Below is a list of problems for our lab session

Wednesday, November 12th, (kl. 13.00-16.00), Room MA436-446.

A number of MATLAB functions have been uploaded to the class website. Download the newest site backup

http://www8.cs.umu.se/kurser/5DV005/HT14/Programs/

and copy it into appropriate directory in your account as you did last week. Be advised that new functions will be added continuously during the class as they are written. Make sure that you always have the latest version. Let me know the instant you find have verified the presence of a bug, so that the problem can be fixed.

**Remark 1** This version corrects four typographical errors in the original version dated November 9th. They were all located in Problem 1. There was also an unintended bug in `wy.m` which was revealed during the lab. I have retained the bug and uploaded a new subroutine `weyland_yutani.m` which does not this problem.
Problem 1 Function bad_wolf is a very, very bad implementation of the natural exponential function

\[ f(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \]  

which has been written by somebody who understands that

\[ f(x) = \lim_{n \to \infty} \sum_{i=0}^{n} \frac{x^i}{i!} \]

but has no real appreciation of the difference between real arithmetic and floating point arithmetic!

1. Explain to your friends how the source code works. In particular, relate the update of the variable term to the fact that

\[ \frac{x^{j+1}}{(j+1)!} = \left( \frac{x^j}{j!} \right) \frac{x}{j+1} \]

2. Do a simple test of bad_wolf by plotting \((x,y)\) where 

\[
x = \text{linspace}(-30,30,1025); \quad y = \text{bad_wolf}(x);
\]

The graph is looking good, but appearances can be deceiving!

3. The real function \(x \to e^x\) is always strictly positive, but bad_wolf returns negative values, which is very, very bad! Verify, that exactly 75 of the 1025 values returned by bad_wolf are negative! The find command is very useful and idx = find(y<0) will return the indices of the negative values.

4. Determine the sign of the entries of \(x\) for which the corresponding entries of \(y\) are all negative. Use the sign command together with the array \(x(idx)\) where \(idx\) are the indices defined in question 3.

The problems in bad_wolf are limited to negative values of the input argument. There is nothing wrong with the output of bad_wolf for positive values of the input argument. There is a function called good_wolf which calls bad_wolf, exploiting the trivial identity

\[ e^x = \frac{1}{e^{-x}} \]

in order to avoid working directly with negative values of the argument.

5. Explain the source code of good_wolf to your friends. In particular, it is important that you understand how the negative and positive arguments are isolated and how equation (4) is exploited.

6. Return to the test case used in question 2. Compare good_wolf to the built-in function exp in MATLAB. Verify that absolute value of the relative error

\[ \frac{e^x - \text{good_wolf}(x)}{e^x} \]

is bounded by \(24u\) where \(u = 2^{-54}\) is the unit round off error in double precision. You will find the commands max and abs useful here.
Problem 2 Consider the function $f : \mathbb{R} \to \mathbb{R}$ given by

$$f(x) = \begin{cases} \frac{e^x - e^{-x}}{2x}, & x \neq 0, \\ 1, & x = 0. \end{cases}$$

(6)

A naive MATLAB implementation is simply

```matlab
f=@(x)(exp(x)-exp(-x))./(2*x)
```

In this problem, we explore the failure of such a naive approach and how to diagnose and fix the problem.

1. Plot the naive implementation of $f$ on the interval $[a, b] = [-2^{-k}, 2^{-k}]$ using 1025 equidistant points and $k = 20$. Verify that the graph looks like Figure 1. In particular, notice the labeling along the y-axis is completely useless!

![Figure 1: A plot of the naive implementation of the function $f$ on the interval $[-2^{-20}, 2^{-20}]$](image)

2. Does this graph meet with your approval? In particular, what happens when you zoom further in using $k = 22$. Is this likely to be the graph of a nice differentiable function which changes only slowly?

3. Use l’Hospital’s rule to verify that

$$f(x) \to 1, \quad x \to 0, \quad x \neq 0.$$  

(7)

4. Verify, that naive implementation at least gives the correct limit for small values of the absolute value of $x$. 


The fundamental problem with the naive implementation is that the two real numbers
\[ a = a(x) = e^x \quad \text{and} \quad b = b(x) = e^{-x} \]
are very close to each other for small values of \( x \). Therefore, there is no guarantee that the computer can evaluate
\[
d(x) = a(x) - b(x)
\]
with a small relative error and the computer is taking this opportunity to produce garbage. It falls to your to force it to produce meaningful numbers.

5. Show that the function \( f \) is well conditioned in the vicinity of \( x = 0 \) by explicitly evaluating the condition number
\[
\kappa_f(x) = \left| \frac{x f'(x)}{f(x)} \right| \quad \text{for} \quad x \neq 0.
\]
You will find
\[
\kappa_f(x) = \left| \frac{\cosh(x)x}{\sinh(x)} - 1 \right| \to 0, \quad x \to 0, \quad x \neq 0,
\]
so \( f \) is actually a very well conditioned function in the vicinity of \( x = 0 \).

Discovering that \( f \) is well conditioned is very reassuring. We now know that it is at least not theoretically impossible to compute \( f \) accurately!

6. Use Taylor series to prove that
\[
f(x) = \sum_{j=0}^{\infty} \frac{x^{2j}}{(2j + 1)!}
\]
\textbf{Hint:} Use the Taylor series for the function \( x \to e^x \).

7. Show that catastrophic cancellation is a not an issue outside of the interval \([-\alpha, \alpha]\) where
\[
\alpha = \frac{\log(2)}{2}.
\]
\textbf{Hint:} Recall that any subtraction \( d = a - b \) is perfectly safe if \( |a| > 2|b| \). In your situation, you have to distinguish between \( x > 0 \) and \( x < 0 \), but that is only a small complication, right?

8. A function \texttt{wy.m} is available on the class website which can be used to compute \( f \) accurately. Unfortunately, critical portions of the code have been lost and the function can not even run. Fix all the problems using the insight that you have just gained and compare you work to the function \texttt{g=@(x)sinh(x)./x}.

9. Verify that relative error of \texttt{wy} compared with \texttt{g} is bounded by \( 4u \) on the interval used in question 1. Moreover, compare the naive implementation of \( f \) to the function \( g \) and verify that the relative error is bounded by \( 9.4 \times 10^7u \approx 10^8u \).

\textbf{Remark 2} Take note of the fact that a little thought on your part has reduced the relative error by a factor of about 25,000,000 from the naive implementation of \( f \) to the finished version of \texttt{wy.m}

\footnote{In fact there was a very bad bug in \texttt{wy.m} which the routine to fail, whenever the input did not contain any values in the interval \([-\alpha, \alpha]\). This issue is fixed in the new version of the function \texttt{weyland_yutani.m}.}
**Problem 3** During Lab Session 1 you executed the bisection algorithm manually in order to hit a target at a given distance. It is time to automate this process. There is a general implementation of the bisection algorithm on the class website. The function is called `bisection`.

1. Read through the documentation of bisection and execute the minimal working example to verify that the code makes sense.

In order to automatically hit a target at coordinates \((d,0)\) we have to set up a suitable function which can be passed to the bisection algorithm. The following steps takes you through this process

2. Let \(d\) denote the distance to the target. Define a global variable \(d\) using the command `global d`. Set \(d=12345\).

3. Use the minimal working example of `range_rkx.m` to initialize a gun and define a range function using those values, i.e.

   \[
   \text{range}=@(\theta)\text{range_rkx}(v0,\theta,\text{method},dt,\text{maxit})
   \]

4. Similarly, define the residual function \(f(\theta) = r(\theta) - d\) in MATLAB. The target will be destroyed if \(|f(\theta)| \leq \rho\), where \(\rho = 30\) is the kill radius (in meters) of the shell.

At this point we have defined a continuous function \(f\) such that \(f(\theta) = 0\) implies that we hit the target exactly. In order to use the bisection method, we need to find a bracket, i.e. a pair of angles \(\theta_1\) and \(\theta_2\) such that \(f(\theta_1)\) and \(f(\theta_2)\) have different sign. In terms of physics, this corresponds to firing a pair of shells, where one passes over the target and the other falls short of the target. Suitable brackets can be found if a ballistic table has already been computed.

5. Recompute a suitable ballistic table using the function `artillery_table` from the class website or your own equivalent version from Lab Session 1,

   \[
   \text{deg}=0:1:90; \ \text{theta}=(\text{deg}*(\pi/180));
   \text{table}=[\text{artillery_table}(v0,\text{theta},\text{method},\text{dt},\text{maxit})];
   \]

6. Explain to your friends, how `find(table(2,:)>d)` can be used to rapidly determine a bracket!

7. Compute the elevation necessary to hit the target. Xenomorphs are very sturdy, so we want a residual which is less than 1 meter, rather than the usual value of 30 meters. Moreover, we want both the high and the low angle. Use the function `range_rkx` to generate the trajectories and plot them in the same figure. You should end up with a figure which looks like Figure 2.

8. A UAV (Unmanned Aerial Vehicle, i.e. a drone) has spotted a group of xenomorphs hiding behind a large hill. Your friend has analyzed the telemetry and gives you the new range. Compute the high and the low trajectory. Explain why the high trajectory is the probably the only viable option.
Problem 4 The function newton contains an elementary implementation of Newton’s method for solving the non-linear equation
\[ f(x) = z, \quad (12) \]
using an initial guess \( x = x_0 \) to start the iteration.

1. Read through the documentation of newton and run the first of the two minimal working examples. Explain to your friends why the minimal working examples demonstrates how to compute the square root of a given number.

2. Compute \( x = \log(2) \) by solving the non linear equation (12) where \( f(x) = e^x \) and \( z = 2 \), and initial guess \( x_0 = 0.7 \). Verify that the number of correct digits is being doubled from one iteration to the next.

3. Run Newton’s method again using a bad initial guess, i.e. \( x_0 = 100 \). Manually adjust maxit until you converge and flag=1 is returned.

4. Examine the sequence of approximations contained in the variable his. Explain to your friends why these numbers are initially decreasing like \( 100, 99, 98, \ldots \).

5. Repeat the previous question, but with an extremely bad initial guess of \( x_0 = 1000 \). You will not converge and you will encounter both “NaN” and “Inf” which is the machines representation of “Not A Number” and \( \infty \).

Remark 3 Newton’s method is an extremely efficient method to solving non linear equations and computing inverse functions. However, the rapid convergence comes at a price! You have to pick a good initial guess and if you are sloppy, then there is no guarantee of success.