transferred to the statement (if any) immediately after the body of for.

The flowchart, further clarryfying the concept:

```
Eg: main ( )
{
   int I;
   for ( I = 1; I<=10; I=I+1)
    printf (“%d in”, I);
 }
```

prints numbers 1 to 10

- → The way if statements can be nested, similarly whiles and fors can also be nested.
- → In the for loop, multiple initializations can be done. That is, more than variables can be initialized with in the for loop

  Eg : for (I=1, j=2, j<=10; j++)
- → A break is usually used when the situations arise that the control has to jump out of the loop instantly, without waiting to get back to the conditional test. When the keyword break is encountered inside any C loop, control automatically passes to the first statement after the loop.
- → A continue statement allows to take the control to the beginning of the loop, bypassing the statements inside the loop which have not yet been executed. When the keyword continue is encountered inside any C loop, control automatically passes to the beginning of the loop. A continue is usually associated with an if.

4. The do-while loop

The do-while loop has a format of:

```
Do
{
  statement 1;
  statement 2;
} while (condition is true);
```

There is a minor difference between the working of while and do-while loops. The place where the condition is tested. `While' tests the condition before executing any of the statements within the while loop. The do-while tests the condition after having executed the statements within the loop. This means that do-while would execute its statement atleast once, even if the condition fails for the first time itself.

Break and continue are used with do-while just as they would be in a while or a for loop.

5. The switch statement

The control statement which allows the user to make a decision from a number of choices is called a switch, or more correctly a switch-case-default, since these three keywords go together to make up the control statement.

```
Switch (integer expression)
{
   case constant 1;
        do this;
   case constant 2;
        do this;
    case constant 3;
        do this;
```

```
    default;
        do this;
}
```

The integer expression following the keyword switch is any C expression that will yield an integer value. The keyword case is followed by an integer or a character constant that which different for each case. First, the integer expression following the keyword switch is evaluated. The value it gives is then matched, one by one, against the constant values that follow the case statements. When match found, the statements following that case are executed, and all subsequent case and default statements as well. If no match found, only the statements following the default are executed.

Eg :

```
Main ( )
{
  int I =2;
  switch (i)
{
  case 1:
        Printf ("I am in case 1 in");
   Case 2:
        Printf ("I am in case 2 in");
  Case 3:
        Printf ("I am in case 3 in");
   Default :
        Printf ("I am in default in");
}
}
```

The output would be
I am in case 2
I am in case 3
I am in default.

If the requirement is to print only the statements of the matched case, then break statement is to be used at the end of every case. No need fro a break after the default, since the control comes to the end anyway. For better understanding:

- → The order of the case number can be either ascending or descending or even without any order even.
- → Char values in case and switch can also be used instead of integer values.
- → Unlike if and else, no need to enclose multiple statements to be executed within braces.

2.5.1.12. The goto statement

The goto statements make a computer or C programmer's life miserable. The reason is programs become unreliable, unreadable and hard to debug. A goto statement can cause program control to end up almost anywhere in the program.

- → When goto's are used, it is never sure how a point is reached in the code. They obscure the flow of control.

2.5.1.13. Functions

A function is a self-contained block of statements that perform a coherent task of some kind. A task that which is always performed exactly in the same way everytime.

Eg:

```
Main ( )
{
  message ( );
  Printf ( "in this is after the function");
}
  message ( )
{
  printf (" in From the message function");
}
```

The output would be:

From the message function

This is after the function

That is, from the main ( ) the function message is called. The activity of main ( ) falls asleep for a while and the message ( ) function is waked up, which is defined after or at the end of the program. The control returns to main ( ) and continues to execute the rest of the statements. Main ( ) is called the calling function and message ( ) the called function.

The general format of a function would be :

```
Function (arg 1, arg 2, arg 3)
        Type arg1, arg2, arg3;
        {
          statement 1;
          statement 4;
        }
```

Arguments are the values passed from one function to the other. Before beginning with the statements in the functions it is necessary to declare the type of the arguments through type declaration statements.

- → Any C program contains at least one function.
- → If a program contains only one function, it must be main ( ), because the program execution always begins with main ( ).
- → There is no limit on the number of functions that might be present in a C program.
- → Each function is called in the sequence specified by the function calls in main ( ).
- → A function gets called when the function name is followed by a semicolon.
- → A function is defined when the function name is followed by a pair of baraces and some statements within them.
- → Any function can be called from any function. Even a function can call itself from within. Such a process is called `recursion'. But a function cannot be defined another function.
- → There are basically two types of functions.

Library functions Eg: printf ( ) , scanf ( ) etc.

User defined functions Eg: message ( ).

Library functions are commonly required functions grouped together and stored in what is called a library. The library is present on the disk and is written for the users. Almost always a compiler comes with a library of standard functions.

User defined functions are defined entirely by the user.

- → Writing functions avoids rewriting the same code over and over.
- → The mechanism used to convey information to the function is the `argument'.

Eg :
```
main ( )
{
 int a, b, c sum;
 printf ("Enter 3 numbers");
 scanf ("%d %d %d", &a, &b, &c);
 sum = calsum (a,b,c);
 printf ("in sum = % d", sum);
}
 calsum (x, y, z)
 int x, y, z;
{
 int d;
 d = x+y+z;
  return (d);
}
```

The output would be:

```
Enter 3 numbers 10  20  30
Sum = 60
```

- → The values of a,b &c are passed in to the function calsum ( ). These variables a, b & c are called `actual arguments and the variables x, y & z are called `formal arguments'. They number of arguments can be passed to the function, the type, order and number of the actual and formal arguments must be same. The compiler treats x, y & z variables as different, though the same names a, b & c are used, as they are in a different function.
- → The return statement serves two purposes.
  1. The control is immediately transferred to the calling function.
  2. To return a value from the called function.
- → Whenever the control returns from a function, some value is definitely returned.
- → `void' keyword placed before any function name at its definition, doesnot return any value.
- → Return function can return only one value at a time.
- → A value of a variable changed in the formal arguments does not affect the value in the actual argument.
- → When a function is called with values of the variables as arguments then it is said to be `calls by value'. Instead when the location

number (address) of a variable is passed as an argument, then it is said to be `call by reference'

→ A pointer points towards the location number, of a variable.

2.5.1.14. Datatypes in C

As seen earlier, it is necessary to define or declare type of data being used in a program. The int or the integer datatype has two classification long and short. The short is the ordinary int used.

Whereas, the long int occupies four bytes of memory and its range would be –2147483648 to +2147483647. They are declared as:

Long I;

Int j; Here I is a long integer variable, j is a short or ordinary int.

Integer variables can be further classified into signed and unsigned, basing on the sign. When the variable is declared as: unsigned int num-students; the range of permissible integer values will shift from the range –32768 to +32767 to the range 0 to 65535. Unsigned integers can be short and long. Unsigned long integer would be 0 to 4294967295.

The way there are signed and unsigned ints there are signed and unsigned chars also. A signed char is same as our ordinary char and ranges from –128 to +127 An unsigned clear from 0 to 255.

A float occupies four types in memory and ranges from –3.4e38 to + 3.4e38. C offers a double data type which occupies 8 bytes in memory and has range from –1.7e308 to + 1.7e308. A double variable can be declared as: double population;

There exists even a long double which can range from –1.7e4932 to +1.7e4932. A long double occupies 10 bytes in memory.

2.5.1.15. Arrays

An – array is a collective name given to a group of `similar quantities'. It would be easier to handle one variable than handling 100 different variables. The similar quantities could be percentage of marks, of 100 students, or salaries etc. The similar elements could all be ints, or all floats or all chars etc. Usually array of characters is called a string.

1. Array declaration

int marks (30);

int specifies the type of variable, marks the variable name and (30) tells how many elements of type int will be in our array. The bracket ([ ]) tells the compiler that it is an array.

2. Entering data into an array

Eg:

For (I=0; I<29; I++)

```
{
 Printf ("in enter marks");
 Scanf ("%d, & marks [I]);
}
```

The for loop, one by one takes inn the marks. The count of I starts from 0. The 1st element of the array would be marks (0) and second marks (1) and so on, until marks (29) not marks (30) as the count starts from 0. Another way of array initialization would be

```
Int num (6) = {2,4,12,5, 45, 5};
Float n ( ) = {12.3, 34.2, -23.4, -11.3};
```

Summary

The C programming language started at AT & T's Bell laboratories, called the middle level language, has its own characterset that which includes alphabets, numbers and special symbols, keywords, constants, variables and flexible datatypes and the opportunity for the programmer to define uses defined functions. Its control structures allows the user to define various conditions and checks to the task to refine it. Arrays, a collection of similar elements is the feature of C programming languages that enables to task with large numbers of data programming pleasure is offered by C programming language with its salient features.

Self assessment questions

1. Write a C program to input two numbers through the keyboard into two locations C and D and swap the contents of C & D, using a third variable.
2. Write a C program to print all prime numbers from 1 to 300.

   (Hint: Use nested loops, break & continue).
3. Write a program to generate all combinations of 1, 2 & 3 using for loop.
4. Write a menu driven C program to perform deposit, withdrawl and checking balance of bank transactions using functions and switch case.
5. Write a C program to calculate the percentages, grade and separate the failed students of 50 students of a class, using arrays.

Reference books

1. Programming in C - Stephen G. Kochan
2. Let us C - Yashwant Kanetkar
3. Learner's guide to `C' programming - NIIT.
4. Programming with `C' - Byron & Gottfried.

Asha Smitha. B.
Research Scholar
Center for Biotechnology
Acharya Nagarjuna University.

Lesson 2.5.2

PROGRAMMING WITH HTML

Contents

Objective
2.5.2.1 Introduction
2.5.2.2 A brief history of HTML
2.5.2.3 Authoring documents with HTML 4
2.5.2.4 HTML Document Structure
2.5.2.5 HTML element overview
2.5.2.6 HTML TAGS
Summary
Model Questions
References

Objective

- To study and know the use of various HTML tags.
- To understand the difference between HTML and DHTML.
- To improve knowledge in creating efficient web pages.

2.5.2.1 Introduction

To publish information for global distribution, one needs a universally understood language, a kind of publishing mother tongue that all computers may potentially understand. The publishing language used by the World Wide Web is HTML (from HyperText Markup Language).

- **Hyper** is the opposite of linear. It used to be that computer programs had to move in a linear fashion. This before this, this before this, and so on. HTML does not hold to that pattern and allows the person viewing the World Wide Web page to go anywhere, any time they want.
- **Text** is what you will use. Real, honest to goodness English letters.
- **Mark up** is what you will do. You will write in plain English and then mark up what you wrote. More to come on that in the next Primer.
- **Language** because they needed something that started with "L" to finish HTML and Hypertext Markup *Louie* didn't flow correctly. Because it's a language, really -- but the language is plain English.

2.5.2.2 A brief history of HTML

HTML was originally developed by Tim Berners-Lee while at CERN, and popularized by the Mosaic browser developed at NCSA. During the course of the 1990s it has blossomed with the explosive growth of the Web. During this time, HTML has been extended in a number of ways. The Web depends on Web page authors and vendors sharing the same conventions for HTML. This has motivated joint work on

specifications for HTML. HTML 2.0 (November 1995) was developed under the aegis of the Internet Engineering Task Force (IETF) to codify common practice in late 1994. HTML+ (1993) and HTML 3.0 (1995) proposed much richer versions of HTML. Despite never receiving consensus in standards discussions, these drafts led to the adoption of a range of new features. The efforts of the World Wide Web Consortium's HTML Working Group to codify common practice in 1996 resulted in HTML 3.2 (January 1997).

Most people agree that HTML documents should work well across different browsers and platforms. Achieving interoperability lowers costs to content providers since they must develop only one version of a document. If the effort is not made, there is much greater risk that the Web will devolve into a proprietary world of incompatible formats, ultimately reducing the Web's commercial potential for all participants. HTML has been developed with the vision that all manner of devices should be able to use information on the Web: PCs with graphics displays of varying resolution and color depths, cellular telephones, hand held devices, devices for speech for output and input, computers with high or low bandwidth, and so on.

HTML, or Hyper Text Markup Language is designed to specify the logical organization of a document, with important hypertext extensions. It is *not* designed to be the language of a WYSIWYG word processor such as Word or WordPerfect. This choice was made because the same HTML document may be viewed by many different "browsers", of very different abilities. HTML 4 extends HTML with mechanisms for style sheets, scripting, frames, embedding objects, improved support for right to left and mixed direction text, richer tables, and enhancements to forms, offering improved accessibility for people with disabilities.

2.5.2.3 Authoring documents with HTML 4

HTML instructions divide the text of a document into blocks called *elements*. These can be divided into two broad categories -- those that define how the BODY of the document is to be displayed by the browser, and those that define information `about' the document, such as the title or relationships to other documents.

HTML gives authors the means to:

- Publish online documents with headings, text, tables, lists, photos, etc.
- Retrieve online information via hypertext links, at the click of a button.
- Design forms for conducting transactions with remote services, for use in searching for information, making reservations, ordering products, etc.
- Include spread-sheets, video clips, sound clips, and other applications directly in their documents.

Text Block Elements:

As mentioned on the previous page, the BODY element contains all the displayed content of a document. Structurally, the document content is organized into blocks of text, such as paragraphs, lists, headings, paragraphs, block quotations, and so on. These are generically called *block elements*, since they "block" chunks of text together into logical units. Block elements can often contain other blocks -- for example, a list item can contain paragraphs or block quotations, so that these elements can often nest together.

The block-level elements are:
Hn (Headings)

- P
- ADDRESS
- BLOCKQUOTE
- PRE
- HR
- FORM
- TABLE
- DIV

Character-Level Elements
Then are what I call character-level elements, namely line breaks (BR) and images (IMG). These are treated much like characters, and can appear wherever there is a character in a document.

2.5.2.4 HTML Document Structure

HTML documents are structured into two parts, the HEAD, and the BODY. Both of these are contained within the HTML element -- this element simply denotes this as an HTML document.

The head contains information about the document that is not generally displayed with the document, such as its TITLE. The BODY contains the body of the text, and is where you place the document material to be displayed. Elements allowed inside the HEAD, such as TITLE, are not allowed inside the BODY, and vice versa.

Some of the most common elements used in HTML documents are listed here:

- The <HTML>, <HEAD>, and <BODY> tag pairs are used to structure the document.
- The <TITLE> and </TITLE> tag pair specifies the title of the document.
- The <H1> and </H1> header tag pair creates a headline.
- The <HR> element, which has no end tag, inserts a horizontal rule, or bar, across the screen.
- The <P> and </P> paragraph tag pair indicates a paragraph of text.

Strucutre

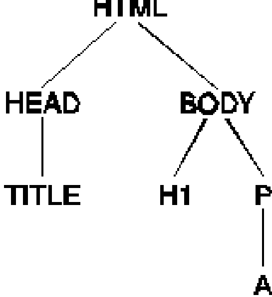
```
<! Doctype HTML PUBLIC “-//W3C/DTD HTML 4.0 Transational // EN”>
<HTML>
<HEAD>
<TITLE> Document title </TITLE>
   ..... Other supplementary information goes here…
</HEAD>
<BODY>
.... Marked - up text goes here ….
</BODY>
</HTML>
```

HTML Rules and Guidelines

The following are some rules to remember when writing HTML.

- HTML documents are structured documents. The structure of HTML Document is specified by a document type definition (DTD). A DTD defines what elements a document can contain, their relationships to one another inside a document, and their possible attributes and values. If the elements in a particular HTML document, agree with this formal definition, the document is said to be valid.
- Element name are not case-sensitive. An element such as <hTml> is equivalent to <html> or <HTML>.
- Attribute names are not case-sensitive.
- Attribute values may e case-sensitive, especially when it refers to a file.
- Attribute values should be quoted. The actual attribute value may contain spaces or special characters if it is enclosed by quotes.
- Element names cannot contain spaces. Browsers treat the first space encountered inside an element as the end of an element's name and the beginning of its attributes.
- Browsers collapse and ignore space characters in HTML content. Browsers collapse any sequence of spaces, tabs and returns in an HTML document into a single space character. These characters convey no formatting information, unless they occur inside a special preformatting element, such as <PRE>, which preserves their meaning. Extra space can be liberally used within HTML document to make it more legible.
- HTML documents may contain comments which are denoted by start value <!--and an end value-->. Comments may be many lines.
- Elements should nest. Any element that starts within a section enclosed by another element must also end within that section.
- Browsers ignore unknown elements and attributes.

2.5.2.5 **HTML element overview**



The HTML element starts with a **start tag**: E.g. <H1>
The HTML element ends with an **end tag**: </H1>

HTML tags are used to mark-up HTML **elements**. HTML tags are surrounded by the **two characters < and >.** The surrounding characters are called **angle brackets**.

The purpose of the <H1> tag is to define an HTML element that should be displayed as a heading.This HTML element starts with the start tag <body>, and ends with the end tag </body>.

The purpose of the <body> tag is to define the HTML element that contains the body of the HTML document. Tags can have attributes.

- Attributes can provide additional information about the HTML elements on your page.
- Attributes always come in name/value pairs like this: name="value".
- Attributes are always added to the start tag of an HTML element.

The most important tags in HTML are tags that define headings, paragraphs and line breaks.

2.5.2.6 HTML TAGS

- **<HTML> </HTML>**

*Attributes***:** VERSION=string
The HTML tag is the outermost tag. It is not required and may safely be omitted. It indicates that the text is HTML (the version can be indicated with the optional VERSION attribute), but this information is almost never used by servers or browsers.

- **Headings**

Headings are defined with the <h1> to <h6> tags. <h1> defines the largest heading. <h6> defines the smallest heading.
<h1>This is a heading</h1>
<h2>This is a heading</h2>
<h3>This is a heading</h3>
<h4>This is a heading</h4>
<h5>This is a heading</h5>
<h6>This is a heading</h6>
HTML automatically adds an extra blank line before and after a heading.

- **Paragraphs**

Paragraphs are defined with the <p> tag.
<p>This is a paragraph</p>
HTML automatically adds an extra blank line before and after a paragraph.

- **Line Breaks**

The <br> tag is used when you want to end a line, but don't want to start a new paragraph. The <br> tag forces a line break wherever you place it.<p>This <br> is a para
graph with line breaks</p>. The <br> tag is an empty tag. It has no closing tag.

- **<ADDRESS> </ADDRESS>**

The ADDRESS tag should be used to enclose contact information, addresses and the likes. It is often rendered with a slightly indented left margin and italics. Appearance:

<A HREF=URL> </A>. Attributes: HREF=URL, NAME=string, REL=string, REV=string, TITLE=string

The anchor tag is the "glue" for hypertext documents. The enclosed text and/or image(s) will be selectable by the user, and doing so will take the user to the location specified in the HREF attribute. The TITLE attribute can be used to provide a description of that location, which is displayed by some browsers when the mouse moves over the URL.
The NAME attribute is used to set up "named anchors." The enclosed text will be marked as a "target" to which a browser can jump directly. For example, if you have "<A NAME="toc">Table of Contents</A>" somewhere in the document, and the user selects the URL . REL and REV are not widely used, although these attributes were already present. Be sure to close the quotes around the value in HREF. a hyperlink with an unclosed quote may not work correctly. The A element used with the NAME attribute requires a closing tag and non-empty content.

- **<BODY> </BODY>**

Attributes: BACKGROUND=URL, BGCOLOR=#RRGGBB, TEXT=#RRGGBB, LINK=#RRGGBB VLINK=#RRGGBB, ALINK=#RRGGBB.

The BODY tag contains the actual contents of the document. The attributes contain the appearance of the document. The BACKGROUND attribute should point to the location of an image, which is used as the (tiled) background of the document. The other attributes set the colors for the background, text, links, visited links and active (currently being selected) links, using the order above.

- **<B> </B>**

B is used to indicate that the enclosed text must be rendered in a bold typeface.

- **<U> </U>**

U is used to indicate the enclosed text should be underlined.

- **<I> </I>**

I is used to indicate that the enclosed text must be rendered in a italic (slanted) typeface

- **<CENTER> </CENTER>**

It is used to indicate that large blocks of text should appear centered.

- **<FORM ACTION=URL> </FORM>**

Attributes: ACTION=URL, METHOD=get|post, ENCTYPE=string.

- <FORM ACTION=URL> </FORM>
  ACTION=URL, METHOD=get|post, ENCTYPE=string**.**

Forms allow a person to send data to the WWW server. FORM has one required attribute, ACTION, specifying the URL of a CGI script which processes the form and sends back feedback. There are two methods to send form data to a server. GET, the default, will send the form input in an URL, whereas POST sends it in the body of the submission. The latter method means you can send larger amounts of data, and that the URL of the form results doesn't show the encoded form. A form should always have at least one submit button. This can be done with <INPUT TYPE=submit

NAME=submitit> or with an image: <INPUT TYPE=image NAME=submitit>.More than one submit button is legal. If each submit button has a unique NAME attribute, the name of the selected submit button is sent along with the rest of the form input. This allows the parsing script to determine which button was pressed.

- **<HEAD> </HEAD>**

The HEAD part of the document provides information about the document. It should not contain text or normal markup. If a browser encounters such markup, it will assume it has arrived in the BODY section of the document already

- **<TITLE> </TITLE>**

Each document must have exactly one TITLE element. This element provides the title of the document. It is usually displayed at the top of the browser's window, but also used to label a bookmark entry for the document and as a caption in search engine results.

- **<HR>**

Attributes: ALIGN=left|right|center, NOSHADE, SIZE=n, WIDTH=n|p%

HR is used to draw horizontal rules across the browser window. None of the attributes for HR existed in HTML 2, so they may not be supported by all browsers. This can produce bizarre effects if you are using multiple HRs in a row to produce growing or shrinking "stripes".

- **<INPUT TYPE=x NAME=y>**

Attributes : TYPE=text|password|checkbox|radio|submit|reset|file|hidden|image, NAME=string, VALUE=string, CHECKED, SIZE=n, MAXLENGTH=n, SRC=URL, ALIGN=top|middle|bottom|left|right

The INPUT tag is probably the most useful tag inside forms. It can generate buttons, input fields and checkboxes. In all cases, the NAME attribute must be set.

TYPE=text

> This generates a input field, where the user can enter up to MAXLENGTH characters. The SIZE attribute lists the length of the input field (if the user enters more characters, the text will scroll). The VALUE attribute specifies the initial value for the input field.

TYPE=password

> Same as TYPE=text, but the text will be hidden by "*" or similar characters. It is still sent in the clear to the server, though.

TYPE=checkbox

> Produces a checkbox. It has two states, on and off. When it is on when the form is submitted, it will be sent as "name=on", otherwise it is ignored altogether. If you use CHECKED, it will come up checked (selected) initially.

TYPE=radio

> Produces a radio button. A radio button always exists in a group. All members of this group should have the same NAME attribute, and different VALUEs. The VALUE of the selected radio button will be sent to the server. You must specify CHECKED on exactly one radio button, which then will come up selected initially.

TYPE=submit

Produces a button, which when pressed sends the contents of the form to the server. You can have more than one submit button in the form. Each should have a different NAME. The name and value of the pressed button will be sent to the server as well. The value of the VALUE attribute is typically used as text on the submit button.

TYPE=reset

Also produces a button, which will restore the form to its original state if pressed. The value of the VALUE attribute is typically used as text on the reset button.

TYPE=file

Allows the user to upload a file. It is still very new, so it is not very widely supported. It is typically presented as an input box with a button to start browsing the local hard disk. This way, a user can specify one or more filename(s) to upload.

TYPE=hidden

Allows you to embed information in the form which you do not want changed. This can be useful if the document is generated by a script and you need to store state information. NAME and VALUE of this input field will be sent to the server without modifications.

TYPE=image

Functions similar to a submit button, but uses an image instead. The ALIGN attribute controls the alignment of the image. The coordinates of the selected region will also be sent to the server, in the form of "NAME.x=n&NAME.y=n". A text browser will treat it as identical to a normal submit button.

- **<LI> </LI>**

TYPE=disc|square|circle when in UL, TYPE=1|a|A|i|I when in OL, VALUE=n

- **<TABLE> </TABLE>**

ALIGN=left|center|right, WIDTH=n|p%, BORDER=n, CELLSPACING=n, CELLPADDING=n

The ALIGN attribute controls the alignment of the table itself, but not of the individual cells. This can be set either in the TR element for an entire row, or in the TD and TH elements for individual cells. The WIDTH attribute can be a pixel width or a percentage. It indicates the suggested width of the table, although the browser can ignore this if it is not possible. A "100%" value means the table will span across the entire browser window.

```
<TABLE BORDER=1>
  <CAPTION>A test table with merged cells</CAPTION>
  <TR><TH ROWSPAN=2><TH COLSPAN=2>Average
  <TH ROWSPAN=2>other<BR>category<TH>Misc
  <TR><TH>height<TH>weight
  <TR><TH ALIGN=LEFT>males<TD>1.9<TD>0.003
  <TR><TH ALIGN=LEFT ROWSPAN=2>females<TD>1.7<TD>0.002
</TABLE>
```

<TD> </TD>

ROWSPAN=n, COLSPAN=n, NOWRAP, ALIGN=left|right|center,
VALIGN=top|middle|bottom, WIDTH=n, HEIGHT=n

The TD tag is used to mark up individual cells inside a table row. It may contain almost all tags, including nested tables. If the cell is a label of some sort, use TH instead of TD. The NOWRAP attribute indicates the contents of the current cell should not be wrapped. You must use BR in the cell to force line breaks to prevent the entire cell from showing up as just one line.

The ROWSPAN and COLSPAN attributes indicate how many rows or columns this cell overlaps. If you use these attributes, make sure you count correctly or you can get some *very* weird results. The ALIGN and VALIGN attributes control the horizontal and vertical alignment of the current cell. ALIGN can be set for left, right or centered cells. VALIGN indicates that the table cell's contents should appear at the top, the middle or the bottom of the row. Note that align and valign attributes for a cell override the values set for the row.

The WIDTH and HEIGHT attributes can be used to suggest a width and height for this cell. This should be a value in pixels. Setting different widths for multiple cells in the same column, or different heights for multiple cells in one row can cause unexpected effects.

<TR> </TR>
ALIGN=left|right|center, VALIGN=top|middle|bottom
HTML tables are constructed as a sequence of rows. Each row of table cells should be enclosed in a TR tag. The end tag is optional, since it is usually obvious to see where a row ends - where the new row begins, or where the entire table ends.

The ALIGN and VALIGN attributes control the horizontal and vertical alignment of the entire row. ALIGN can be set for left, right or centered cells. VALIGN indicates that the table cell's contents should appear at the top, the middle or the bottom of the row.

- **<TEXTAREA NAME=string, ROWS=n, COLS=n> </TEXTAREA>**

Attributes: NAME=string, ROWS=n, COLS=n
The TEXTAREA tag, used inside FORMs, sets up an area in which the user can type text. This text will be sent to the server when the form is submitted. The user can enter more than one line (as opposed to <INPUT TYPE=text> which only permits one line), although he will have to break lines himself.

The NAME attribute assigns the text area a name, used by the script, which processes the form. ROWS and COLS are used to specify the height and width of the text area, in number of characters.

To supply default text for the text area, put it inside the TEXTAREA tag.

Summary
To publish information for global distribution, one needs a universally understood language, a kind of publishing mother tongue that all computers may potentially understand. The publishing language used by the World Wide Web is HTML (from HyperText Markup Language).
HTML gives authors the means to:

- Publish online documents with headings, text, tables, lists, photos, etc.
- Retrieve online information via hypertext links, at the click of a button.
- Design forms for conducting transactions with remote services, for use in searching for information, making reservations, ordering products, etc.
- Include spread-sheets, video clips, sound clips, and other applications directly in their documents.
-

Model Questions

1. What is HTML and how is it used to construct web documents?
2. Explain the rules and regulations of HTML programming.
3. How is a HTML document structure explained?

References

1. HTML Pocket Reference, 2nd Edition by Jennifer Niederst
2. Html for the World Wide Web Visual Quickstart Guide: Visual QuickStart Guide by Elizabeth Castro -

Lesson 2.5.3

Dynamic HTML (DHTML)

CONTENTS

Objective
2.5.3.1 Introduction
2.5.3.2 Dynamic Positioning
2.5.3.3 Properties of STYLE
2.5.3.4 Putting the D in DHTML
2.5.3.5 Dynamic Programming
2.5.3.6 The Event Connection

Objective

In this we explore three main components of Dynamic HTML authoring:

1. Positioning; precisely placing blocks of content on the page and, if desired, moving these blocks around.
2. Style modifications; on-the-fly altering the aesthetics of content on the page.
3. Event handling; how to relate user events to changes in positioning or other style modifications.

2.5.3.1 Introduction

The Web is an ever evolving environment, and Web pages themselves are steadily blooming from static displays of data to interactive applications. "Dynamic HTML" is an umbrella term encompassing several ways in which Web developers can breathe life into pages which have traditionally been still portraits of information.

The basic notion behind Dynamic HTML is quite simple: allow any element of a page to be changeable at any time. Sounds like a dream, but as with any simple plan, "God is in the details," as they say. In the olden days, you could only change content on a page via CGI. This required a server to perform the changes to the page and re-serve the entire page, modifications and all, back to the client. While workable, this process was quite slow, as it placed a burden on both network traffic and server processing time. With long delays between a user's action and an on-screen response, building effective Web-based applications was quite constricting.

With DHTML, the magic occurs entirely on the client-side. This means that page modifications should appear immediately following a trigger, such as a user selection. And, remember, the DHTML dream is that you can modify any aspect of the currently loaded page -- text styles, swapped images, context-sensitive forms and tables, and even the on-screen data itself.

It's worth noting here, then, that "Dynamic HTML," isn't really about HTML, the markup language. By and large, DHTML describes the abstract concept of breaking up a page into manipulable elements, and exposing those elements to a scripting language which can perform the manipulations. The degree, or fineness, to which these elements are defined and actionable is a function of DHTML's maturity. Because we're only seeing the second generation of browsers supporting DHTML (MSIE 5 and the upcoming

Netscape 5 based on Mozilla), DHTML is still an evolving and at times, inconsistent set of tools.

Technically speaking, DHTML positioning is actually a type of style modification; however, this introduction will be all the clearer if we address positioning as its own topic.

The Old Standard about Standards

Dynamic HTML was and still is logical and wonderful idea whose time has come. Netscape thought so. Microsoft thought so. Yet, they didn't think alike. Netscape introduced a very modest vision of DHTML, debuted in Navigator 4.0. By and large, Netscape's DHTML was limited to the concept of "layers," also known as Cascading Style Sheet Positioning. Microsoft, on the other hand, chose to shoot for the moon, providing developers with surprisingly complex DHTML support in Internet Explorer 4.0, wherein many objects on the page were manipulable, including support for CSS Positioning.

In the past few years, quite a lot of Web development time has been spent haranguing over the challenge of coding pages which function under both versions of DHTML. Known as "cross browser DHTML," a cottage industry has sprung up to span the ravine between browsers. Visual editors such as Macromedia's Dreamweaver have matured to include pre-built DHTML objects which are already coded to work in both browsers. Still, the development community has its collected fingers crossed, however, that soon a common standard will emerge, allowing developers to focus on the true subject at hand, rather than compatibility issues. For the intranet developer, you may find it easier to focus on a single DHTML implementation, such as MSIE 5 or Netscape 4, in cases where your intranet requires the use of a particular browser platform.

Much of what we term "Dynamic HTML" -- the set of page elements which are manipulable -- is defined by a construct called the Document Object Model (or "DOM"). This model describes each page element and which of its characteristics may be modified, and how so. As you can imagine, Netscape and Microsoft do not share a common DOM. The hopes of DHTML standardization rest strongly on the success of a standardized DOM, currently in development by the World Wide Web Consortium (W3C) -- http://www.w3c.org/DOM/.

DHTML, to stress the point, is not a language itself. In practice, one programs Dynamic HTML using a combination of HTML, Cascading Style Sheets, and JavaScript. The Document Object Model, described earlier, provides a programming interface between HTML/CSS and JavaScript. In theory, other client-side scripting languages such as VBScript can be used with the Document Object Model, and therefore DHTML; but JavaScript remains the de facto standard scripting language for DHTML.

2.5.3.2 Dynamic Positioning

A Web developer may wear many hats -- writer, editor, programmer, and artist. As artists, we care how content is positioned on the page, whether for navigational or aesthetic purposes. Using DHTML, you can define a block of HTML content and precisely position it on the page.

There are two ways to position a block of content: Netscape's <LAYER> tag or Cascading Style Sheet Positioning (CSSP). Although Netscape's proprietary tag is simple to use, it is only supported within Netscape. Alternatively, CSSP is supported by both Netscape and Microsoft, so we will look at positioning using the style sheet syntax. There is no real loss of functionality, as layers and CSSP blocks behave nearly identically.

DHTML Programming

Typically, a block of content is contained within <DIV> tags. Imagine, then, that you would like to create a small block of content -- an image plus a mailto hyperlink:

```
<DIV>
<A HREF="mailto:me@myhouse.com">Mail me!</A><BR>
<IMG SRC="mailbox.gif" width=30 height=15 alt='Mailbox'>
</DIV>
```

To position this block, however, you must tell the <DIV> block to use a style sheet. A style sheet, as you may know, defines display characteristics. Style sheets can either be pre-defined, and applied to tags as style sheet classes, or defined on-the-fly. Typically, when using CSS Positioning, you define the block's style on-the-fly. We do this by using the STYLE attribute for the <DIV> tag. The STYLE attribute contains a string which lists each desired style property and its value. For example:

```
<DIV ID="mailblock" STYLE="position:absolute; width:auto; height:auto; left:400px;
top:50px; background-color:white">
<A HREF="mailto:me@myhouse.com">Mail me!</A><BR>
<IMG SRC="mailbox.gif" width=30 height=15 alt="Mailbox">
</DIV>
```

In the above example, we've first added the ID attribute to assign an identifier to this content block. This may become useful later when manipulating this block using JavaScript. Next, the STYLE attribute contains a list of property parameters and values. Notice that a colon separates the parameter (left) from the value (right) and a semicolon separates each parameter/value pair from the next. In this case, we've specified that *mailblock* should be absolutely positioned on the page (meaning that it is not fixed relative to the position of any other page elements), with automatic width and height, 400 pixels right from the left edge of the browser's content window and 50 pixels down from the top edge of the browser's content window.

Simple enough -- except for one bit of trouble: we don't know the size of the user's screen or browser window! Thus, we don't really know where 400 pixels to the right will land this block relative to the rest of the page. This could result in an ugly layout. The solution to precise, but adaptable, layouts is to use relative positioning. Imagine, for example, that we want *mailblock* to appear slightly indented leftwards from the upper-right corner of the page. To achieve this position, we will create a two-level <DIV> block -- a parent and a child:

```
<DIV ID="mailblock_parent" ALIGN="right">
<DIV ID="mailblock_child" STYLE="position:relative; width:auto; height:auto; left:10px;
top:0px; background-color:white">
<A HREF="mailto:me@myhouse.com">Mail me!</A><BR>
<IMG SRC="mailbox.gif" width=30 height=15 alt="Mailbox">
```

</DIV>
</DIV>
In this example, the parent <DIV> block contains, as a child <DIV> block, the content for the mailto link and image. The parent block is aligned to the right edge of the page. The child block, which contains the actual content, is dynamically positioned relative to the parent, 10 pixels to the left of the parent's left edge.

2.5.3.3 Properties of STYLE

Clearly, positioning your block of content depends upon the specified STYLE properties. We seemed to pull our example properties out of a hat -- position, width, left? Where did we get these from? Well, several documents on Web contain references to the positioning properties available for style sheets -- two worth reviewing are Microsoft's CSS Attributes Reference (under "Positioning Properties") and Netscape's Defining Positioned Blocks of HTML Content.

Below we've included a handy chart summarizing the common STYLE properties which you may want to use when using the CSS syntax to position blocks of content.

position	Specifies how the block should be positioned on the page with respect to other page elements. Possible values: *"position:absolute**;"*** Block is positioned absolutely within the browser window, relative to <BODY> block. *"position:relative*;" Block is positioned relative to its parent block, if any, or else normal flow of page. *"position:static*;" Block is positioned according to standard HTML layout rules -- follows flow of document. (**MSIE 4+ only; Netscape 5)**
width	Specifies the width at which the block's contents should wrap. You may specify width in measured units (*50px*), as a percentage of the parent block's width (*50%*), or *auto* which wraps the block according to its parent's width. Examples: *"width:50px;"* or *"width:50%;"*
height	Specifies the height of the block, measured in units (*50px*), percentage of the parent block's height (*50%*), or *auto*. Note that the height of the block will be forced to the minimum necessary to display its contents; however, you can make the block taller than necessary. Examples: *"height:50px;"* or *"height:50%;"*
left	Specifies the offset of the left edge of the block in accordance with the *position* attribute. Positive measures (*5px*) are offset towards the right while negative measures (*-5px*) are offset towards the left. Examples: *"left:5px;"* or *"left:-5px;"*
top	Specified the offset from the top edge of the block in accordance with the *position* attribute. Positive measures (*5px*) are offset towards the bottom of the page while negative measures (*-5px*) are offset towards the top of the page. Examples: *"top:10px;"* or *"top:-10px;"*

clip	Specifies a rectangular portion of the block which is visible. Typically, you use the *clip* property if you want to show only a portion of the block, therefore hiding a portion. Syntax: **MSIE:** clip:rect(*top right bottom left*) Example: *"clip:rect(0px 30px 50px 0px);"* **Netscape**: clip:rect(*left,top,right,bottom)* Example: *"clip:rect(0,0,30,50);"* Notice that the syntaxes vary between browsers, both in the need to specify measurement units (MSIE) and the order of the parameters.
visibility	Specifies whether a block should be visibile. If not visible, the block will not appear on the page, although you can make it visible later using JavaScript. The possible values for this property again vary between browsers. **MSIE:** *"visbility:inherit;"* Block inherits the visibility property of its parent. *"visibility:visible;"* Block is visible. *"visibility:hidden;"* Block is invisible. **Netscape:** *"visbility:inherit;"* Block inherits the visibility property of its parent. *"visibility:show;"* Block is visible. *"visibility:hide;"* Block is invisible.
z-index	Specifies the "stacking order" of blocks, should they happen to overlap other positioned blocks. A block is assigned a z-index, which is any integer. When blocks overlap, that which has the greater positive z-index appears above a block with a lower z-index. Blocks with an equal z-index value are stacked according to the order in which they appear in the source code (bottom-to-top: first block defined appears on bottom, last block defined appears on top). *Example: "z-index:3;"*
background-color	Specifies the background color for the block. Examples: *"background-color:green;"* or *"background-color:FF8F00;"*
background-image	Specifies a background image for the block. Example: *"background-image:url('images/ tilewood.jpg');"*

Employing the various STYLE properties gives you powerful control over the position and look of the blocks of content on your page. Conceptually, then, when we think in DHTML we think of a page as made up of one or more blocks. Of course, this is not immediately evident to the viewer, who essentially sees a flat Web page, without realizing that several smaller blocks of content are positioned here and there to create the overall effect.

Still, the question begs: how is this *dynamic*? Precise, yes, but dynamic?
Consider the fact that, now that the blocks have been put into place, you can -- at any time -- change their properties. Position, background, clipping region, z-index -- it's all plastic, with the help of JavaScript, and that is the reason to be excited!

2.5.3.4 Putting the D in DHTML

Once a content block is positioned on the page using <DIV> tags in your HTML, you can use JavaScript to modify its properties. This has many possible consequences: you can move the entire block up, down, left or right. You can change its background color, or change the clipping region, causing more or less of the block to be visible. Speaking of visibility, you can even hide or show the entire block in an instant via the visibility property.

How, then, does JavaScript access the style properties of the content block? The answer is twofold:

1. Assuming that you are familiar with JavaScript, you know that data structures are represented as *objects*, and each object has a set of *properties*. JavaScript statements can read or write to the properties of an object.
2. Content blocks contained in <DIV> tags are exposed as objects to JavaScript by the Document Object Model. This means that, following the object and property specifications defined by the DOM, JavaScript can access the style sheet properties of the content block. Voila!

Let's recall our example content block, *mailblock*. In its simple form:

```
<DIV ID="mailblock" STYLE="position:absolute; width:auto; height:auto; left:400px;
top:50px; background-color:white">
<A HREF="mailto:me@myhouse.com">Mail me!</A><BR>
<IMG SRC="mailbox.gif" width=30 height=15 alt="Mailbox">
</DIV>
```

Because Netscape and Microsoft do not currently share compatible DOM's, the above block is exposed to different objects in each browser. For now, we'll consider each case separately -- a future article will look at the issue of patching over this problem in "cross browser DHTML."

In Netscape's DOM, each <DIV> block takes the form of a Layer Object. In Microsoft, each block is exposed as a DIV object, which in turn possesses a STYLE object, whose properties reflect the familiar style attribute properties.This might sound confusing, but it's all a matter of syntax. Let's first consider how you would construct a JavaScript reference to *mailblock*. You use the identifier specified in the ID attribute:

Microsoft:

```
document.all["mailblock"]
```

Netscape:

```
document.layers["mailblock"]
```

Now, let's consider the background color property for *mailblock*. The CSS property for the background color was "background-color" as specified in our <DIV> tag. When reflected via the DOM, however, this property takes on a different name between browsers:

Microsoft:
document.all["mailblock"].style.backgroundColor
Netscape:
document.layers["mailblock"].bgColor

Knowing that, we can easily create a JavaScript statement which would change the background color of *mailblock* to green.
Microsoft:
document.all["mailblock"].style.backgroundColor="green";
Netscape:
document.layers["mailblock"].bgColor="green";

Similarly, we can modify the *top* and *left* style properties which will move the block on-screen. Thus, once we know the correct DOM property for the style sheet *left* property, we can move *mailblock* directly to the left edge of the page:
Microsoft:
document.all["mailblock"].style.left="0px";
OR
document.all["mailblock"].style.pixelLeft=0;
Netscape:
document.layers["mailblock"].left=0;

Once again, it is likely to ask just how one is supposed to discover which DOM properties for which browser map to the style sheet properties we've seen. Two handy on-line references are once again provided -- Netscape's Using JavaScript with Positioned Content lists the Layer Object's properties and Microsoft's currentStyle Object Reference lists the properties relevant to their style object. In the effort to make life simpler, however, the following table summarizes which style sheet properties map onto which DOM properties for each browser.

CSS Property	Netscape Layer Object Property	Microsoft currentStyle Object Property
position	none	*position* Note: read-only, cannot modify via script
width	none	*pixelWidth*
height	none	*pixelHeight*
left	*left* Note: Accepts either an integer (assumed pixel units) or a percentage string. Examples: *left=10* or *left="20%"*	*left* or *pixelLeft* Note: *pixelLeft* accepts a string which specifies measurement units; e.g. *pixelLeft="10px"* whereas *left* accepts an integer and assumes pixel units; e.g. *left=10.*

top	*top* Note: Accepts either an integer (assumed pixel units) or a percentage string. Examples: *top=10* or *top="20%"*	*top* or *pixelTop* Note: *pixelTop* accepts a string which specifies measurement units; e.g. *pixelTop="10px"* whereas *top* accepts an integer and assumes pixel units; e.g. *top=10*.
clip	*clip.top* *clip.left* *clip.right* *clip.bottom* *clip.width* *clip.height* Each dimension of the clipping coordinates is its own property, as seen here. Changing any property immediately causes the on-screen clip to change.	*"rect(top right bottom left)"* To change the clip in MSIE you must redefine all clipping coordinates. The syntax can be a bit confusing. Example: document.all.blockname.style.clip="rect(0 25 25 0)"< /FONT>
visibility	*visibility* May contain any of "inherit", "show", or "hide"	*visibility* May contain any of "inherit", "visible", or "hidden"
z-index	*zIndex* Any non-negative integer.	*zIndex* MSIE allows negative integers, but you may as well stick with positive for the sake of Netscape.
background-color	*bgColor* Accepts string containing pre-defined color name or hex RGB triplet. Examples: document.layers["blockname"].bgColor="black" document.layers["blockname"].bgColor="#000000"< /FONT>	*backgroundColor* Accepts string containing pre-defined color name or hex RGB triplet. Examples: document.all.blockname.style.backgroundColor="black" document.all.blockname.style.backgroundColor="#000000"< /FONT>
background-image	*background.src* Example: document.layers["blockna	*backgroundImage* Example: document.all.blockname

	me"].background.src="images/tile.jpg"< /FONT>	

2.5.3.5 Dynamic Programming

Armed with the understanding of positioned content blocks, and the ability to manipulate their characteristics via JavaScript, you basically need only to rely on your imagination (although JavaScript skill can certainly help). Consider some nifty uses of dynamic positioning:

- Animation: images contained in a positioned block can be moved around the page following a certain path.
- Drop-down menus: modifying clipping regions allows you to show or hide portions of a content block. To create a drop down menu, you would initially show only the top strip of the block containing the menu choices. When the user triggers the menu to drop, you chance the clipping region to display the vertical length of the menu chosen. We'll look at triggers from user events later in this article.
- Content-swapping: by positioning multiple blocks of content at the same spot on the page, yet keeping all but one block invisible, you can quickly swap a new block of content into place by changing their visibility properties. Alternatively, by keeping two overlapped blocks visible, you can modify their clipping regions in such a pattern as to produce a "transition" effect.

Dynamic Styles

As we've seen, positioning a content block is a form of style sheet. However, there are other uses for style sheets aside from placing blocks on the page. Style sheets (aka *Cascading Style Sheets* or *CSS*) can be used to define any set of styles which may apply to certain HTML elements or block of HTML. Like JavaScript, Cascading Style Sheets are their own in-depth web authoring topic. Because our focus is on DHTML, insofar as it uses and combines web authoring technologies, this isn't the place for a detailed tutorial on creating styles or style sheets.

In brief, there are several ways to apply styles using style sheets. You can define styles which apply globally to all HTML elements of a certain type. For instance, you might create a style sheet which specifies that all <H3> elements be bold (*font-weight*) and red (*color*).

It may also be defined style "classes," which are predefined styles applied to an element on-demand. For example, you might specify that the style class "redbold" contains the styles bold (*font-weight*) and red (*color*). This *redbold* class would only apply to elements in which you demand it; e.g. <BLOCKQUOTE CLASS=redbold>.

Lastly, it is already seen, you can define and apply styles on-the-fly, using the STYLE attribute for an element tag; e.g. <H4 STYLE="color:red; font-weight:bold;">. These on-the-fly styles apply only to the tag in which they are specified.

Both Microsoft and Netscape support CSS as described above, although Microsoft supports additional style properties which do not exist under Netscape. The critical difference between the browsers, however, is the degree to which their CSS support is *dynamic*.

Quasi-Dynamic HTML

The style sheet determines how, aesthetically, the document is rendered on the screen. Once rendered, though, can these styles be modified on-screen? This is where the two browsers diverge markedly.

Within Microsoft's Internet Explorer, you can, via JavaScript, modify many of the properties of a style sheet or an element's individual style. These changes will be immediately reflected on-screen -- so, if you changed the *redbold* class's *color* property to blue, any on-screen text using the *redbold* class will suddenly change to blue. Similarly, you can change any style property for those elements which possess inline ("on-the-fly") styles.

Netscape, on the other hand, is not so accomodating. While there is an interface to style sheets via JavaScript (albeit very different from the Microsoft implementation), you can only use JavaScript to define styles *before* they are rendered. By and large, this would be used as a slightly more flexible replacement for definining your styles rather than using standard CSS syntax. On the other hand, using standard CSS syntax would maintain compatibility between browsers. For this reason, it could be argued that Netscape's support for CSS via JavaScript is not terribly useful, and not at all dynamic, since changes applied to styles once the page has been rendered do not appear on-screen.

Stuck in the middle are we -- but because Microsoft's support for style modifications is far more robust, we will focus here on dynamic styles in MSIE. These concepts will apply to Netscape 5, based on Mozilla, whenever it finally arives, although the exact syntax may differ.

Inline Styles

Speaking non-strictly, we can say that an "inline style" is a style which applies only to an individual element rather than all elements of a certain type or class. Typically, an inline style is defined using the STYLE attribute for the element's tag; e.g. <H2 STYLE="color:blue">.

Within MSIE, then, you can use JavaScript to modify an inline style at any time. The modification will take effect on-screen immediately. You do this using the Style Object, which we looked at earlier with Dynamic Positioning. The Style Object in MSIE's DOM supports every property which CSS supports for styles. However, the property names differ slightly between CSS syntax and DOM syntax. Fortunately, the naming differences follow a rule. Consider the CSS property *font-weight*. The corresponding Style Object property is named *fontWeight*. Similarly, the CSS property *text-align* maps to the Style Object property *textAlign*. Of course, *color* retains the same name in both syntaxes. As you can see, then, CSS property names simply lose their hyphen and capitalize the first letter following the hyphen when used in JavaScript syntax.

So, then, confusing though that might sound, let's look at an example which illustrates just how easy changing styles can be. Imagine that you have the following HTML element:

```
<P ID="selectrule" ALIGN="left" STYLE="color:blue; font-weight:normal;">
Please select one item below.
</P>
```

Now, some series of events occur and the user fails to follow instructions. So, you'd like to emphasize the exhortation:

```
document.all["selectrule"].style.color="red";
document.all["selectrule"].style.fontWeight="bold";
```

When the above statements are executed, the text associated with the element named *selectrule* will become red and bold. Of course, you can set the element back to its original state by setting *color* back to "blue" and *fontWeight* back to "normal".
Notice that our JavaScript statements are modifying the styles directly associated with the particular element. You can always do this, even if the element's style wasn't defined inline, but was inherited from a class or global style sheet. No matter how the element gained its original style, whether inline or by style sheet, directly modifying the style in the above manner only applies to the individual element. For this reason, this form of JavaScript dynamic styles is always an inline style.

Changing the Rules

Modifying the style of an individual element is very precise, but sometimes you may want to have a more global effect: perhaps you'd like to modify a style defined by a style sheet as a class or global style. Such a modification would have a ripple-effect, immediately re-rendering all elements on the page which are subject to that style sheet.
Below is a style sheet definition which specifies two styles: a global style which applies to all <H3> element (blue, arial font family, large size), and a generic style class named *warning* (red, bold, italic)which will apply to any element which uses that class.

```
<STYLE ID="sheet1">
H3 { color:blue; font-family:"Arial"; font-size:large; }
.warning { color:red; font-weight:bold; font-style:italic; }
</STYLE>
```

Once the document has been rendered according to the above style sheet, to modify this style sheet via JavaScript such that any changes are dynamically reflected on-screen is the goal. To do this, you'll need to learn some new terminology. Each "style definition" in the style sheet is known as a *"rule*". Thus, the line defining the style for H3 is one rule, while the line which defines the *warning* class is a second rule.
In Microsoft's Document Object Model, you access a style sheet through the styleSheets[] array (aka "collection"). A single style sheet is defined as everything between a set of <STYLE>...</STYLE> tags. The above example, then, is one style sheet. If it were the first instance of such tags in your source code then it would be the first style sheet, as well. You access the first style sheet via JavaScript using the reference:

```
document.styleSheets[0]
```

Within each styleSheet object is a rules collection. This array contains an entry for each rule within the style sheet. Therefore, you would access the first rule -- the H3 rule -- of this style sheet as:

```
document.styleSheets[0].rules[0]
```

And the second rule -- the *warning* class -- as:

```
document.styleSheets[0].rules[1]
```

Now, before this grows ever more baffling, let me add that each rule object possesses a style object. And it is through that style object which you can modify the rule. What?? Example: You want to change the H3 rule in this example so that its color becomes yellow:

```
document.styleSheets[0].rules[0].style.color="yellow";
```

Now you want to change this rule's font family to courier:

```
document.styleSheets[0].rules[0].style.fontFamily="courier";
```

You can modify any style property for the rule, even one which wasn't specified in the original style sheet. Thus, if you'd suddenly like to make all H3 text on the page italic:

```
document.styleSheets[0].rules[0].style.fontStyle="italic";
```

Any of the above JavaScript statements will apply the changes to all <H3> elements on the page. Similarly, making any changes to the style object for the *warning* class rule (document.styleSheets[0].rules[1].style.*property*) will apply all changes to any tags which specify the attribute STYLE="warning"; e.g. <P STYLE="warning">.
Clearly, dynamic styles are extremely powerful, allowing you to alter the look of a page in seconds. A particularly interesting style property you should investigate is *display*. This property allows you to set whether a style is rendered at all or not. An un-rendered style will not take up any screen space, but if you later change the *display* property from *"none"* to *"block"*, any element with that style will suddenly appear. The rest of the page is automatically re-rendered to accomodate the new content. This differs from an invisible block precisely because the entire page expands and contracts to accomodate the presence or absence of the content contained with the style.

2.5.3.6 The Event Connection

Quite frequently, you want some type of trigger to cause your DHTML to kick in. Whether repositioning blocks or changing style properties, some *event* usually causes these changes.
Many different types of event can occur on a Web page, most of them caused by the user. He or she might click on a button (*click* event), might move the mouse over an element (*mouseover* event), or move the mouse off of an element (*mouseout* event). The user may submit a form (*submit* event) or resize the browser window (*resize* event). Additionally, some events occur without direct user intervention -- the page may finish loading (*load* event), for example.
Events, although an intrinsic part of DHTML, are not part of the standard DOM Level 1 specification. Consequently, event handling between Microsoft and Netscape browsers can vary enormously in practice. The DOM Level 2 specification does include events, but unfortunately as of this writing, DOM Level 2 support in both major browsers is still more of a dream than a reality.
Managing events can be relatively simple or quite complex, depending on the ambitions of your project. Since this is an introduction, we'll focus mainly on event basics. Fortunately, basic events are handled most similarly between browsers.

Event Handlers

Events occur on a Web page whether you choose to act on them or not. When the user moves the mouse over an element, a *mouseover* event occurs. If you would like to leverage this event as a trigger for some dynamic action, you must construct an *event handler*.
An event handler is created as an attribute for the tag which defines the element at which you wish to catch the event. Event handler attribute named follow the syntax *onEventname*, and they accept JavaScript statements as their action. For example, the following tag creates a hyperlink with a *mouseover* event handler specified.

```
<A HREF="page.html" onMouseOver="changeStatus('Read this page');">Click here</A>
```

The *onMouseOver* event handler catches a *mouseover* event. When this event occurs at this element, a JavaScript function is called named *changeStatus()*. This is a fictional function, you can imagine that it might exist to change the message on the browser's status bar. Any JavaScript statement is allowed in an event handler, so we could also execute direct statement rather than call a function. For example, suppose that a *mouseover* for this element in MSIE should modify a style sheet's color property:

<A HREF=
"page.html"onMouseOver="document.styleSheets[0].rules[0].color='blue'">Click
here</A>
The above example assumes MSIE, since live style sheet modifications aren't supported in Netscape. We also assume that a style sheet exists in this document! But, hey, it's just an example.
Once again, a convenient table will help summarize the common event and their event handler names.

Event	Event Handler Syntax	Description
click	onClick	User clicks (left) mouse button on an element.
submit	onSubmit	User submits a form, this event fires before the form submission is processed.
reset	onReset	User resets a form.
focus	onFocus	User brings focus to an element either via mouse click or tabbing.
blur	onBlur	User loses focuses from an element by clicking away or tabbing away.
mouseover	onMouseOver	User moves mouse over an element.
mouseout	onMouseOut	User moves mouse off of an element.
mousemove	onMouseMove	User moves mouse.
change	onChange	User changes value in a text, textarea, or select field.
select	onSelect	User selects (highlights) a portion of text in a text or textarea field.
resize	onResize	User resizes browser window or frame.
move	onMove	User moves browser window or frame.
load	onLoad	Page completes loading.
unload	onUnload	User exits page (by navigating to a new page or quitting browser).
error	onError	An error occurs loading an image or document.
abort	onAbort	User stops an image loading using

		the stop button.

The above table is a quick reference guide. There are some important caveats to keep in mind. For one, this is not a comprehensive list of all events, although these are by far the most commonly used. Both browser support additional events for detecting keypresses and other mouse actions, while MSIE supports additional events above and beyond Netscape's. Secondly, you must keep in mind that not every event is applicable to every element. This also varies between browsers. For example, within Netscape a *mouseover* event only applies to a hyperlink <A>, area <AREA> or layer <LAYER>. Yet, within MSIE, you can apply a *mouseover* event to almost any element, including images <IMG>, and paragraphs <P>.
In general, the above rule holds between browsers: Netscape restricts each event to certain limited elements, while MSIE allows most events to be handled at most elements. The best way to clarify these distinctions is to read the documentation -- Netscape's Events and Event Handlers and Microsoft's DHTML Events Reference.

Mastering event handling is most certainly a topic unto itself. At the surface, handling basic events is simple, as you've seen. The classic "rollover effect" is a perfect example of a simple event used in DHTML. A rollover occurs when the user moves the mouse over an element; while the mouse hovers over the element, its appearance changes to reflect that it is "active". When the mouse moves off the element is reverts to a more subdued state. Rollovers commonly use changes in either image or style. Below is a basic MSIE style-based rollover, which makes the "active" text red and bold, and returns to normal blue when inactive. Remember that such dynamic styles don't work so easily under Netscape, although you could certainly re-imagine this example for Netscape, perhaps by swapping an image for the rollover effect; or swapping a layer.

MSIE Dynamic Style Rollover Example

```
<A HREF="special.html" TARGET="mainframe" STYLE="color:blue;font-
weight:normal;font-
family:Arial"<BR>onMouseOver="this.style.color='red';this.style.fontWeight='bold'"<BR>
onMouseOut="this.style.color='blue';this.style.fontWeight='normal'">Today's
special</A>
```

Summary

As with any introduction, we hope that much of what's been written serves as inspiration for your own ideas. Coding impressive DHTML is not necessarily obscure or difficult, but it does benefit from imagination and cleverness. No doubt you'll find plenty of pre-built DHTML effects in your Web travels, and these, too, are useful sources of inspiration and programming techniques. Always approach an interesting effect with a "How'd they do that?" mindset, rather than simply adopting what they did.
Because DHTML is a moving target, as a developer you necessarily need to be sensitive to the browser environment that you are designing for. This is a major reason why cross-browser coding is such a challenge. The promise of a standardized DOM adopted by both browsers seems an ever distant possibility, even with fifth generation browsers. Intranet developers who have the relative "luxury" of a single browser platform can enjoy some additional freedom from the strain of balancing multiple browser environments.

Model Questions

1. What is DHTML and what is its role in web pages?
2. Explain dynamic positioning.
3. Explain the properties of Style.

References

1. JavaScript & DHTML Cookbook by Danny Goodman
2. Dhtml and Css for the World Wide Web: Visual QuickStart Guide by Jason Cranford Teague

Author :-
Asha Smitha .B.
Center for Biotechnology
Acharya Nagarjuna University

Lesson 2.5.4

WEB PAGES

Contents

Objective
2.5.4.1 Introduction
2.5.4.2 Planning
2.5.4.3 Technology
2.5.4.4 Web server support
2.5.4.5 Organizing information
2.5.4.6 Hierarchy of importance
2.5.4.7 Relations
2.5.4.8 Function
2.5.4.9 Section contents
Summary
Model questions
References

Objective

- What is the mission of your organization?
- How will creating a Web site support your mission?
- What are your two or three most important goals for the site?
- Who is the primary audience for the Web site?
- What do you want the audience to think or do after having visited your site?
- What Web-related strategies will you use to achieve those goals?
- How will you measure the success of your site?
- How will you adequately maintain the finished site?

2.5.4.1Introduction

Though still young, the World Wide Web has already undergone several transformations. The framers of the Web were scientists who wanted to create a device-independent method for exchanging documents. They devised HTML (HyperText Markup Language) as a method for "marking up" the structure of documents to allow for exchange and comparison. The focus was on the structural logic of documents, not the visual logic of graphic design.

But the Web quickly caught on as a publishing medium; no communication device is more inexpensive or far-reaching. As a tool for communication, however, Web authoring with HTML has limitations. With their focus on the structure of documents, the originators of the Web ignored those visual aspects of information delivery that are critical to effective communication. Once the Web was established as a viable publishing medium, these limitations became obvious and cumbersome. Pages that conformed to HTML standards lacked visual appeal, showing little evidence of the past five centuries of progress in print design. Graphic designers took this relatively primitive authoring and layout tool and began to bend and adapt it to a purpose it was never intended to serve: graphic page design.

The Web viewing audience was also beginning to refine its tastes. The pioneering Web "surfers" who were content to skim the surface of Internet documents are now outnumbered. People are turning to the Web for information — information with depth, breadth, and integrity.

Web design should be almost transparent to the reader. How to create a user interface that will allow visitors to your site to navigate your content with ease. This is a new genre with its own style and guidelines.

As is the case with many innovations, the Web has gone through a period of extremes. At its inception, the Web was all about information. Visual design was accidental at best. Web pages were clumsily assembled, and "sites" were accumulations of hyperlinked documents lacking structure or coherence. Designers then took over and crafted attractive, idiosyncratic, and often baffling containers for information. The Web became a better-looking place, but many users hit up against barriers of large graphics, complex layouts, and nonstandard coding. Every site was different, and each required users to relearn how to use the Web, because "real" designers could not be bound by standards or conventions. Instead, designers pushed the boundaries of HTML, using workarounds, kludges, and sleight of hand to design on the cutting edge.

Today, the field of Web design is seen much more as a craft than an art, where function takes precedence over form and content is king. Innovative designs using fancy navigational doodads are generally seen as an annoyance standing between the user and what he or she seeks. Large graphic eye-candy, no matter how pleasing, is simply wasted bandwidth. Like 1960s architecture, much of yesterday's Web design now makes users wince and wonder how it could ever have been fashionable. Instead, today's Web designers are also information architects and usability engineers, and a user-centered design approach is the key to a successful Web site. Instead of constantly requiring users to relearn the Web, sites are beginning to look more alike and to employ the same metaphors and conventions. The Web has now become an everyday thing whose design should not make users think.

When things change this fast, humans have a hard time adapting, keeping up, and just plain understanding what's going on. But people's reactions to the Web changed so fast precisely because so few of us really understood what it is. In fact, most of us didn't have the time to think hard about how Web sites could truly be useful and good things and how important sound design principles are to making them so.

Frenzied anxiety forced us to rush to legions of "experts" who played upon our fears that "we didn't get it." Through hype and jargon (not to mention wildly creative business modeling), they bullied us back to where we are today: square one.

Fortunately, this wonderful little book is still here to show us those fundamentals.

2.5.4.2 Planning

Web sites are developed by groups of people to meet the needs of other groups of people. Unfortunately, Web projects are often approached as a "technology problem," and projects are colored from the beginning by enthusiasms for particular Web techniques or browser plug-ins (Flash, digital media, XML, databases, etc.), not by real human or business needs. People are the key to successful Web projects. To create a substantial site you'll need content experts, writers, information architects, graphic designers, technical experts, and a producer or committee chair responsible for seeing the project to completion. If your site is successful it will have to be genuinely useful to your target audience, meeting their needs and expectations without being too hard to use.

Although the people who will actually use your site will determine whether the project is a success, ironically, those very users are the people least likely to be present

and involved when your site is designed and built. Remember that the site development team should always function as an active, committed advocate for the users and their needs. Experienced committee warriors may be skeptical here: these are fine sentiments, but can you really do this in the face of management pressures, budget limitations, and divergent stakeholder interests? Yes, you can — because you have no choice if you really want your Web project to succeed. If you listen only to management directives, keep the process sealed tightly within your development team, and dictate to imagined users what the team imagines is best for them, be prepared for failure. Involve real users, listen and respond to what they say, test your designs with them, and keep the site easy to use, and the project will be a success.

What are your goals?

A short statement identifying two or three goals should be the foundation of your Web site design. The statement should include specific strategies around which the Web site will be designed, how long the site design, construction, and evaluation periods will be, and specific quantitative and qualitative measures of how the success of the site will be evaluated. Building a Web site is an ongoing process, not a one-time project with static content. Long-term editorial management and technical maintenance must be covered in your budget and production plans for the site. Without this perspective your electronic publication will suffer the same fate as many corporate communications initiatives — an enthusiastic start without lasting accomplishments.

Know your audience

The next step is to identify the potential readers of your Web site so that you can structure the site design to meet their needs and expectations. The knowledge, background, interests, and needs of users will vary from tentative novices who need a carefully structured introduction to expert "power users" who may chafe at anything that seems to patronize them or delay their access to information. A well-designed system should be able to accommodate a range of users' skills and interests. For example, if the goal of your Web site is to deliver internal corporate information, human resources documents, or other information formerly published in paper manuals, your audience will range from those who will visit the site many times every day to those who refer only occasionally to the site.

Design critiques

Each member of a site development team will bring different goals, preferences, and skills to the project. Once the team has reached agreement on the mission and goals of the project, consensus on the overall design approach for the Web site needs to be established. The goal at this stage is to identify potential successful models in other Web sites and to begin to *see the design problem from the site user's point of view.*

Unfortunately, production teams rarely include members of the target audience for the Web site. And it is often difficult for team members who are not already experienced site designers to articulate their specific preferences, except in reference to existing sites. Group critiques are a great way to explore what makes a Web site successful, because everyone on the team sees each site from a user's point of view. Have each team member bring a list of a few favorite sites to the critique, and ask them to introduce their sites and comment on the successful elements of each design. In this

way you will learn one another's design sensibilities and begin to build consensus on the experience that your audience will have when they visit the finished site.

Content inventory

Once you have an idea of your Web site's mission and general structure, you can begin to assess the content you will need to realize your plans. Building an inventory or database of existing and needed content will force you to take a hard look at your existing content resources and to make a detailed outline of your needs. Once you know where you are short on content you can concentrate on those deficits and avoid wasting time on areas with existing resources that are ready to use. A clear grasp of your needs will also help you develop a realistic schedule and budget for the project. Content development is the hardest, most time-consuming part of any Web site development project. Starting early with a firm plan in hand will help ensure that you won't be caught later with a well-structured but empty Web site.

Developing a site specification

The site specification is the planning team's concise statement of core goals, values, and intent, to provide the ultimate policy direction for everything that comes next. Designing a substantial Web site is a costly and time-consuming process. When you're up to your neck in the daily challenges of building the site, it can be surprisingly easy to forget why you are doing what you are, to lose sight of your original priorities, and to not know on any given day whether the detailed decisions you are making actually support those overall goals and objectives. A well-written site specification is a powerful daily tool for judging the effectiveness of a development effort. It provides the team with a compass to keep the development process focused on the ultimate purposes of the site. As such, it quickly becomes a daily reference point to settle disputes, to judge the potential utility of new ideas as they arise, to measure progress, and to keep the development team focused on the ultimate goals.

At minimum, a good site specification should define the content scope, budget, schedule, and technical aspects of the Web site. The best site specifications are very short and to the point, and are often just outlines or bullet lists of the major design or technical features planned. The finished site specification should contain the goals statement from the planning phase, as well as the structural details of the site.

Production issues

- How many pages will the site contain? What is the maximum acceptable count under this budget?
- What special technical or functional requirements are needed?
- What is the budget for the site?
- What is the production schedule for the site, including intermediate milestones and dates?
- Who are the people or vendors on the development team and what are their responsibilities?

These are big questions, and the broad conceptual issues are too often dismissed as committees push toward starting the "real work" of designing and building a Web site. However, if you cannot confidently answer all of these questions, then no amount of design or production effort can guarantee a useful result.

Avoiding "scope creep"

The site specification defines the scope of your project — that is, what and how much you need to do, the budget, and the development schedule. "Scope creep" is the most prevalent cause of Web project failures. In badly planned projects, scope creep is the gradual but inexorable process by which previously unplanned "features" are added, content and features are padded to mollify each stakeholder group, major changes in content or site structure during site construction are made, and more content or interactive functionality than you originally agreed to create is stuffed in. No single over commitment is fatal, but the slow, steady accumulation of additions and changes is often enough to blow budgets, ruin schedules, and bury what might have been an elegant original plan under megabytes of muddle and confusion.

Don't leap into building a Web site before you understand what you want to accomplish and before you have developed a solid and realistic site specification for creating your Web site. The more carefully you plan, the better off you will be when you begin to build your site.

One excellent way to keep a tight rein on the overall scope of the site content is to specify a maximum page count in the site specification. Although a page count is hardly infallible as a guide (after all, Web pages can be arbitrarily long), it serves as a constant reminder to everyone involved of the project's intended scope. If the page count goes up, make it a rule to revisit the budget implications automatically — the cold realities of budgets and schedules will often cool the enthusiasm to stuff in "just one more page." A good way to keep a lid on scope creep is to treat the page count as a "zero sum game." If someone wants to add pages, it's up to them to nominate other pages to remove or to obtain a corresponding increase in the budget and schedule to account for the increased work involved.

Changes and refinements can be a good thing, as long as everyone is realistic about the impact of potential changes on the budget and schedule of a project. Any substantial change to the planned content, design, or technical aspects of a site must be tightly coupled with a revision of the budget and schedule of the project. People are often reluctant to discuss budgets or deadlines frankly and will often agree to substantial changes or additions to a development plan rather than face an awkward conversation with a client or fellow team member. But this acquiescence merely postpones the inevitable damage of not dealing with scope changes rationally.
The firm integration of schedule, budget, and scope is the only way to keep a Web project from becoming unhinged from the real constraints of time, money, and the ultimate quality of the result. A little bravery and honesty up front can save you much grief later. Make the plan carefully, and then stick to it.

Site definition and planning

This initial stage is where you define your goals and objectives for the Web site and begin to collect and analyze the information you'll need to justify the budget and resources required. This is also the time to define the scope of the site content, the interactive functionality and technology support required, and the depth and breadth of information resources that you will need to fill out the site and meet your reader's expectations. If you are contracting out the production of the Web site, you will also

need to interview and select a site design firm. Ideally, your site designers should be involved as soon as possible in the planning discussions.

Site production checklist

Not every site will require consideration of every item below. Developers within corporations or other large enterprises can often count on substantial in-house technology support when creating new Web sites. If you are on your own as an individual or small business, you may need to contract with various technology and design vendors to assemble everything you'll need to create a substantial content site or small e-commerce site.

Production

- Will your site production team be composed of in-house people, outside contractors, or a mix of the two?
- Who will manage the process?
- Who are your primary content experts?
- Who will be the liaison to any outside contractors?
- Who will function long-term as the Webmaster or senior site editor?
-

2.5.4.3 Technology

- What browsers and operating systems should your site support?
    - Windows, Macintosh, UNIX, Linux
    - Netscape Navigator, Internet Explorer; minimum version supported
- Network bandwidth of average site visitors
    - Internal audience or largely external audience?
    - Ethernet or high-speed connections typical of corporate offices
    - ISDN, or DSL medium-speed connections typical of suburban homes
    - Modem connections for rural, international, or poorer audiences
- Dynamic HTML (HyperText Markup Language) and advanced features?
    - JavaScript or vbscript required
    - Java applets required
    - Style sheets required
    - Third-party browser plug-ins required
    - Special features of the UNIX or NT server environments required
    - Special security or confidentiality features required
- How will readers reach the support personnel?
    - Email messages from readers
    - Chat rooms, forums, help desks, or phone support
- Database support?
    - User log-ins required to enter any site areas?
    - Questionnaires required?
    - Search and retrieval from databases needed?
- Audiovisual content
    - Video or audio productions?

2.5.4.4 Web server support

- In-house Web server or outsourced to Internet Service Provider (ISP)?
    - Unique domain names available (multihoming)

- Disk space or site traffic limitations or extra costs
- Adequate capacity to meet site traffic demands?
- Twenty-four-hour, seven-days-a-week support and maintenance?
- Statistics on users and site traffic?
- Server log analysis: in-house or outsourced?
- Search engine suitable for your content?
- CGI (Common Gateway Interface), programming, and database middleware support available?
- Database support or coordination with in-house staff?

Budgeting

- Salaries and benefits for short-term development staff and long-term editorial and support staff
- Hardware and software for in-house development team members
- Staff training in Web use, database, Web marketing, and Web design
- Outsourcing fees
  - Site design and development
  - Technical consulting
  - Database development
  - Site marketing
- Ongoing personnel support for site
  - Site editor or Webmaster
- Ongoing server and technical support
- Database maintenance and support
- New content development and updating

Appoint a site editor

A site that is "everyone's responsibility" can quickly become an orphan. A maintenance plan should specify who is responsible for the content of each page in the site. To maintain consistent editorial, graphic design, and management policies you'll also need one person to act as the editor of the overall Web site. The site editor's duties will vary according to how you choose to maintain your site. Some editors do all the work of maintaining site content, relieving their coworkers of the need to deal directly with Web page editing. Other editors coordinate and edit the work of many contributors who work directly on the site pages. If multiple people contribute to site maintenance, the site editor may choose to edit pages after they are created and posted to avoid becoming a bottleneck in the communications process. However, high-profile public pages or pages that contain very important content should be vetted by the editor before public posting.

In addition to ensuring editorial quality, a site editor must also ensure that the content of the site reflects the policies of the enterprise, is consistent with local appropriate use policies, and does not contain material that violates copyright laws. Many people who post pictures, cartoons, music files, or written material copied from other sites on their own sites do not understand copyrights and the legal risks in using copyrighted materials inappropriately. A site editor is often an institution's first line of defense against an expensive lawsuit over the misuse of protected material.

2.5.4.5 Organizing information

Our day-to-day professional and social lives rarely demand that we create detailed architectures of what we know and how those structures of information are linked. Yet without a solid and logical organizational foundation, your Web site will not function well even if your basic content is accurate, attractive, and well written. Cognitive psychologists have known for decades that most people can hold only about four to seven discrete chunks of information in short-term memory. The way people seek and use reference information also suggests that smaller, discrete units of information are more functional and easier to handle than long, undifferentiated tracts.

There are five basic steps in organizing your information:

1. Divide your content into logical units
2. Establish a hierarchy of importance among the units
3. Use the hierarchy to structure relations among units
4. Build a site that closely follows your information structure
5. Analyze the functional and aesthetic success of your system

"Chunking" information

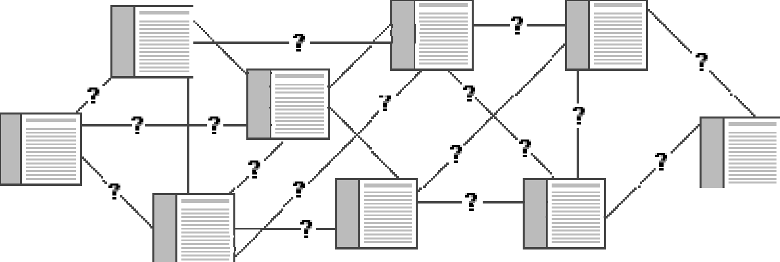
Most information on the World Wide Web is gathered in short reference documents that are intended to be read nonsequentially. This is particularly true of sites whose contents are mostly technical or administrative documents. Long before the Web was invented, technical writers discovered that readers appreciate short "chunks" of information that can be located and scanned quickly. This method for presenting information translates well to the Web for several reasons:

- Few Web users spend time reading long passages of text on-screen. Most users either save long documents to disk or print them for more comfortable reading.
- Discrete chunks of information lend themselves to Web links. The user of a Web link usually expects to find a specific unit of relevant information, not a book's worth of general content. But don't overly subdivide your information or you will frustrate your readers. One to two pages (as printed) of information is about the maximum size for a discrete chunk of information on the Web.
- Chunking can help organize and present information in a uniform format. This allows users not only to apply past experience with a site to future searches and explorations but also to predict how an unfamiliar section of a Web site will be organized.
- Concise chunks of information are better suited to the computer screen, which provides a limited view of long documents. Long Web pages tend to disorient readers; they require users to scroll long distances and to remember what is off-screen.

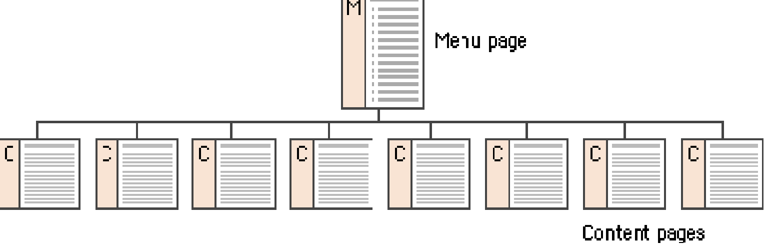
The concept of a chunk of information must be flexible and consistent with common sense, logical organization, and convenience. Let the nature of the content suggest how it should be subdivided and organized. At times it makes sense to provide long documents as a subdivided and linked set of Web pages. Although short Web documents are usually preferable, it often makes little sense to divide a long document arbitrarily, particularly if you want users to be able to print easily or save the entire document in one step.

2.5.4.6 Hierarchy of importance
Hierarchical organization is virtually a necessity on the Web. Most sites depend on hierarchies, moving from the most general overview of the site (the home page), down through increasingly specific submenus and content pages. Chunks of information should be ranked in importance and organized by the interrelations among units. Once you have determined a logical set of priorities, you can build a hierarchy from the most important or general concepts down to the most specific or detailed topics.

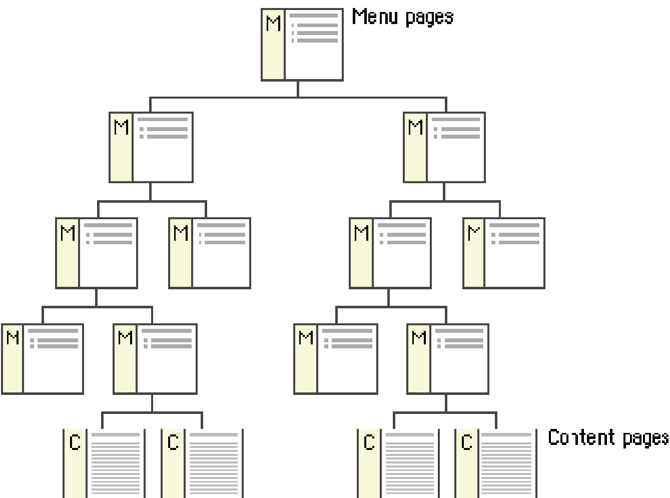
2.5.4.7 Relations
When confronted with a new and complex information system, users build mental models. They use these models to assess relations among topics and to guess where to find things they haven't seen before. The success of the organization of your Web site will be determined largely by how well your system matches your users' expectations. A logical site organization allows users to make successful predictions about where to find things. Consistent methods of displaying information permit users to extend their knowledge from familiar pages to unfamiliar ones. If you mislead users with a structure that is neither logical nor predictable, they will be frustrated by the difficulties of getting around. You don't want your users' mental model of your Web site to look like this:



2.5.4.8 Function
Once you have created your site, analyze its functionality. Efficient Web site design is largely a matter of balancing the relation of menu, or home, pages with individual content pages. The goal is to build a hierarchy of menus and pages that feels natural to users and doesn't mislead them or interfere with their use of the site.
Web sites with too shallow a hierarchy depend on massive menu pages that can degenerate into a confusing "laundry list" of unrelated information:



Menu schemes can also be too deep, burying information beneath too many layers of menus. Having to navigate through layers of nested menus before reaching real content is frustrating:



If your Web site is actively growing, the proper balance of menus and content pages is a moving target. Feedback from users (and analyzing your own use of the site) can help you decide if your menu scheme has outlived its usefulness or has weak areas. Complex document structures require deeper menu hierarchies, but users should never be forced into page after page of menus if direct access is possible. With a well-balanced, functional hierarchy you can offer users menus that provide quick access to information and reflect the organization of your site.

Site elements

Web sites vary enormously in their style, content, organization, and purpose, but all Web sites that are designed primarily to act as information resources share certain characteristics.

2.5.4.9 Section contents

Home pages

All Web sites are organized around a home page that acts as a logical point of entry into the system of Web pages in a site. In hierarchical organizations, the home page sits at the top of the chart, and all pages in the Web site should contain a direct link back to the home page. The World Wide Web URL for a home page is the Web "address" that points users to the Web site. In many cases, home page addresses are used more than home and business street addresses.

Menus and subsites

Unless your site is small you will probably need a number of submenu pages that users enter from a general category listing on your home page. In complex sites with multiple topic areas it is not practical to burden the home page with dozens of links — the page grows too long to load in a timely manner, and its sheer complexity may be off-putting to many users. Providing a submenu page for each topic will create a mini-home page for each section of the site. For specialized, detailed submenus you could even encourage frequent users to link there directly. In this way the submenus will become alternate home pages in "subsites" oriented to a specific audience. Be sure to include a

basic set of links to other sections of the site on each subsite home page, and always include a link back to your main organization home page

Resource lists, "other related sites" pages

The World Wide Web is growing so rapidly that even the large commercial Web index services such as Yahoo! and Excite are only partial listings of the information accessible through the Web. When authors begin to build Web sites, their first page is often a collection of favorite links to sites related to their profession, industry, or interests. In a corporate or institutional site, a well-edited, well-maintained "Other useful sites" page may be the most valuable and heavily used resource.

Site guides

Unlike print media, where the physical heft and dimensions of a book or magazine give instant cues to the amount of information to expect, Web sites often give few explicit indications of the depth and extent of the content available. This is especially true when the home page does not provide an extensive listing of internal site links. Although search facilities offer users quick access to your content, they are no substitute for a clear, well-organized exposition of your site's contents. Even the best search engines are relatively stupid and have only the most primitive means of assessing the priority, relevance, and interrelations of the information resources you offer in your Web site.

"**What's new?" pages**

Many Web sites need to be updated frequently so that the information they present doesn't become stale. But the presence of new information may not be obvious to readers unless you make a systematic effort to inform them. If items that appear on your home page menu are updated, you could place a "new" graphic next to each updated item. If, however, your site is complex, with many levels of information spread over dozens (or hundreds) of pages, you might consider making a "What's New" page designed specifically to inform users of updated information throughout the site.

Search features

The search software you use will often dictate the user interface for searching. If you update your content frequently, be sure that your search engine's indexing is done at least daily. Also be sure that your readers understand exactly what content is being searched: the entire Web site or just a subsection? If your site is complex you may wish to offer readers a pop-up menu that lists the areas of your site and allows them to limit their search to a specific area. And make sure that the results page also matches the graphic design of the site.

Contact information and user feedback

The Web is a bidirectional medium — people expect to be able to send you comments, questions, and suggestions. Always provide at least one link to an email address in a prominent location in your site. You can request user information and feedback using Web page forms and then use a database to store and analyze their input.

Bibliographies and appendixes

The concept of "documents" in electronic environments like the Web is flexible, and the economics and logistics of digital publishing make it possible to provide information without the costs associated with printing paper documents. Making a report available to colleagues on paper usually means printing a copy for each person, so costs and practicality dictate that paper reports be concise and with limited supporting material. Bibliographies, glossaries, or appendixes that might be too bulky to load into a task force report or committee recommendations document can instead be placed in a Web site, making the information available to colleagues as needed.

FAQ pages

The Web and other Internet-based media have evolved a unique institution, the FAQ or "frequently asked questions" page, where the most commonly asked questions from users are listed along with answers. FAQ pages are ideal for Web sites designed to provide support and information to a working group within an institution or to a professional or trade group that maintains a central office staff. Most questions from new users have been asked and answered many times before. A well-designed FAQ page can improve users' understanding of the information and services offered and reduce demands on your support staff.

Custom server error pages

Most Web users are familiar with the "404 error, file not found" screens that pop up on the screen when a Web server is unable to locate a page. The file may be missing because the author has moved or deleted it, or the reader may simply have typed or copied the URL of the page incorrectly. One mark of a really polished Web site is custom-designed and useful error and server message pages. Most standard error screens are generic, ugly, and uninformative. A well-designed error screen should be consistent with the graphic look and feel of the rest of the Web site. The page should offer some likely explanations for the error, suggest alternatives, and provide links to the local home page, site index, or search page

Summary

The most important step in planning your site is to organize your information. Thinking carefully about what you want to say and how you want to say it requires that you become intimately acquainted with your site content. Create outlines, chunk your information into sections and subsections, think about how the sections relate to one another, and create a table of contents. This exercise will help immensely when it comes time to build the individual pages of your site and may determine the eventual success of your Web site.

A well-organized table of contents can be a major navigation tool in your Web site. The table is more than a list of links — it gives the user an overview of the organization, extent, and narrative flow of your presentation

Model questions

1) How are web pages designed?
2) What is the technology used to design web pages?
3) Explain the various elements of a site ?How are sites related?

References:

Fleming, Jennifer. 1998. *Web navigation: Designing the user experience.* Sebastopol, Calif.: O'Reilly.

Harrower, Tim. 1998. *The newspaper designer's handbook, 4th ed.* Boston: McGraw-Hill.

Krug, Steve. 2001. *Don't make me think! A common sense approach to Web usability.* Indianapolis, Ind.: Que.

Nielsen, Jakob. 1995. *The alertbox: Current issues in Web usability.* http://www.useit.com/alertbox.

Author

B.Asha smitha
Centre for biotechnology
ANU