Teknisk vetenskabliga beräkningar, Fall 2014
Lab problems

Carl Christian Kjelgaard Mikkelsen

November 25, 2014

Below is a list of problems for our lab session

Wednesday, November 24th (kl. 13.00-16.00), Room MA436-446.

As usual there are new files in the site back-up and some typo's have been fixed. Moreover, there was a single index error which threw of the off the flight time calculated by the `range_rk*` famlily of function by exactly one time step. It is likely that I will start maintaining a git repository instead of issuing these updates as zip files.

Problem 1 (Maximal range of artillery) Find the maximal range of the gun defined by the minimal working example of `range_rkx2` by solving the equation

$$r'(\theta) = 0 \tag{1}$$

with respect to θ using the bisection algorithm as implemented in `bisection2.m`.

Hint: You will be able to extract the derivative from your extended artillery table function `my_eat.m` by defining a function

```
f=@(theta)[0 0 0 1]*my_eat(v0,theta,method,dt,maxstep);
```

and then feeding `f` to the bisection algorithm along with a suitable bracket. Make sure that everything else is defined before you define the auxiliary function `f`.

Problem 2 (Locating the apex of a shell's trajectory) Shells can be fired "manually" and tracked for T seconds by first defining the initial condition and the calling the function `rk.m` which computes the trajectory using one of several methods. The commands are very simple. The command

```
y0=[0; 0; v0*cos(theta); v0*sin(theta)];
```

puts the muzzle at $(0, 0)$ and sets the elevation to θ. The command

```
[t, tra]=rk(@shell4,0,T,y0,N,1,'rk2');
```

fires the shell, tracking it for T seconds, using timestep T/N. Compute the apex of the shell's trajectory when a shell is fired by the gun defined by the minimal working example of `range_rkx2` and an initial elevation of 60 degrees.

Hint: You should write your own function `my_apex.m` which calls `rk` and extracts `tra(k,end)`. Then `my_apex` can be feed to the bisection algorithm.

Remark 1 You are always looking for a way to verify that your numbers make sense, right? If `tra` is a trajectory, then

```
plot(tra(1,:),tra(2,:),tra(1,1:k:end),tra(2,1:k:end),'*')}
```

puts a star at every `k`th timestep.

Problem 3 (Succes and failure of interpolation) Functions `cnf.m` and `enf.m` for computing and evaluation the Newton form of the interpolating polynomial are now available on the website.

1. (Basic sanity check) Construct the interpolating polynomial p of degree at most 10 which interpolates $f(x) = \sin(x)$ on $k = 11$ equidistanct nodes on the interval $[-\pi, \pi]$. The absolute error will be small, but there just might be a problem at the endpoints.

2. (Possible failure of interpolation) Increase k to $k = 21$ and $k = 31$ and you will get into trouble at the endpoints.

3. (Practical application of interpolation) Construct a small artillery table for the gun defined by the minimal working example of `range_rkx2`. Use 10 equidistant nodes, i.e. `deg=0:10:90`. Construct the polynomial which interpolated the range function r on these 10 nodes. Compare the range function with the interpolated value on a few favorite values, say 45 degrees and 36 degrees. You should be pleasantly surprised at the quality of the approximation. Do a more extensive artillery table and plot the result against the values interpolated from the small table.

4. (Runge's function and failure of interpolation) Interpolate the function $f(x) = 1/(1+25x^2)$ on the interval $[-1, 1]$ using k equidistant nodes. Pay particularly attention to the size of the relative error as a function of k. Initially, it will decrease, but as k becomes larger, the error will skyrocket.

5. Return to the polynomial which interpolates the artillery table. Technically, the polynomial has degree 9, but have a close look at the coefficients! They are not all equally large?

Remark 2 Interpolation with a polynomial of high degree is not necessarily a good idea. The only thing which will ensure succes is choosing a few nodes on a short interval. Recall the error formula:

$$f(x) - p(x) = \frac{f^{(n+1)}(\xi)}{(n+1)}\omega(x), \quad \omega(x) = \prod_{i=0}^{n}(x - x_i). \tag{2}$$

Certainly, the derivatives of f might very well be bounded, as in the case of $f(x) = \sin(x)$ or they might become very large as we increase n, but the only thing that will ensure that $\omega(x)$ is small, is if you pick x inside a short interval which contains the nodes x_i. Therefore, if you must deal with a large interval, then break it into small subintervals and use a different polynomial of low degree on each subinterval.

Problem 4 (Evaluation of polynomials) A polynomial p can be evaluated using Horner's method which is discussed in Problem 39 of the auxiliary problems in the directory `http://www8.cs.umu.se/kurser/5DV005/HT14/Notes/`. The method is implemented as `woo.m` which also computes an upper bound on the error, i.e. a number μ, such that

$$|p(x) - \hat{y}| \leq \mu u \tag{3}$$

where $\hat{y}$ is the computed value of $p(x)$ and u is the unit round off.

1. Begin by recalling the problem of applying the bisection algorithm to the equation

   $$g(x) = 0 \tag{4}$$

   where $g(x) = x^3 - 3x^2 + 3x - 1$ is evaluated using Horner's method, i.e.

   $$g(x) = ((x - 3)x + 3)x - 1 \tag{5}$$

   The fundamental problem is that any error made while computing

   $$a(x) = ((x - 3)x + 3)x \tag{6}$$

   is raised to prominence when we carry out the final subtraction

   $$g(x) = a(x) - 1. \tag{7}$$

   Reissue the commands

   ```
   g=@(x)((x-3).*x+3).*x-1;
   x=1+linspace(-1,1,1025)*2^-22;
   plot(x,g(x))
   ```

   in order to refresh your memory of the difficulty of computing the sign of g correctly.

2. Use `woo` to compute g, i.e. `y=woo(coef,x)` where `coef` is a vector of coefficients which define the polynomial g.

   Remark 3 The situation will remain unchanged as we are in fact doing exactly the same operations as before, but we just eliminated any lingering doubt as to what `MATLAB` really does when it evalutes the expression for `g`.

3. The fundamental question is if `woo` can be used to automacially recognize when the computed sign of g is untrust worthy. This is the point were the computed error bound is important. Issue `plot(x,abs(y),x,mu*u)` where $u = 2^{-53}$ is the unit round off error in double precsion and verify that the error estimate is in fact larger than the absolute value of the computed values of g!

4. Explain why this means that we can not be sure that the computed values of $g(x)$ have the correct sign!

5. Redefine `x` until you can find the largest interval $I = [s, t]$ around 1 where the `woo` recognizes that the computed sign of g can not be trusted. A command such as

   ```
   find(abs(y)<mu*2^-53)
   ```

   will probably be helpful.

6. Reissue the commands which will attempt to solve $g(x) = 0$ using the bisection algorithm. If you start with the bracket $a = 0.3$ and $b = 1.3$, then the algorithm will "fail" as the final interval does not even contain the true root. Verify that the failure takes place just inside I, the exact interval were `woo` correctly warns that the computed sign can not be trusted.

Remark 4 You just had an encounter with a technique called running error analysis, where estimates of the rounding error are computed together with the primary objective. This is a very powerful technique, which has fallen into disuse.