MATH 561: Numerical Analysis I

Instructor: Prof. Wolfgang Bangerth
bangerth@colostate.edu

Partial answers for homework assignment 5

Problem 1 (Convergence order for sequences). Since we know that

$$\|x_n\| \approx C\|x_{n-1}\|^r,$$

we know that in particular

$$\|x_3\| \approx C\|x_2\|^r,$$
$$\|x_2\| \approx C\|x_1\|^r,$$

and similarly for all other successive three numbers. We can therefore compute r by taking the ratio of the two equations:

$$\frac{\|x_3\|}{\|x_2\|} \approx \frac{\|x_2\|^r}{\|x_1\|^r},$$

which immediately gives us

$$r \approx \frac{\log \frac{\|x_3\|}{\|x_2\|}}{\log \frac{\|x_2\|}{\|x_1\|}}.$$

Once we have r, we can compute C via

$$C \approx \frac{\|x_3\|}{\|x_2\|^r}.$$

In practice, convergence is often so that we only know that

$$\|x_n\| = C\|x_{n-1}\|^r + \text{higher order terms},$$

i.e., the approximate equality is due to the higher order terms (which may be of the form $C'\|x_{n-1}\|^{r+1} + C''\|x_{n-1}\|^{r+2} + \cdots$). These terms become smaller and smaller compared to $C\|x_{n-1}\|^r$ the smaller x_{n-1} is, i.e., the approximate equality gets closer and closer to an equality the further we go down the sequence. As a consequence, one typically gets the best estimates for C and r by using the last three data points of the sequence.

For the examples in question, we get the following values:

(a) $C = \frac{4}{9}, r = 1$;

(b) $C = 0.8, r = 1$;

(c) $C = 1, r = 1.5$;

(d) $C = 1.01, r = 1.1$.

With these numbers, it takes the sequences approximately 25, 85, 7, and 90 terms to get below 10^{-8}. The exact number will depend on how accurately, and from which terms of the sequence, you determined C and r.

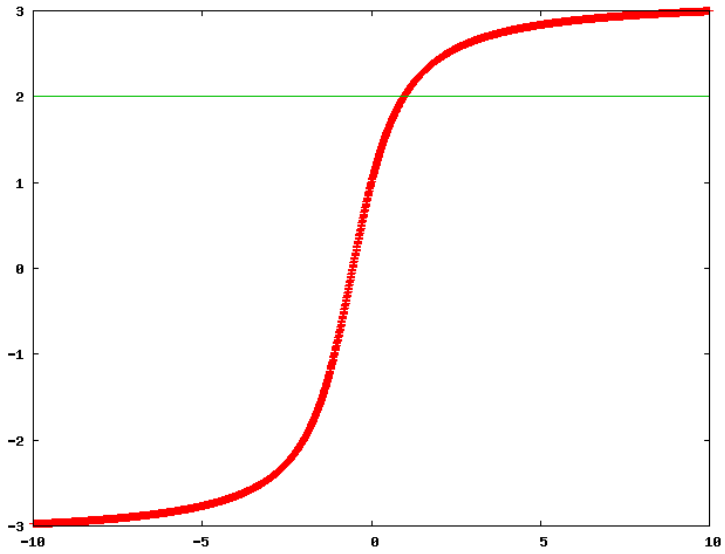
Problem 2 (Secant method). Here is a small program that implements the function $g(x)$:

```
#include <iostream>
#include <cmath>

double g (const double x)
{
  double a = 1;
  for (unsigned int i=1; i<=10; ++i)
    a = a + (x*std::cos(a) + x)/10;
  return a;
}


int main ()
{
  for (double x=-100; x<100; x+=.01)
    std::cout << x << ' ' << g(x) << std::endl;
}
```

It creates output that consists of pairs $(x, g(x))$ for values of x equally spaced between -1 and +1, and we can visualize these to the following graph that shows the values of the function:



The picture is definitely helpful in understanding where the zero of $f(x)$, i.e., the point where $g(x) = 2$, is going to lie: somewhere between zero and two.

Using this function, we can apply the secant method where we start with $x_0 = 0$ and $x_1 = 2$ because the point where the secant intersects the x-axis is going to already be pretty close to where we expect the root to be. Here is the little program:

```
#include <iostream>
#include <cmath>

double g (const double x)
{
  double a = 1;
```

```
  for (unsigned int i=1; i<=10; ++i)
    a = a + (x*std::cos(a) + x)/10;
  return a;
}

double f (const double x)
{
  return g(x) - 2;
}



int main ()
{
  std::cout.precision(16);

  double x_k          = 1;
  double x_k_minus_1 = 0;

  for (unsigned int iteration=2; iteration<=20; ++iteration)
    {
      double new_x = x_k - f(x_k)/(f(x_k)-f(x_k_minus_1))*(x_k-x_k_minus_1);
      x_k_minus_1 = x_k;
      x_k          = new_x;

      std::cout << "Iteration=" << iteration
                << ", best guess=" << new_x
                << std::endl;
    }
}
```

The output of this program is as follows:

```
Iteration=2, best guess=0.9775780807522463
Iteration=3, best guess=0.9634318705356449
Iteration=4, best guess=0.9636650901748191
Iteration=5, best guess=0.9636636171736778
Iteration=6, best guess=0.9636636170179089
Iteration=7, best guess=0.9636636170179093
Iteration=8, best guess=0.9636636170179093
Iteration=9, best guess=-nan
Iteration=10, best guess=-nan
```

In other words, within just 4 iterations the method has solved to six digits of accuracy – in fact, in the fourth iteration, we already have 9 digits correct. (The program is not efficient: it evaluates $f(x_{k-1})$ in each iteration, although we already know that value from the previous iteration. We could make the program more efficient by storing this value.)

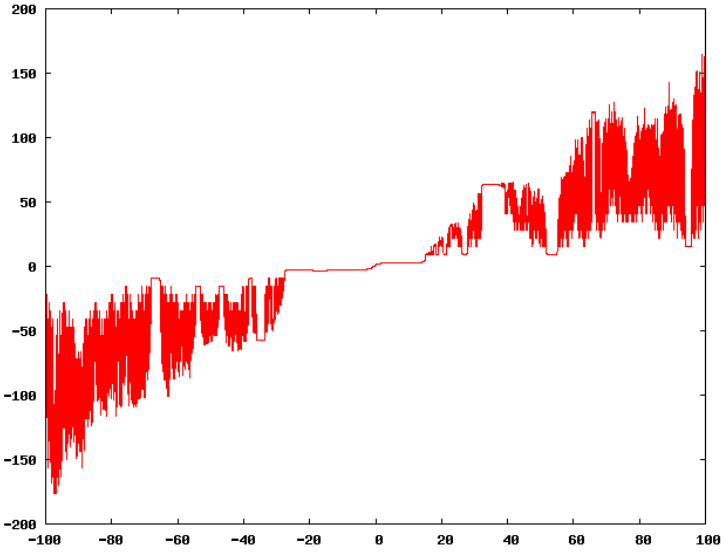
In iteration 9, something obviously goes wrong: this is because we divide by $f(x_k) - f(x_{k-1})$, and because x_k is so close to x_{k-1}, this difference is zero. A practical program would catch this, and simply abort the loop.

Instead of writing a program for the bisection method, let us simply state this observation: in each iteration of the bisection method, we gain a factor of two in accuracy. Starting with an interval length of

one, the error in iteration k is then $\frac{1}{2^k}$. To get this to be below 10^{-6} then requires about 19 iterations – far more than the bisection method.

There remains the question of why we can't use Newton's method? We could – we would just have to find $f'(x)$, and that will be tedious given the form of the function. (To see why it is possible, imagine that the function $g(x)$ only did two iterations; then the result of the first iteration – which explicitly depends on x could be inserted into the second iteration, resulting in some complicated formula that could, nevertheless, be differentiated with respect to x. Doing ten iterations changes nothing fundamentally, except that it makes the overall expression as a function of x unwieldy and probably too complicated in practice to do anything useful with. So, while we could use Newton's method *in theory*, in practice the tedium of computing the derivative means that we'd rather use the secant method.)

As a final thought: The function I gave you looks quite nice on the interval $[-10 : 10]$ as shown in the figure – well behaved, smooth, bounded, everything you want from a function. But if you extend the interval a bit, it starts to look very different:



This is an example of a function that starts to become chaotic outside a rather narrow range of function values.

Problem 3 (Root finding methods). Of the three methods, the bisection method will always converge to a root as long as we can find starting points a_0, b_0 so that the sign of $f(\cdot)$ is different for the two starting points. Neither the Newton method nor the secant method can provide this guarantee: both require that one starts close enough to the solution; because the secant method approximates the derivative that is used in Newton's method, one would expect that the radius of convergence for the secant method is slightly smaller than that of the Newton method, but not so that it makes a significant difference in practice.

On the flip side, both the Newton and secant methods converge quadratically or at least superlinearly, whereas the bisection method only converges linearly (i.e., *much* slower). If one were to really quantify things, it turns out that the secant method converges with a convergence order of about 1.6, but each iteration is only half as expensive as a Newton iteration because we do not need to evaluate the derivative – and so, *per function evaluation*, the secant method is actually significantly faster.