Unit-2

MODULE

- Computer Arithmetic
- Numerical treatment of differential equations
- Numerical linear algebra
- High performance linear algebra

Computer Arithmetic

- Integers
- Real numbers
- Round-off error analysis
- Compilers and round-off

Computer Arithmetic

- Numbers are the lifeblood of statistics, and computational statistics relies heavily on how numbers are represented and manipulated on a computer.
- Computer hardware and statistical software handle numbers well, and the methodology of computer arithmetic is rarely a concern.
- Of the various types of data that one normally encounters, the ones we are concerned with in the context of scientific computing are the numerical types:

Continue..

- integers (or whole numbers) -2,-1,0,1,2...
- real numbers 0, 1,-1.5, 2/3, √2, log 10....
- complex numbers 1 + 2i, √3 - √5 i...
- Computer memory is organized to give only a certain amount of space to represent each number, in multiples of bytes, each containing 8 bits.
- Typical values are 4 bytes for an integer, 4 or 8 bytes for a real number, and 8 or 16 bytes for a complex number.

Continue..

- Since only a certain amount of memory is available to store a number, it is clear that not all numbers of a certain type can be stored.
- Therefore, any representation of real numbers will cause gaps between the numbers that are stored. Calculations in a computer are sometimes described as finite precision arithmetic.

Integers

- In scientific computing, most operations are on real numbers. Computations on integers rarely add up to any serious computation load1.
- It is mostly for completeness that we start with a short discussion of integers.
- Integers are commonly stored in 16, 32, or 64 bits, with 16 becoming less common and 64 becoming more and more.

Continue..

- So, The main reason for this increase is not the changing nature of computations, but the fact that integers are used to index arrays.
- As the size of data sets grows (in particular in parallel computations), larger indices are needed.
- For instance, in 32 bits one can store the numbers zero through $2^{32} -1 \approx 4.10^9$.

Continue..

- In other words, a 32 bit index can address 4 gigabytes of memory. Until recently this was enough for most purposes; these days the need for larger data sets has made 64 bit indexing necessary.
- When we are indexing an array, only positive integers are needed.
- In general integer computations of course, we need to accommodate the negative integers too.

Continue..

- discuss several strategies for implementing negative integers.
  - Our motivation here will be that arithmetic on positive and negative integers should be as simple as on positive integers only and also it usable for bitstring integers.

Continue..

- There are several ways of implementing negative integers. The simplest solution is to reserve one bit as a sign bit, and use the remaining 31 bits to store the absolute magnitude.
- By comparison, we will call the straightforward interpretation of bitstring unsigned integers.

bitstring	$00\cdots0$	$\ldots$	$01\cdots1$	$10\cdots0$	$\ldots$	$11\cdots1$
interpretation as unsigned int	0	$\ldots$	$2^{31}-1$	2^{31}	$\ldots$	$2^{32}-1$
interpretation as signed integer	0	$\cdots$	$2^{31}-1$	-0	$\cdots$	$-(2^{31}-1)$

Continue..

- This scheme has some disadvantages, one being that there is both a positive and negative number zero.
- This means that a test for equality becomes more complicated than simply testing for equality as a bitstring.
- More importantly, in the second half of the bitstrings, the interpretation as signed integer decreases, going to the right.

Continue..

- This means that a test for greater-than becomes complex; also adding a positive number to a negative number now has to be treated differently from adding it to a positive number.
- Another solution would be to let an unsigned number n be interpreted as n-B where B is some plausible base, for instance 2^{31}.

bitstring	$00\cdots0$	$\ldots$	$01\cdots1$	$10\cdots0$	$\ldots$	$11\cdots1$
interpretation as unsigned int	0	$\ldots$	$2^{31}-1$	2^{31}	$\ldots$	$2^{32}-1$
interpretation as shifted int	-2^{31}	$\ldots$	-1	0	$\ldots$	$2^{31}-1$

Continue..

- This shifted scheme does not suffer from the ±0 problem, and numbers are consistently ordered.
- However, if we compute n - n by operating on the bitstring that represents n, we do not get the bitstring for zero. To get we rotate the number line to put the pattern for zero back at zero.
- The resulting scheme, which is the one that is used most commonly, is called 2’s complement.

Continue..

- Using this scheme, the representation of integers is formally defined as follows.
- If $0 \leq m \leq 2^{31} - 1$, the normal bit pattern for m is used.
- for, $-2^{31} \leq n \leq -1$, n is represented by the bit pattern for $2^{32} - |n|$
- The following diagram shows the correspondence between bitstrings and their interpretation as 2's complement integer:.

bitstring	$00\cdots0$	$\ldots$	$01\cdots1$	$10\cdots0$	$\ldots$	$11\cdots1$
interpretation as unsigned int	0	$\ldots$	$2^{31}-1$	2^{31}	$\ldots$	$2^{32}-1$
interpretation as 2's comp. integer	0	$\cdots$	$2^{31}-1$	-2^{31}	$\cdots$	-1

Continue..

- Some observations:
  - There is no overlap between the bit patterns for positive and negative integers, in particular, there is only one pattern for zero.
  - The positive numbers have a leading bit zero, the negative numbers have the leading bit set.

Real numbers

- In this section we will look at how real numbers are represented in a computer, and the limitations of various schemes.
- The next section will then explore the ramifications of this for arithmetic involving computer numbers.

They’re not really real numbers

- numbers in a computer have only a finite number of bits, most real numbers can not be represented exactly.
- In fact, even many fractions can not be represented exactly, since they repeat; for instance, 1/3 = 0.333 ....

Continue..

- An illustration of this is given in appendix

```
#include <stdlib.h>
#include <stdio.h>
int main(){
  double x,div1,div2;
  scanf("%lg",&x);
  div1 = x/7; div2 = (7*x)/49;
  printf("%e %2.17e %2.17e\n",x,div1,div2);
  if (div1==div2) printf("Lucky guess\n");
  else printf("Bad luck\n");
}
```

Representation of real numbers

- Real numbers are stored using a scheme that is analogous to what is known as ‘scientific notation’, where a number is represented as a significant and an exponent,
- for instance $6.022 .10^{23}$, which has a significant 6022 with a radix point after the first digit, and an exponent 23.

Continue..

- This number stands for

$$6.022 \cdot 10^{23} = [6 \times 10^0 + 0 \times 10^{-1} + 2 \times 10^{-2} + 2 \times 10^{-3}] \cdot 10^{23}.$$

- We introduce a base, a small integer number, 10 in the preceding example, and 2 in computer numbers, and write numbers in terms of it as a sum of t terms:

$$x = \pm 1 \times \left[d_1\beta^0 + d_2\beta^{-1} + d_3\beta^{-2} + \cdots + d_t\beta^{-t+1}b\right] \times \beta^e = \pm\Sigma_{i=1}^{t} d_i\beta^{1-i} \times \beta^e$$

Continue..

Where the components are :

- the sign bit: a single bit storing whether the number is positive or negative;
- β is the base of the number system;
- $0 \le d_i \le \beta - 1$ the digits of the mantissa or significant – the location of the radix point (decimal point in decimal numbers) is implicitly assumed to the immediately following the first digit;
- t is the length of the mantissa;
- $e \in [L,U]$ exponent; typically $L < 0 < U$ and $L \approx U$.

Continue..

- Note:
  - there is an explicit sign bit for the whole number; the sign of the exponent is handled differently.
  - For reasons of efficiency, e is not a signed number; instead it is considered as an unsigned number in excess of a certain minimum value.
  - For instance, the bit pattern for the number zero is interpreted as e = L.

Continue..

- Let us look at some specific examples of floating point representations.
- Base 10 is the most logical choice for human consumption, but computers are binary, so base 2 predominates there.
- Old IBM mainframes grouped bits to make for a base 16 representation.

	β	t	L	U
IEEE single precision (32 bit)	2	24	-126	127
IEEE double precision (64 bit)	2	53	-1022	1023
Old Cray 64 bit	2	48	-16383	16384
IBM mainframe 32 bit	16	6	-64	63
packed decimal	10	50	-999	999
Setun	3			

Limitations

- we use only a finite number of bits to store floating point numbers, not all numbers can be represented.
- The ones that can not be represented fall into two categories: those that are too large or too small and those that fall in the gaps.
- The largest number we can store is

$$(\beta-1)\cdot 1+(\beta-1)\cdot\beta^{-1}+\cdots+(\beta-1)\cdot\beta^{-(t-1)}=\beta-1\cdot\beta^{-(t-1)},$$

Continue..

- and the smallest number is $-(\beta - \beta^{-(t-1)}$;
- anything larger than the former or smaller than the latter causes a condition called overflow.
- The number closest to zero is $\beta^{-(t-1)} \cdot \beta^{L}$.
- A computation that has a result less than that (in absolute value) causes a condition called underflow.
- The fact that only a small number of real numbers can be represented exactly is the basis of the field of round-off error analysis.

The IEEE 754 standard for floating point numbers

- Some decades ago, issues like the length of the mantissa and the rounding behaviour of operations could differ between computer manufacturers, and even between models from one manufacturer.
- This was obviously a bad situation from a point of portability of codes and reproducibility of results.

Continue..

- The IEEE standard 75434 codified all this, for instance stipulating 24 and 53 bits for the mantissa in single and double precision arithmetic, using a storage sequence of sign bit, exponent, mantissa.
- The standard also declared the rounding behaviour to be correct rounding: the result of an operation should be the rounded version of the exact result.

Continue..

- But we have seen the phenomena of overflow and underflow, that is, operations leading to unrepresentable numbers.
- There is a further exceptional situation that needs to be dealt with: what result should be returned if the program asks for illegal operations such as $\sqrt{-4}$?
- The IEEE 754 standard has two special quantities for this: Inf and NaN for 'infinity' and 'not a number'.
- Infinity is the result of overflow or dividing by zero, not-a-number is the result of, for instance, subtracting infinity from infinity.
- The rule for computing with Inf is a bit more complicated

Continue..

- An inventory of the meaning of all bit patterns in IEEE 754 double precision is given in figure

sign	exponent	mantissa
s	$e_1 \cdots e_8$	$s_1 \ldots s_{23}$
31	$30 \cdots 23$	$22 \cdots 0$

$(e_1 \cdots e_8)$	numerical value
$(0 \cdots 0) = 0$	$\pm 0.s_1 \cdots s_{23} \times 2^{-126}$
$(0 \cdots 01) = 1$	$\pm 1.s_1 \cdots s_{23} \times 2^{-126}$
$(0 \cdots 010) = 2$	$\pm 1.s_1 \cdots s_{23} \times 2^{-125}$
$\cdots$	
$(01111111) = 127$	$\pm 1.s_1 \cdots s_{23} \times 2^{0}$
$(10000000) = 128$	$\pm 1.s_1 \cdots s_{23} \times 2^{1}$
$\cdots$	
$(11111110) = 254$	$\pm 1.s_1 \cdots s_{23} \times 2^{127}$
$(11111111) = 255$	$\pm\infty$ if $s_1 \cdots s_{23} = 0$, otherwise NaN

Round-off error analysis

- Numbers that are too large or too small to be represented, leading to overflow and underflow, are uncommon: usually computations can be arranged so that this situation will not occur.
- By contrast, the case that the result of a computation between computer numbers is not representable is very common.
- Thus, looking at the implementation of an algorithm, we need to analyze the effect of such small errors propagating through the computation.
- This is commonly called round-off error analysis.

Correct rounding

- The IEEE 754 standard, mentioned in above section, does not only declare the way a floating point number is stored, it also gives a standard for the accuracy of operations such as addition, subtraction, multiplication, division.
- The model for arithmetic in the standard is that of correct rounding: the result of an operation should be as if the following procedure is followed:
  - The exact result of the operation is computed, whether this is representable or not.
  - This result is then rounded to the nearest computer number.

Continue..

- In a decimal number system with two digits in the mantissa, the computation $1.0 - 9.4 \cdot 10^{-1} =$

  $1.0 – 0.94 = 0.06 = 0.6 \cdot 10^{-2}$
- Note that in an intermediate step the mantissa 0.94 appears, which has one more digit than the two we declared for our number system. The extra digit is called a guard digit.
- Without a guard digit, this operation would have proceeded as $1.0 - 9.4 \cdot 10^{-1}$
- where $9.4 \cdot 10^{-1}$ would be rounded to 0.9, giving a final result of 0.1, which is almost double the correct result.

Addition

- Addition of two floating point numbers is done in a couple of steps. First the exponents are aligned: the smaller of the two numbers is written to have the same exponent as the larger number.
- Then the mantissas are added. Finally, the result is adjusted so that it again is a normalized number.

Continue..

- As an example, consider $1.00 + 2.00 \times 10^{-2}$.
- Aligning the exponents, this becomes $1.00 + 0.02 = 1.02$,and this result requires no final adjustment.
- We note that this computation was exact, but the sum $1.00 + 2.55 \times 10^{-2}$ has the same result, and here the computation is clearly not exact:

  the exact result is 1.0255,which is not representable with three digits to the mantissa.

Continue..

- In the example $6.15 \times 10^1 + 3.98 \times 10^1 = 10.13 \times 10^1 = 1.013 \times 10^2 \rightarrow 1.01 \times 10^2$ we see that after addition of the mantissas an adjustment of the exponent is needed.
- The error again comes from truncating or rounding the first digit of the result that does not fit in the mantissa:
- if x is the true sum and ~x the computed sum, then $\tilde{x} = x(1 + \epsilon)$ where, with a 3-digit mantissa $|\epsilon| < 10^3$.

Continue..

- Formally, let us consider the computation of $s = x1+x2$, and we assume that the numbers x_i are represented as $\tilde{x}_i = x_i(1 + \epsilon i)$. Then the sum s is represented as

$$\begin{aligned}
\tilde{s} &= (\tilde{x}_1 + \tilde{x}_2)(1 + \epsilon_3) \\
&= x_1(1 + \epsilon_1)(1 + \epsilon_3) + x_2(1 + \epsilon_2)(1 + \epsilon_3) \\
&\approx x_1(1 + \epsilon_1 + \epsilon_3) + x_2(1 + \epsilon_2 + \epsilon_3) \\
&\approx s(1 + 2\epsilon)
\end{aligned}$$

- under the assumptions that all ϵi are small and of roughly equal size, and that both $xi > 0$. We see that the relative errors are added under addition.

Multiplication

Floating point multiplication, like addition, involves several steps. In order to multiply two numbers $m_1 \times \beta^{e_1}$ and $m_2 \times \beta^{e_2}$, the following steps are needed.

- The exponents are added: $e \leftarrow e_1 + e_2$.
- The mantissas are multiplied: $m \leftarrow m_1 \times m_2$.
- The mantissa is normalized, and the exponent adjusted accordingly.

For example: $1.23 \cdot 10^0 \times 5.67 \cdot 10^1 = 0.69741 \cdot 10^1 \rightarrow 6.9741 \cdot 10^0 \rightarrow 6.97 \cdot 10^0$.

Subtraction

- Subtraction behaves very differently from addition. Whereas in addition errors are added, giving only a gradual increase of overall round off error, subtraction has the potential for greatly increased error in a single operation.
- For example, consider subtraction with 3 digits to the mantissa: $1.24 - 1.23 = 0.01 \rightarrow 1.00 \cdot 10^{-2}$. While the result is exact, it has only one significant digit

Continue..

- example, showing how this can be caused by the rounding behaviour of floating point numbers. Let floating point numbers be stored as a single digit for the mantissa, one digit for the exponent, and one guard digit; now consider the computation of 4 + 6 + 7. Evaluation left-to-right gives:

$$
\begin{aligned}
(4 \cdot 10^0 + 6 \cdot 10^0) + 7 \cdot 10^0 &\Rightarrow 10 \cdot 10^0 + 7 \cdot 10^0 && \text{addition} \\
&\Rightarrow 1 \cdot 10^1 + 7 \cdot 10^0 && \text{rounding} \\
&\Rightarrow 1.0 \cdot 10^1 + 0.7 \cdot 10^1 && \text{using guard digit} \\
&\Rightarrow 1.7 \cdot 10^1 && \\
&\Rightarrow 2 \cdot 10^1 && \text{rounding}
\end{aligned}
$$

Continue..

- On the other hand, evaluation right-to-left gives:

$$
\begin{aligned}
4 \cdot 10^0 + (6 \cdot 10^0 + 7 \cdot 10^0) &\Rightarrow 4 \cdot 10^0 + 13 \cdot 10^0 && \text{addition} \\
&\Rightarrow 4 \cdot 10^0 + 1 \cdot 10^1 && \text{rounding} \\
&\Rightarrow 0.4 \cdot 10^1 + 1.0 \cdot 10^1 && \text{using guard digit} \\
&\Rightarrow 1.4 \cdot 10^1 && \\
&\Rightarrow 1 \cdot 10^1 && \text{rounding}
\end{aligned}
$$

Round off error in parallel computations

- As we discussed in the above example of summing a series, addition in computer arithmetic is not associative.
- A similar fact holds for multiplication. This has an interesting consequence for parallel computations: the way a computation is spread over parallel processors influences the result.
- As a very simple example, consider computing the sum of four numbers a+b+c+d. On a single processor, ordinary execution corresponds to the following associativity:$((a + b) + c) + d$:

Continue..

- On the other hand, spreading this computation over two processors, where processor 0 has a, b and processor 1 has c,d. corresponds to ((a + b) + (c + d)):
- Generalizing this, we see that reduction operations will most likely give a different result on different numbers of processors. (The MPI standard declares that two program runs on the same set of processors should give the same result.)

Continue..

- It is possible to circumvent this problem by replace a reduction operation by a gather operation to all processors, which subsequently do a local reduction.
- this increases the memory requirements for the processors.
- So There is an intriguing other solution to the parallel summing problem.

Continue..

- If we use a mantissa of 4000 bits to store a floating point number, we do not need an exponent, and all calculations with numbers thus stored are exact since they are a form of fixed-point calculation [109, 108].
- While doing a whole application with such numbers would be very wasteful, reserving this solution only for an occasional inner product calculation may be the solution to the reproducibility problem.

Compilers and round-off

- From the above discussion it should be clear that some simple statements that hold for mathematical real numbers do not hold for floating-point numbers. For instance, in floating-point arithmetic
- $(a + b) + c \neq a + (b + c)$:
- This implies that a compiler can not perform certain optimizations without it having an effect on round-off behaviour.

Continue..

- In some codes such slight differences can be tolerated, for instance because the method has built-in safeguards.
- For instance, the stationary iterative methods of damp out any error that is introduced.
- On the other hand, if the programmer has written code to account for round-off behaviour, the compiler has no such liberties.
- This was hinted at in above exercise. We use the concept of value safety to describe how a compiler is allowed to change the interpretation of a computation.

Continue..

- At its strictest, the compiler is not allowed to make any changes that affect the result of a computation.
- Compilers typically have an option controlling whether optimizations are allowed that may change the numerical behaviour.
- For the Intel compiler that is -fp-model=.... On the other hand, options such as -Ofast are aimed at performance improvement only, and may affect numerical behaviour severely.
- For the Gnu compiler full 754 compliance takes the option -frounding-math whereas -ffast-math allows for performance-oriented compiler transformations that violate 754 and/or the language standard.

Continue..

- These matters are also of importance if you care about reproducibility of results.
- If a code is compiled with two different compilers, should runs with the same input give the same output? If a code is run in parallel on two different processor configurations? These questions are very subtle.
- In the first case, people sometimes insist on bitwise reproducibility, whereas in the second case some differences are allowed, as long as the result stays ‘scientifically’ equivalent.
- Of course, that concept is hard to make rigorous. Here are some issues that are relevant when considering the influence of the compiler on code behaviour and reproducibility.

Continue..

- **Re-association** Foremost among changes that a compiler can make to a computation is re-association, the technical term for grouping a + b + c as a + (b + c).
- The C language standard and the C++ language standard prescribe strict left-to-right evaluation of expressions without parentheses, so re-association is in fact not allowed by the standard.
- The Fortran language standard has no such prescription, but there the compiler has to respect the evaluation order that is implied by parentheses.

Continue..

- A common source of re-association is loop unrolling. Under strict value safety, a compiler is limited in how it can unroll a loop, which has implications for performance.

- The amount of loop unrolling, and whether it’s performed at all, depends on the compiler optimization level, the choice of compiler, and the target platform.

- A more subtle source of re-association is parallel execution; we see in section above section. This implies that the output of a code need not be strictly reproducible between two runs on different parallel configurations.

Continue..

- **Constant expressions** It is a common compiler optimization to compute constant expressions during compile time.

  For instance, in

  ```
  float one = 1.;
  ...
  x = 2. + y + one;
  ```

- the compiler change the assignment to x = y+3.. However, this violates the re-association rule above, and it ignores any dynamically set rounding behaviour.

Continue..

- **Expression evaluation** In evaluating the expression a+(b+c), a processor will generate an intermediate result for b+c which is not assigned to any variable. Many processors are able to assign a higher precision of the intermediate result. A compiler can have a flag to dictate whether to use this facility.

- **Behaviour of the floating point unit** Rounding behaviour (truncate versus round-to-nearest) and treatment of gradual underflow may be controlled by library functions or compiler options.

- **Library functions** The IEEE 754 standard only prescribes simple operations; there is as yet no standard that treats sine or log functions. Therefore, their implementation may be a source of variability