Notes from January 26 – Thursday

- Project 3 requires you to program the summation of two mathematical series. The first step is to obtain the recurrence relations of the two series. The program can be done without the recurrence by simply using factorials, but your credit received would be 80% because of the inefficiency of the program.

- The steps for the program are (a) Initialize the first term, initialize the index; initialize the total to equal the first term as an intermediate total. (b) Create a counter loop without the aid of conditional statements. In Matlab, the “for loop” can be used. The syntax is “for i=bc:inc:ec” and the end of the loop structure is the statement “end”. The label “bc” is the beginning count of index i, “inc” is the increment of the index and “ec” is the end count of the index i. The loop would repeat until the index enumeration is completed. (c) Inside the loop, update the index, calculate the new term using the recurrence relation and the previous term, update the intermediate total by doing “total=total+term”, a familiar expression in computer programming to update the state of the total variable. This is standard practice, it is not a mystery.

- One example which would help you understand the concept of an intermediate sum is the case of a low-cost calculator with one display. If the sequence of keys of “1+3+5+7=” are entered, the display after each keystroke is: (1),(1),(3),(4),(5),(9),(7),(16). The (4), (9) and (16) are the intermediate sums. As you can see, a sum using a sequentially programmed code requires intermediate sums to be produced. Since it is likely that we are not interested in the intermediate sums, it makes sense to overwrite the value of sum as a new one is obtained. Likewise, it is fine to use just one memory to store a new term because the old one would not be needed again.

- The above paragraph uses the codes: “term=term*rr” and “total=total+term” and those erase earlier values (states) of the variables “term” and “total”. It is assumed that the intermediate values are not important and that is why it is fine to erase them. If the terms are to be kept, use an indexed array.

- The program instruction “term=term*rr” indicates that the existing value of a term in the mathematical series is fetched from the memory, fetch the recurrence relationship from the memory, multiply
the two values to obtain a new term and then store that to the same memory location “term”, thus erasing an earlier value.

- The program instruction “total=total+term” means that the intermediate “total” and the newly calculated “term” are fetched from the memory, added, then stored back into the same memory location “total”, thus erasing the previous intermediate total. This procedure is common in many mathematical algorithms as summation is quite similar to integration in nature.

- As an example, consider the sum of the cosine series using factorials:
  ```
  x=input('Enter x: '); 
  total=1; 
  sign=-1; 
  for i=2:2:20 
      total=total+sign*x^i/factorial(i); 
      sign=-sign 
  end
  ```

- As another example, consider the sum of the sine series using factorials:
  ```
  x=input('Enter x: '); 
  total=0; 
  sign=1; 
  for i=1:2:21 
      total=total+sign*x^i/factorial(i); 
      sign=-sign 
  end
  ```

- To sum the sine series with a recurrence relationship (without factorials), the code could be written as:
  ```
  x=input('Enter x: '); 
  term=x; 
  total=term; 
  for i=3:2:21 
      rr=-x*x/(i*(i-1)); 
      term=term*rr; 
      total=total+term; 
  end
  ```

- Presently, the mathematical algorithms are written as main programs, eventually, math series are normally written as functions to be called by the main program or other subprograms. For example, the “canned” functions like sin(x), cos(x), exp(x), etc, were written to be accessed as parts of a general mathematical package. There is never a need to have the intermediate sum or one of the terms revealed.

- Using matlab, it is not good to write programs with a large number of loops because it is an interpreter; each program instruction is interpreted, again and again, each time around the loop. For a
compiled language, the code is converted to machine language one
time and then there is no more need for interpretation. It is very
likely that library functions were written in a lower level language,
such as C, to make their execution faster.

- A word concerning the level of a language. A high level language
  means it is very close to human language, therefore, it is easier to
  understanding and to apply. A low level language is one that is closer
to the machine, i.e., more difficult for the user to understanding and
therefore not as user friendly. High level languages have more
overhead and therefore it executes slower. Matlab has many “canned”
functions and they are written in lower level languages, so it is not
slow when you call those functions to do your job. But if you have
many loops that are custom made for your algorithm, it would be
better to write the program in a lower level programming language.

- C and Fortran are considered lower level than Matlab. C++ is higher
  level than C so it has more overhead. Visual Basic and C++ are Object
  Oriented Programming (OOP) languages; they are different from
  procedural languages like C, Fortran and Matlab. OOP allows several
  tasks to operate at the same time with relations between the tasks.

- Assembler Language is the closest to the binary machine language.
  Each Assembler Language instruction converts directly into one
  machines instruction. It is awkward to write, but it is the most
  efficient. For example, the two matlab instructions, “term=term*rr”
  and “sum=sum+term” require the value of term to be stored into
  memory and then few instructions later the program fetches the same
  number from the memory to add to the sum. If assembler language is
  used, the value of “term” could be kept in one of the CPU’s floating
  point registers. Higher level languages do not have direct access to
  the CPU registers or the FPU registers. (FPU=Floating Point Processor.)

- An introduction to efficient programming was discussed. Horner’s Rule
  was introduced using the PDF file link
  http://www-classes.usc.edu/engr/ce/108/horner.pdf
  By rearranging the terms using many parentheses, the number of
  multiplication operation can be reduced significantly. The point was
  made that mathematical series should never be summed using
  factorials and high powers of the independent variable.
• The efficiency of FFT (Fast Fourier Transform) was mentioned as a significant contribution to data processing. For a large data set, FFT can be hundreds or even a thousand times faster than convention algorithms. The concept of the binary divide and conquer scheme can be shown using a simple illustration of how a power of x can be determined using very few operations. See http://www-classes.usc.edu/engr/ce/108/xpower.pdf

• In Matlab, a sum can also be performed using a matrix product. A product of two matrices consists of many “inner products” between row vectors and column vectors. Consider two row vectors: \(a=[1 \ 2 \ 3 \ 4]\) and \(b=[3 \ 4 \ 5 \ 6]\). They are not compatible for matrix multiplication because the number of columns in the first matrix must equal the number of rows in the second matrix. The inner product between a and b can be performed as \(a^*b^t\) in which \(b^t\) is the transpose of b. The result would be \((1)(3)+(2)(4)+(3)(5)+(4)(6)=3+8+15+24=50\).

• If the above example was performed as \(a^*b\), then the result would be a 4 X 4 matrix.

• To perform a sum on a polynomial, a matrix product could be used. Let the row vector “a” contains the coefficient of the polynomial; then the Matlab instruction \(n=\text{size}(a,2)\) would yield the number of columns in “a”. If x contains the value of the independent variable, then \(x.^(0:n-1)\) would contain a row vector with powers of x. The sum of the polynomial would then be \(a^*(x.^(0:n-1))'\). For a large value of n, Horner’s rule would be much more efficient, but the Matlab features are very enticing and addictive.