MATH 561: Numerical Analysis I

Instructor: Prof. Wolfgang Bangerth
bangerth@colostate.edu

Answers for homework assignment 1

Problem 1 (Continuous vs. discrete).

a) The following C++ program computes these numbers. For the largest representable number, we start with one and multiply it by a fixed factor until the `isinf` function tells us that the result is too large to be represented. Alternatively, if you don't want to use the magic `isinf` function, you can also just output the number and visually inspect when the program starts to print infinities. For the smallest number, we again start with one and divide it by a certain factor until the result is zero (meaning that the number is not representable any more with the number of bits this system uses to represent non-zero numbers). Since the question only asked for an approximation and since division is so much slower on today's computers than multiplication, we choose a larger reduction factor than for the largest representable number.

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

int main ()
{
  // largest number for double
  {
    const double factor = 1.0001;
    double large = 1;
    double next_large = large*factor;
    while (!isinf(next_large))
      {
        large = next_large;
        next_large = large*factor;
      }
    std::cout << "Largest double precision="
              << large << std::endl;
  }


  // largest number for float
  {
    const float factor = 1.0001;
    float large = 1;
```

```
    float next_large = large*factor;
    while (!isinf(next_large))
      {
        large *= factor;
        next_large = large*factor;
      }
    std::cout << "Largest single precision="
              << large << std::endl;
  }

  // smallest number for double
  {
    // we would like to use the same
    // reduction factor as above, 1.0001, but
    // if one does so one realizes that
    // floating point divisions are _way_
    // slower than floating point
    // multiplications. since we don't want
    // to wait quite as long, we choose a
    // coarser reduction factor
    const double factor = 2;
    double small = 1;
    double next_small = small/factor;
    while (next_small != 0)
      {
        small /= factor;
        next_small = small/factor;
      }
    std::cout << "Smallest double precision="
              << small << std::endl;
  }

  // smallest number for float
  {
    const float factor = 2;
    float small = 1;
    float next_small = small/factor;
    while (next_small != 0)
      {
        small /= factor;
        next_small = small/factor;
      }
    std::cout << "Smallest single precision="
              << small << std::endl;
  }
}
```

The output of this program is

```
homework-01> c++ problem-1a.cc -o problem-1a
homework-01> ./problem-1a
Largest double precision=1.79769e+308
Largest single precision=3.40267e+38
Smallest double precision=4.94066e-324
Smallest single precision=1.4013e-45
```

The use of variables such as `next_large` instead of using the corresponding expression right in the `while` condition seems odd. For those interested, the reason is that Intel compatible processors have a historic quirk: they compute things in their registers to a greater accuracy than what they can store in variables kept in memory. For example, the smallest number that can be added to one without resulting in one (see problem 1b below) is around 10^{-19} instead of 10^{-16}. In order to determine the "real" accuracy of double precision numbers, one therefore always has to force intermediate computations into memory by assigning them to variables. (This quirk is no longer the case in newer 64-bit processors with 64-bit operating systems, but if you have an older 32-bit operating system, you may still run into this issue.)

For practical matters, this quirk is annoying but not terribly important. The worst that can happen is that programs compiled with optimization yield results that differ in the last significant digit of results.

b) Use the following program. It repeatedly tests $1 + eps$ against 1, and if the two numbers are not equal, it reduces *eps* by a factor of two.

```
#include <iostream>

int main ()
{
  // for double
  {
    double eps = 1;
    double one = 1;
    double one_plus_eps;

    one_plus_eps = one + eps;
    while (one_plus_eps != one)
      {
        eps = eps/2;
        one_plus_eps = one + eps;
      }
```

```
    // one_plus_eps is now one, so we have to
    // go one step back
    eps = eps*2;
    std::cout << "double precision epsilon="
              << eps << std::endl;
  }


  // same for float
  {
    float eps = 1;
    float one = 1;
    float one_plus_eps;

    one_plus_eps = one + eps;
    while (one_plus_eps != one)
      {
        eps = eps/2;
        one_plus_eps = one + eps;
      }

    eps = eps*2;
    std::cout << "single precision epsilon="
              << eps << std::endl;
  }
}
```

The output is

```
homework-01> c++ problem-1b.cc -o problem-1b
homework-01> ./problem-1b
double precision epsilon=2.22045e-16
single precision epsilon=1.19209e-07
```

c) No, there are no such floating point numbers that satisfy the right hand side of the linear system of equations exactly. However, this isn't a problem, because the right hand side of the second equation isn't exactly representable in floating point arithmetic anyway: in floating point arithmetic using double precision numbers, $1+10^{20} = 10^{20}$; with this right hand side, the floating point solution is again $x_1 = x_2 = 1$.

Problem 2 (Floating point vs real numbers). In floating point arithmetic, the expressions have the following values:

$$(1+\tfrac{\varepsilon}{2})-1 = 1-1 = 0$$
$$1+(\tfrac{\varepsilon}{2}-1) = 1+(-1) = 0$$
$$(1-1)+\tfrac{\varepsilon}{2} = 0+\tfrac{\varepsilon}{2} = \tfrac{\varepsilon}{2}$$

In contrast to exact arithmetic, floating point arithmetic therefore does not have the property that associativity holds, i.e. $a+b+c \neq a+c+b$.

Problem 3 (Associativity of addition). The following little program runs the summation with varying batch sizes (a batch size of one corresponds to the original sum):

```
#include <iostream>
using namespace std;

void harmonic(unsigned int batch_size)
{
  float sum = 0.;
  float last = -1.;
  unsigned int k = 1;
  while (sum != last)
    {
      last = sum;
      float sum2 = 0.;
      for (unsigned int i=0;i<batch_size;++i,++k)
        sum2 += 1./k;
      sum += sum2;
    }
  cout << "Iterations=" << k
       << ", sum=" << sum << std::endl;
}


int main()
{
  harmonic (1);
  harmonic (10);
}
```

The output of the program, after running for about 0.1 seconds, is

```
homework01> c++ harmonic.cc
homework01> ./a.out
Iterations=2097153, sum=15.4037
Iterations=10485771, sum=17.0396
```

In other words, we do get farther in our sum if we do batch data. Both results, however, are completely wrong because the exact sum should of course be infinite.

The reason for the wrong result is that after some time, the sum $S_N = \sum_{k=1}^{N} \frac{1}{k}$ is already large, and adding a term $\frac{1}{N+1}$ as the next summand that by comparison is very small, will not yield a result that is different from S_N in floating point arithmetic. If we batch terms, then the terms in the inner sum are of comparable size and so the inner sum can be computed with relatively good accuracy; however, when we add the inner sum to the terms already computed for the outer sum, we run into the same problem after some time: the outer sum is so much larger than the inner one that adding a small number does no longer change the large one.

The upshot of this exercise is that adding smaller and smaller numbers to large ones can not be done accurately in floating point arithmetic. A stable way to add numbers is to start with the smallest terms and then add larger and larger terms to it. This is always possible for finite sums, but of course doesn't help if as in this exercise you have an infinite sum.

Problem 4 (Taylor series). Derive the first four terms and integral remainder term of the Taylor series of

a)

$$\sin x = x - \tfrac{1}{6}x^3 + E_4(x)$$
$$E_4(x) = \frac{1}{4!}\int_0^x (\cos t)(x-t)^4 dt$$

b)

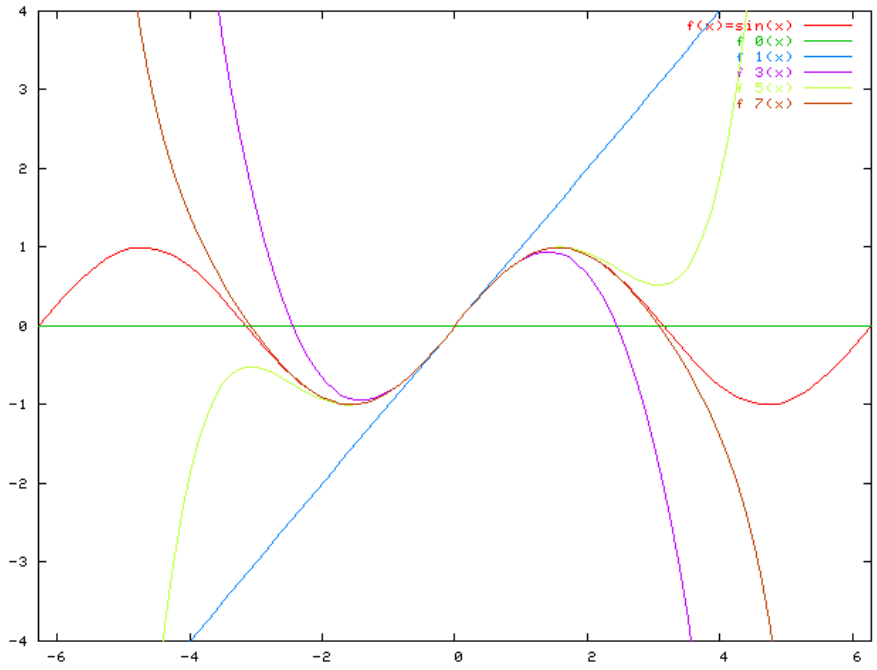
$$x\sin x = \frac{\pi}{2} + \left(x - \frac{\pi}{2}\right) - \frac{\pi}{4}\left(x - \frac{\pi}{2}\right)^2 - \frac{1}{2}\left(x - \frac{\pi}{2}\right)^3 + \frac{\pi}{48}\left(x - \frac{\pi}{2}\right)^4 + E_4(x)$$
$$E_4(x) = \frac{1}{4!}\int_1^x (t\cos t + 5\sin t)(x-t)^4 dt$$

c) The expansion of $f(x) = 4(x-3)^2(x+2)$ equals $f(x)$ (although one may write it as an expanded, rather than a factorized polynomial). There is no remainder term, because the fourth derivative of $f(x)$ that would appear in it is zero (this is a cubic polynomial after all). This means that the Taylor expansion with only four terms is an exact representation of the original function.

d)

$$x^x = 1 + (x-1) + (x-1)^2 + \frac{1}{2}(x-1)^3 + \frac{1}{3}(x-1)^4 + E_4(x)$$
$$E_4(x) = \frac{1}{4!}\int_1^x \left[\frac{d^5}{dt^5}t^t\right](x-t)^4 dt$$

Problem 5 (Taylor series). When one plots the original function along with the first few Taylor expansion terms, one gets this picture:



Note that all even derivatives of $f(x)$ at $x = 0$ are zero, so $f_2(x) = f_1(x)$, $f_4(x) = f_3(x)$, etc.

As can be seen from the graph, successive Taylor approximations provide a better and better match reaching out from the point of expansion, $x_0 = 0$: while f_1 is only a reasonable match for $|x| < \frac{\pi}{4}$, f_7 is already a rather good match for $|x| < \pi$. However, none of the shown function are anywhere close to $f(x)$ at the right margin $x = 2\pi$. One would apparently need many more terms until one gets a good approximation to $f(2\pi)$ using a Taylor series.

To check this assertion, let us use the following program:

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

int main ()
{
  double n_factorial = 1;
  double taylor_approximation_at_x = 0;

  std::cout << std::setprecision(16);
```

```
  // evaluate Taylor approximations at x=2*pi
  double x = M_PI*2;

  for (int n=1; n<50; ++n)
    {
      n_factorial *= n;

      // evaluate f'; it is zero for all even
      // n, and +-1 for all other cases
      double f_prime_at_x0;
      if (n % 2 == 0)
        f_prime_at_x0 = 0;
      else
        {
          if (n % 4 == 1)
            f_prime_at_x0 = 1;
          else
            f_prime_at_x0 = -1;
        }

      // now add the new term to the previous
      // ones
      taylor_approximation_at_x += 1./n_factorial * f_prime_at_x0
                                   * std::pow (x, n);


      std::cout << "f_" << n << "(x) = "
                << taylor_approximation_at_x
                << std::endl;
    }
}
```

If we run it, we get this sequence (note again that every second term is equal to the previous one, because the corresponding Taylor series term is zero):

$$
\begin{aligned}
f_1(x) &= 6.283185307179586 \\
f_2(x) &= 6.283185307179586 \\
f_3(x) &= -35.05851693322017 \\
f_4(x) &= -35.05851693322017 \\
f_5(x) &= 46.54673234285487 \\
f_6(x) &= 46.54673234285487 \\
f_7(x) &= -30.15912741020649 \\
f_8(x) &= -30.15912741020649 \\
f_9(x) &= 11.89956653469115 \\
f_{10}(x) &= 11.89956653469115 \\
f_{11}(x) &= -3.195076042131838 \\
f_{12}(x) &= -3.195076042131838 \\
f_{13}(x) &= 0.6248765427164428 \\
f_{14}(x) &= 0.6248765427164428 \\
f_{15}(x) &= -0.09324575906205741 \\
f_{16}(x) &= -0.09324575906205741 \\
f_{17}(x) &= 0.01098340314608236 \\
f_{18}(x) &= 0.01098340314608236 \\
f_{19}(x) &= -0.001048182796038258 \\
f_{20}(x) &= -0.001048182796038258 \\
f_{21}(x) &= 8.274095221353732e-05 \\
f_{22}(x) &= 8.274095221353732e-05 \\
f_{23}(x) &= -5.494383778892663e-06 \\
f_{24}(x) &= -5.494383778892663e-06 \\
f_{25}(x) &= 3.112686240572392e-07 \\
f_{26}(x) &= 3.112686240572392e-07 \\
f_{27}(x) &= -1.522421074247229e-08 \\
f_{28}(x) &= -1.522421074247229e-08 \\
f_{29}(x) &= 6.49459795770924e-10 \\
f_{30}(x) &= 6.49459795770924e-10 \\
f_{31}(x) &= -2.437611221052859e-11 \\
f_{32}(x) &= -2.437611221052859e-11 \\
f_{33}(x) &= 8.151523463872753e-13
\end{aligned}
$$

To achieve the desired accuracy of 10^{-4}, we therefore need 21 terms. For an accuracy of 10^{-12} we need 33 terms.

Note that while $n!$ is really an integer, it should be computed using a `double` variable, since already 13! is not representable as an integer any more on machines that have 32-bit integers.

Problem 6 (Gaussian elimination). The computation is long and tedious and isn't repeated here. The result is

$$x = \begin{pmatrix} -64 \\ 900 \\ -2520 \\ 1820 \end{pmatrix}$$

The fact that the computation was tedious was the point of this exercise – it's not that hard to find the solution of a 3×3 system on a piece of paper, but it's *much* more work to do the same for a 4×4 system. This illustrates the difficulty with Gaussian elimination: it's run-time just grows unacceptably quickly.

The Hilbert matrix is pretty badly conditioned. The 4×4 matrix already has a condition number $\kappa_\infty = 28375$, and this is growing very quickly as we increase the size: for the 6×6 matrix, we have $\kappa_\infty = 2.9 \cdot 10^7$ and for 10×10 the condition number is $\kappa_\infty = 3.5 \cdot 10^{13}$. With such a large condition number, numerical solution of linear systems with these matrices become very unstable.

Problem 7 (Gaussian elimination). There are a number of ways to compute the inverse. One way is to solve for the vectors z_i that satisfy the set of n equations

$$Az_i = e_i,$$

where e_i are the unit vectors with a single one in row i and zeros everyone else. If one has all these vectors z_i, it is easy to see that z_i is the ith column of A^{-1}. (This is so, because $z_i = A^{-1}e_i$, and the multiplication of a matrix, here A^{-1} with the ith unit vector yields its ith column – do the experiment with a matrix X for which you have all the entries and multiply it with $e_1, e_2, \ldots$.)

A more elegant way is to compute the LU decomposition $A = LU$, then $A^{-1} = (LU)^{-1} = U^{-1}L^{-1}$. L and U are both triangular matrices for which it is simple to determine the inverses.

For the present matrix, the inverse is

$$A^{-1} = \begin{pmatrix} 16 & -120 & 240 & -140 \\ -120 & 1200 & -2700 & 1680 \\ 240 & -2700 & 6480 & -4200 \\ -140 & 1680 & -4200 & 2800 \end{pmatrix}.$$

Problem 8 (Gaussian elimination). The process shown in class breaks down because there is a zero entry on the diagonal, by which we would have to divide. That doesn't work, of course.

However, remember that there are three things you can do to a linear system that leave the solution unchanged: (i) multiplication of a single equation by a scalar, (ii) addition of a multiple of one equation to another equation, and (iii) exchanging two equations. The Gaussian elimination process, just as the LU decomposition, only used rules (i) and (ii). In order to make the system in the problem solvable, we have to use the third rule, by sorting the equations in order (2), (3), (1). This yields the same matrix as at the beginning of the Problem, though with a different right hand side. The matrix is now in a form that is readily invertible.

The process of picking an order of the equations is usually called "pivoting".