1 Theory

1.1 Stability of iteration

- Show that the iterations:
$$a_n = \frac{2}{3}\,a_{n-1} \qquad \text{with } a_0 = 1 \tag{1}$$
and
$$a_n = 2\,a_{n-1} - \frac{8}{9}\,a_{n-2} \qquad \text{with } a_0 = 1 \text{ and } a_1 = \tfrac{2}{3}. \tag{2}$$
each have the sequence
$$a_n = \left(\frac{2}{3}\right)^n \tag{3}$$
as exact an solution.
- Let us suppose that we make a trunction error specifying the value of a_1 in Eq. (2). That is we take $a_1 = \frac{2}{3} + \varepsilon$. By making the ansatz,
$$a_n = x^n,$$
find the exact solution of Eq. (2) with the perturbed initial condition. Show that the Eq. (3) is an unstable solution of Eq. (2).
- Suppose that we specify a_1 to 4-digit precision and use Eq. (2) to compute the sequence (3). At what value of n does the resulting sequence become incorrect in its leading digit?

1.2 Solving linear recursion relations

Linear recursion relations are very similar to linear ordinary differential equations. Many concepts such as characteristic polynomial, general and particular solutions etc have discrete analogues.

- Find the general solution of the homogeneous linear recursion relation
$$a_n = 5\,a_{n-1} - 6\,a_{n-2} \tag{4}$$
and use it to find the solution which satisfies the starting conditions $a_0 = 2$, $a_1 = 5$.
- Find a particular solution of the inhomogeneous recursion relation
$$a_n = 5\,a_{n-1} - 6\,a_{n-2} + 4\,n, \tag{5}$$
by making an ansatz inspired by the functional form of the inhomogeneous part of the recursion.
- Write down the solution of Eq. (5) with the starting conditions $a_0 = 2$, $a_1 = 5$.

1.3 Computational complexity of mergesort

- Explain why the computational complexity, $F(n)$, of the mergesort algorithm satisfies the recursion
$$F(n) = 2\,F\left(\frac{n}{2}\right) + n, \tag{6}$$
with $F(1) = 1$.
- Introduce the new variable p defined by $n = 2^p$ and let $b_p = F(2^p)$. Show that Eq. (6) in these variables takes the form
$$b_p = 2\,b_{p-1} + 2^p, \tag{7}$$
with $b_0 = 1$.
- Find the solution of the homogeneous version of Eq. (7) which satisfies the starting condition.
- The obvious ansatz for finding a particular solution fails in this case. Explain why and use your knowledge of differential equations to suggest an ansatz for the particular solution which allows you to solve Eq. (7) completely.
- Use your results to show that the asymptotic complexity of the mergesort algorithm is $F(n) = O(n\,\log(n))$. Remember that $x^{\log(y)} = y^{\log(x)}$.

2 Implementation

2.1 Rounding error and loss of significance *

Consider the family of quadratic equations

$$x^2 - 3x + \frac{9}{4}\varepsilon = 0$$

for ε taking the values $\left\{\varepsilon_i = \left(\frac{3}{2}\right)^{-i} : i = 1, 2, \ldots, 40\right\}$. Call the root near zero $x_i^{(-)}$ and the root near 3 $x_i^{(+)}$.

- Show that

$$\begin{aligned} x_i^{\pm} &= \frac{3}{2}\left(1 \pm \sqrt{1-\varepsilon_i}\right) && (8) \\ &= \frac{3}{2}\frac{\varepsilon_i}{1 \mp \sqrt{1-\varepsilon_i}}. && (9) \end{aligned}$$

- Write a program to compute $x_i^{(-)}$ and $x_i^{(+)}$ using single precision arithmetic (float in C/C++) using both Eq. (8) and Eq. (9) for each value of ε_i. Denote the result by $\mathrm{SP}\left[x_i^{(\pm)}\right]$.
- Write a program to compute $x_i^{(-)}$ and $x_i^{(+)}$ using double precision arithmetic (double in C/C++). Denote the result by $\mathrm{DP}\left[x_i^{(\pm)}\right]$. For the purposes of measuring rounding errors in the single precision calculation, we can treat the double precision results as "exact".
- Plot the relative errors

$$\Delta_i = \frac{\left|\mathrm{SP}\left[x_i^{(\pm)}\right] - \mathrm{DP}\left[x_i^{(\pm)}\right]\right|}{\mathrm{DP}\left[x_i^{(\pm)}\right]},$$

as a function of ε_i for the method (8) and (9). Explain what you observe.

2.2 Linear search using a linked list

Download the sample code timer.tgz from the class website. It contains everything you need to get started running C++ code on the CSC taskfarm. It contains an implementation of an insertion sort and a wrapper routine which measures the time required to sort a list of n random numbers into rank order.

- Learn how to uncompress and compile the code on godzilla and submit it for execution on the taskfarm.
- Use the sample code to verify that the insertion sort is an $O(n^2)$ algorithm

Download the sample code linkedlist.tgz from the class website. It contains a basic C++ implementation of a linked list which can store integers. The class contains methods which can insert an integer at the head of the list, insert an integer at the tail of the list and print the list contents.

- Modify the List class to store a real number, x, and an integer, k, at each node.
- Add a method which computes the sum of the x's and the k's stored in the list.
- Cannibalise the timer code to produce a list, $\{x_i\}$, of n uniformly distributed random numbers in the interval $[0, 1]$ and calculate the associated list, $\{s_i\}$, of partial sums. Store the list of keys, $\{k_i\}$ and the list of partial sums in the linked list.
- Add a method to the List class which takes a uniformly distributed random number, y, in the interval $[0, s_n]$ and returns the key value, k_i, of the list element which satisfies $s_{i-1} \leq y < s_i$. Provide some sample output to demonstrate that your code works.
- Use the timer code to verify that the search algorithm which you have just built runs in $O(n)$ time.

2.3 Binary tree search *

Download the sample code binarytree.tgz from the class website. It contains a basic C++ implementation of a Fenwick binary tree which can store key-value pairs consisting of an integer, k, and an associated real number x. The class BinaryTree contains a method which builds the Fenwick tree recursively from a list of real numbers, $\{x_i\}$. The main program reads in an integer n from the command line and builds a tree containing the first n multiples of π.

- Add a method to the class which prints the contents of the tree. Run the code for $n = 8$ and $n = 10$ and draw sketches indicating which values are stored at which nodes in the tree in each case.
- Add the method

```
Datum BinaryTree::search(BinaryTree *T, double x);
```

which implements the binary search algorithm which we discussed in class: given the list of partial sums, $\{s_i\}$, and a real number y in the $[0, s_n]$, returns the Datum object stored at the node which satisfies $s_{i-1} \leq y < s_i$. Provide some sample output to demonstrate that your code works.

- Use the timer code to verify that the search algorithm which you have just built runs in $O(\log(n))$ time on arrays of random numbers. You will need to do multiple iterations of the same search to measure a detectable search time since the algorithm is very fast.