MCS 471 Project Two: Müller's Method for Polynomial Roots

The goal of the project is to study the method of Müller to compute all roots of a polynomial. This method is explained in section 1.4 of our textbook.

In our experiments we will use MATLAB or Octave. MATLAB is available in all computer labs on campus, but is commercial and expensive. Octave is similar to MATLAB (for our work, Octave runs just as fine), and is free to download from `http://www.octave.org/`.

1. Polynomials in MATLAB or Octave

Polynomials are represented by their coefficient vectors. For example, the cubic polynomial $3.3x^3 - 2.23x^2 - 5.1x + 9.8$ is entered typing $c = [3.3 \quad -2.23 \quad -5.1 \quad 9.8]$ at the prompt. With $y = \mathbf{polyval}(c, x)$, we evaluate the polynomial at some point or vector x. After typing $r = \mathbf{roots}(c)$, the vector r will contain approximations for the roots of the polynomial. The complement to the **roots** command is **poly**, e.g, $c = \mathbf{poly}(r)$ returns the coefficient vector of the monic polynomial which has its roots in r.

We can group commands into .m files, defining functions. The name of the function should match the file name. For this project, we need the functions "mueller" and "run_mueller" with their definitions in the respective files "mueller.m" and "run_mueller.m" (download "mueller.m" and "run_mueller.m" from the class web site). The scripts have been tested to run both with Octave and MATLAB.

If the path is set correctly (do **help path** otherwise to see how to set the search path), then you can call this function just as a regular command. Typing **help mueller** and **help run_mueller** in an Octave or MATLAB session will show you how to use these functions.

2. Numerical Experiments on Random and Special Polynomials

In the following assignments we study the finding of the roots for four classes of polynomials: random polynomials, polynomials with roots of varying magnitude, multiple roots, and ill-conditioned polynomials.

2.1 Convergence of the Method. We generate random polynomials to investigate experimentally the convergence of the method.

Assignment One. Generate random polynomials with d roots in $[0, 1]$, using **p = poly(rand(1,d))**, for degrees $\mathbf{d} = 3, 4, \ldots, 10$. Take as desired accuracy each time `1.0e-12` and the maximum number of steps equal to 20. Do three random experiments for every d. Record in a table (rows indexed by d and columns by the trial: one, two, three) the number of steps it takes to reach the desired accuracy.

What can you say about the convergence of the method? Make a distinction between the global and local convergence, i.e.: how does the method converge when still far from the roots and how does it converge when already close to the roots?

2.2. Accuracy of the Method. The script "run_mueller" finds all roots of a polynomial by "deflation", i.e.: the routine continues on the quotient after division of the previous polynomial by $x - r$, where r is an approximate root of the polynomial. The residuals on return are the residuals at the quotients, but not at the original polynomial. So the user of "run_mueller" may not immediately know the extent of the accumulation of roundoff that has occurred. Especially if the accuracy requirement is not that small, the roots computed first will be more accurate than the last roots.

Assignment Two. Generate an example, using roots of varying magnitude to show the accumulation of roundoff, by calculation of the residuals of the approximate roots at the *original* polynomial.

Create a script "ref_mueller" which is a copy of "run_mueller" but with a couple of iterations of "mueller" on the **original** *polynomial instead of on the quotient, refining the calculated approximate root. Illustrate on your example which showed the accumulation of roundoff in "run_mueller" that the "ref_mueller" gives more accurate results.*

2.3 Multiple roots. Consider a polynomial which has only one root with multiplicity d at $x = 1$, i.e.: $p(x) = (x-1)^d$.

Assignment Three. Take $p(x) = (x-1)^d$, using the command **p = poly(ones(1,d))**, for **d** $= 3, 4, \ldots, 10$. Run the method as in assignment one, but preferably use the "ref_mueller" of assignment two, eventually allowing more iterations. If you failed to complete assignment two, then you may still use "run_mueller", but mention this in your report.

Make a triangular table, with rows indexed by d, the distance of each computed approximation (there are d columns in row d, one column for each root) to the multiple root one.

What is the relationship between the multiplicity d of the root one and the distance of the computed approximations to one?

For the same polynomials $p(x) = (x-1)^d$, $d = 2, 3, \ldots, 10$, run the method with a desired accuracy equal to zero and the maximum allowed steps equal to 100.

Describe what happens when Müller's method is allowed to run long enough on $p(x) = (x-1)^d$. *Can you explain your observations?*

In class we have seen how to restore the quadratic convergence of Newton's method when the multiplicity of the root is known.

Create a new script "new_mueller" which uses a modified version of Newton's method for roots of multiplicity m where m is given on input. (modify the script "newton" posted at the class web site for Lecture 8). Redo the experiments again on $p(x) = (x-1)^d$, *for* $d = 2, 3, \ldots, 10$, *and describe what happens. As a bonus, you may also modify "newton" so that, instead of given the multiplicity m on input, the "newton" function will estimate the multiplicity of the root and adjust itself accordingly.*

2.4 A perfidious polynomial. An easy looking test polynomial is $p(x) = (x-1)(x-2)(x-3)\cdots$. We can generate its coefficient vector using **poly(1:d)**, for any degree **d**.

Assignment Four. Run the method of Müller on this polynomial for increasing values of d, starting at $d = 3$. As in assignment three, your "ref_mueller" is preferred, but you may also use the given "run_mueller" if you failed to complete assignment two. Each time compute and record the maximal distance between the exact root and its corresponding approximation. Use `1.0e-8` as desired accuracy and allow a sufficiently large number of iterations. Stop at that degree d for which the method is no longer able to compute more than two significant decimal places of each root.

Explain why it is so hard to compute all roots of this polynomial.

3. Deadline is Friday 30 September 2005 at 3PM

Bring *your* solution to the project to class. The *your* is emphasized to stress that your solution is the result of an *individual* effort. Collaborations are **not** permitted.

Your solution should contain the following:

1. Answers to the questions in the assignments in complete grammatically correct sentences.
2. Tables summarizing the numerical experiments you have done. Use the **format short e** command when generating data, so your numbers in the tables will also be in scientific notation.
3. Sequences of commands used to generate the data for the experiments. It is good practice to store these commands in little .m files, e.g., in `assignment_one.m`, `assignment_two.m`, etc.
4. Output of your sessions with MATLAB or Octave as *an appendix*. Typing **diary** followed by the name of a file in a session creates a new file with the given name which will contain everything you see on the screen during the session. You may edit out mistakes from the output of diary.

The solution to the project is essentially a report on paper.

If you have questions or difficulties with the assignments, feel free to come to my office for help.