Guide of Implemented Programs

1 Newton_Method.c

This program uses the Newton's method to find the zeros of monic complex polynomials $p(z) = a_0 + a_1 z + \ldots + a_{d-1} z^{d-1} + z^d$.

• Right after the user creates the polynomial, the program computes the value

$$R = \max\{1, |a_0| + |a_1| + \ldots + |a_{d-1}|\}.$$

The search will be made inside the complex square of side R. Actually, all zeros of p are located in the disk of center 0 and radius R, but the Newton's method still might work outside this disk. Furthermore, is much more fast to generate points in the square than in the disk.

• In each step, the program draws a random point z in the square (with uniform distribution) and verifies if

$$|z - \zeta| > \frac{3 - \sqrt{7}}{2\gamma(p, \zeta)},$$

for all solutions ζ already computed. In the case this condition is true, the program applies the Newton iteration in z. Doing this we have a sequence z_i of approximations, with $z_0 = z$.

• The iteration continues until we obtain $|p(z_i)| < \varepsilon$ (the tolerance value $\varepsilon > 0$ is specified inside the code). In the case this condition is not satisfied before t steps (the value of t is also defined inside the code) or the iteration generates a value outside the square, then the iteration stops and the program uses the last value z_i computed. Furthermore, if during the iterations we have $p'(z_i) = 0$, then the iteration stops and the program uses the value z_{i-1}.

• Suppose z_i is the value computed by the Newton iteration. Then the program checks if $|p(z_i)| < \varepsilon$. In the case this is true, then the program verifies if

$$|z_i - \zeta| > \frac{5 - \sqrt{17}}{4 \max\{\gamma(p, z_i), \gamma(p, \zeta)\}},$$

for all solutions ζ already computed. If this condition also is true, then z_i is considered to be a new solution.

• After this, the program checks if there is already d solutions. In the case this is true, we are done.

• It may happen that the program takes a lot of time to compute all solutions. Then we impose that the procedure above is repeated a certain number of times (this quantity is also defined inside the code). At the moment we defined that the procedure is repeated $10000 \cdot \lceil R \rceil$ times. With this, the program generates $10000 \cdot \lceil R \rceil$ random points in the complex square to use the Newton iteration. This means we are using the Newton iteration in about 10000 points for each unit complex square.

• During the computation, the program shows the zeros already computed, with 15 digits of precision. At the end of the computation, the program show how many random points were used.

In general, the Newton's method is used to computed only one zero of the polynomial. After we have one zero, there are other better methods which can be used. The Newton's method is also bad when the polynomial has multiple zeros. Finally, we can note that the program described here gets much slower as the size of the coefficients increases. This is because the sample space increases together (note that the sample square increases quadratically as one of the coefficients increases linearly).

2 QR_method.c

If we want to compute just one zero of a polynomial p, Newton's method is a great method. But if we want to compute all the zeros, we must use another method. A good one is to consider the companion matrix of p and compute it's eigenvalues using algorithms of balancing followed by the QR algorithm (this is the method used by MATLAB with the command `roots`). This method is fast and is insensitive to multiple zeros, i.e., it computes all repeated zeros without any problem.

This program finds all zeros (with multiplicity) of a monic complex polynomial $p(z) = a_0 + a_1 z + \ldots + a_{d-1} z^{d-1} + z^d$.

• Right after the user creates the polynomial, the program uses the lapack driver routine `zgeevx`.

• After using this routine, the program shows the eigenvalues of the companion matrix of p. These eigenvalues are the zeros of p.

The approximations computed are very precise. The article [1] gives an explanation about this matter, so we won't talk about this in this text. This article also talks about the running time of this method. We will give some analysis of the running time of this method in the next section.

3 QR_random_samples.c

To analyse the running time of the QR algorithm, we will generate a lot of random polynomials and compute their zeros with the algorithm used in `QR_method.c`.

- This program generates 100 random polynomial such that for each coefficient a_i, we have that $\mathrm{Re}(a_i)$ and $\mathrm{Im}(a_i)$ are drawn from the standard normal distribution $N(0, 1)$. The degree of the polynomials are fixed inside the code.

- The zeros of these polynomials are computed with the previous algorithm used in `QR_method.c`.

- It's optional to show all the computed zeros.

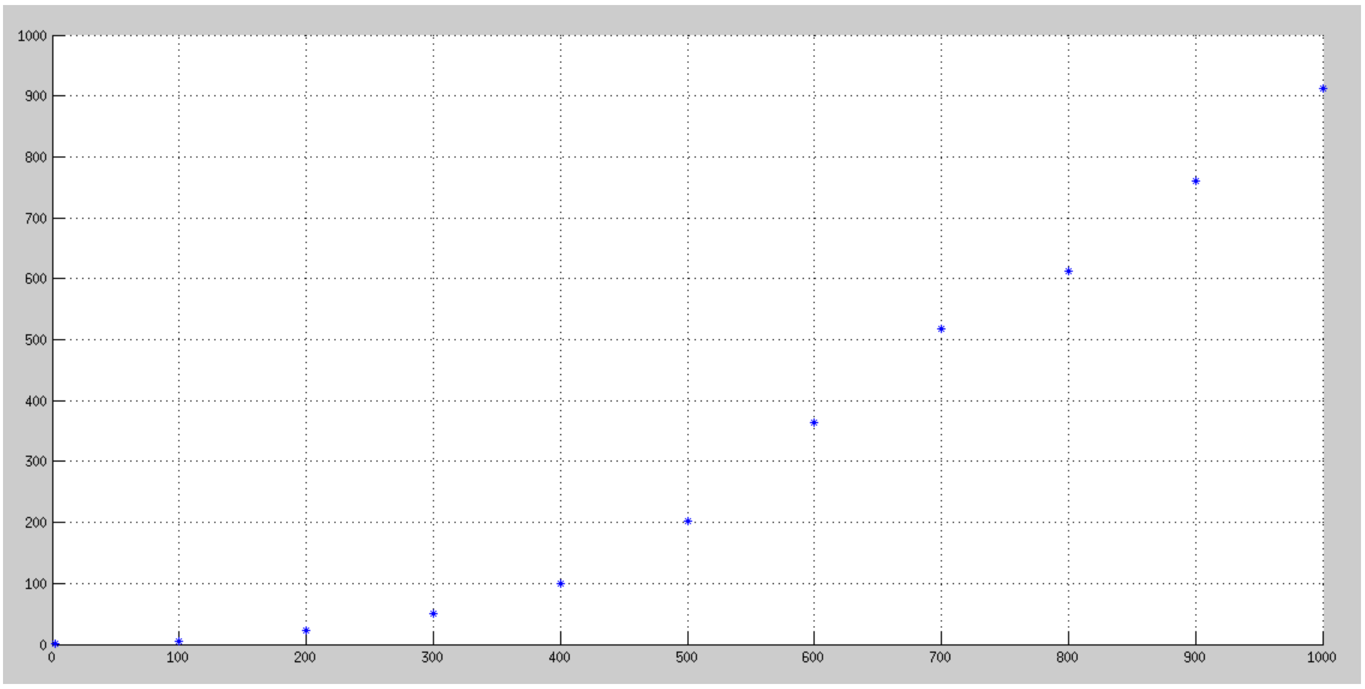

Figure 1: This figure shows the time used to compute the zeros of the 100 random polynomials, for $d = 2, 100, 200, 300, \ldots, 1000$. The horizontal axis represents the value of the degree d and the vertical axis represents the time to make the computations, in seconds.

This figure is plot of the following table.

d	Time in computation
2	0.034 s
100	4.018 s
200	23.650 s
300	50.823 s
400	100.041 s
500	202.513 s
600	364.154 s
700	518.512 s
800	611.745 s
900	760.565 s
1000	911.775 s

Denote by $N(0, I_d)$ the distribution described above for polynomials of fixed degree d and denote by $\text{Cost}(p)$ the time necessary to compute the zeros of p (in this context, note that $\text{Cost}(p)$ is a random variable in the space of the polynomials with degree d). Then the table above gives some estimates to the mean

$$\underset{p \in N(0,I_d)}{E}[\ \text{Cost}(p)\ ],$$

for each d.

As the article [1] indicates, we should expect to have $\underset{p \in N(0,I_d)}{E}[\ \text{Cost}(p)\ ] = O(d^3)$. The results here indicates that the constant in this asymptotic is very small, which is a good thing. Of course this is the case only from random polynomials with normal distribution. Others distributions might have other constants, and they may be bigger.

[1] POLYNOMIAL ROOTS FROM COMPANION MATRIX EIGENVALUES, ALAN EDELMAN AND H. MURAKAMI.