

NUMERICAL ALGORITHMS

Dr Derek O'Connor

Assignment No. 2 : Zero Finding Methods.

OUT: Wed 25 Oct 2006

IN : Wed 8 Nov 2006

1 PURPOSE

Although MATLAB has built-in functions for solving 1- and n -dimensional non-linear equations it is essential that you program some of the methods discussed in class so that you appreciate how difficult it is to write (and use) good software. This appreciation will, it is hoped, make you a critical user of other people's software.

2 EXERCISE PART 1. Zeros of General Functions

Write MATLAB Functions for :

1. Bisection Method.
2. Newton Method.
3. Secant Method.
4. ModFzero. A modified version of MATLAB's `fzero`. This is a function that is given the arguments specified below and calls MATLAB's `fzero` with the appropriate arguments.

2.1 Specifications for Part 1

1. Each methods should be written as an independent m.file function with the following names
bisect.m, newton.m, secant.m, modfzero.m
2. The calling conventions are as follows :
 - (a) [zero flag iters] = Bisect(X1, X2, Xtol, Maxits)
 - (b) [zero flag iters] = Newton(X1, Xtol, Maxits)
 - (c) [zero flag iters] = Secant(X1, X2, Xtol, Maxits)
 - (d) [zero flag iters] = ModFzero(X1, X2, Xtol, Maxits)

3. The flag conventions are as follows :
 - (a) flag = 0 : x -sequence convergence.
 - (b) flag = 1 : F -value is zero.
 - (c) flag = -1 : Maxits reached without X or F convergence.
 - (d) flag = -2 : Cannot proceed.
4. Your MATLAB programs for these methods should be able to run without error when called by the MATLAB test program `drvnle.m`. This function and the function definitions and their derivatives `fun.m`, `dfun.m` are in `Assign2.ZIP` on the class webpage. This includes all the files for this part of the assignment.

The test functions are as follows (in `FUN.m`) :

$$f_1(x) = x - \cos(x) \quad (1)$$

$$f_2(x) = e^{x+1.00202} - e^1 \quad (2)$$

$$f_3(x) = \frac{\log_2 x}{(x-1)} - \frac{1}{\ln 2} \quad (3)$$

$$f_4(x) = x^{20} - 1.0 \quad (4)$$

$$f_5(x) = -1.4 + x^2 + \frac{\log_{10}(|1 + 3(1-x)|)}{80} \quad (5)$$

$$f_6(x) = (y - 4.0) \times (y + 2.0) \times (y + 41.0) \quad (6)$$

$$\text{where } y = (x - 1.01 \times 10^{-9}) \times 10^8$$

$$f_7(x) = (x^2 - 2.2x + 1.21) \times \frac{x - 1.1}{|x - 1.1|} \quad (7)$$

$$f_8(x) = |x - 9.1|^{4.5} \quad (8)$$

$$f_9(x) = |x - 8.4317|^{0.4} \quad (9)$$

$$f_{10}(x) = \begin{cases} 0, & |x| < 3.8 \times 10^{-4} \\ x e^{-1/x^2}, & |x| \geq 3.8 \times 10^{-4} \end{cases} \quad (10)$$

$$f_{11}(x) = \begin{cases} e^x, & x > -1 \times 10^6 \\ e^{-1 \times 10^6} - (x + 1 \times 10^6)^2, & x \leq -1 \times 10^6 \end{cases} \quad (11)$$

$$f_{12}(x) = \tan^{-1}(x) \quad (12)$$

The derivatives of these functions are in the file `dfun.m`.

You need to unzip all the files in `Assign2.zip` and make them accessible from your current directory. These files are

1. `drvnle.m`
2. `getdat.m`
3. `fun.m`
4. `dfun.m`

See the Notes below.

```

function y = FUN(x);

% This is a function picker.
% For example,
% if FNO = 4 then this returns y=F4(x)
% Derek O'Connor, UCD, Nov 2005.

global FNO;

switch FNO
  case 1
    y = F1(x);
  case 2
    y = F2(x);
  case 3
    y = F3(x);
  case 4
    y = F4(x);
  case 5
    y = F5(x);
  case 6
    y = F6(x);
  case 7
    y = F7(x);
  case 8
    y = F8(x);
  case 9
    y = F9(x);
  case 10
    y = F10(x);
  case 11
    y = F11(x);
  case 12
    y = F12(x);
  otherwise
    error('Global var FNO must be in
    [1,2,...,12]')
end;

function y = F1(x);
  y = x-cos(x);
function y = F2(x);
  y = exp(x-1.00202)-exp(1);
function y = F3(x);
  y = log2(x)/(x-1) - 1/log(2);
function y = F4(x);
  y = x^20-1;
function y = F5(x);
  y = -1.4+x^2+log(abs(1+3.*(1-x)))/80;
function y = F6(x);
  z = (x-1.01*1.0e-9)*1.0e8;
  y = (z-4.0)*(z+2.0)*(z+41.0);
function y = F7(x);
  y = (x^2-2.2*x+1.21)*(x-1.1)/abs(x-1.1);
function y = F8(x);
  y = abs(x-9.1)^4.5;
function y = F9(x);
  y = abs(x-8.4317)^0.4;
function y = F10(x);
  if abs(x) < 3.8e-4
    y = 0.0;
  else
    y = x*exp(-1.0/x^2);
  end;
function y = F11(x);
  if x > -1.0e6
    y = exp(x);
  else
    y = exp(-1.0e6)- x+1.0+1.0e6)^2;
  end;
function y = F12(x);
  y = atan(x);

```

2.2 Notes

1. You must read the programs Assign2.zip and be sure that you understand what they do before using them. In particular, drvnlc.m is a driver program that tests each of the methods you have written (Bisect, etc.), by calling them with a set of test functions (in fun.m) and a set of starting points (in getdat.m).
2. All MATLAB statements FUN.m are in one .m-file and define the single function FUN(x). The actual functions (F1,F2, etc.) are *subfunctions* of FUN and are visible to (callable by) other functions in this .m-file only. There is a similar function picker for the derivatives of these functions in DFUN.m, except for functions 3, 5, and 12. You must write the derivative functions for these yourself.
3. *Good Programming Practice.* The quickest way to get the driver program working on the 4 methods you have to write is to write each as a *stub*. This is a skeleton function that has the correct function header and a single statement that returns 'null' information, e.g., [zero flag iters]=[0 0 0].

Once you have the driver and stubs working correctly, fill in the details of each method and test it, one-at-a-time, until all are working. Using stubs is a general idea that can be applied to any large program.

3 EXERCISE PART 2. Zeros of Polynomials.

Finding the zeros of a polynomial is computationally difficult when the it has high degree and especially when it has high multiplicity.

For example, Wilkinson's polynomial

$$P_{w30}(x) = (x-1)(x-2)\cdots(x-30),$$

has the exact and well-spaced zeros $z_k = 1, 2, \dots, 30$, but is very ill conditioned.

The polynomial

$$P_{100}(x) = (x-1)^{40}(x-2)^{30}(x-3)^{20}(x-4)^{10}$$

also has well-spaced zeros but has very high multiplicities

To see what happens in MATLAB type in

```
>>pw30 = poly(1:30);
>>zw30 = roots(pw30)
>>plot(zw30, '.k')

>>p100 = poly([ones(1,40) 2*ones(1,30) 3*ones(1,20) 4*ones(1,10)])
>>z100 = roots(p100)
>>plot(z100, '.k')
```

Because of these difficulties, general-purpose zero-finders have trouble with polynomials. For this reason MATLAB has the function `roots(p)` for finding the zeros of polynomials. Even this function gets into trouble when presented with an polynomial with high multiplicities, as can be seen with $p_{100}(x)$ above.

A new method by Z. Zeng (July 2003) claims to overcome these difficulties. We wish to test this claim, to a limited extent.

3.1 Specifications Part 2

1. Download and unzip the files `multroot.zip` and read the `readme.txt` file and test `multroot.m` as shown in this file.
2. Test and compare MATLAB's `roots(poly)` and Zeng's `multroot(poly)` on the following functions :

$$P_{w30}(x) = (x-1)(x-2)\cdots(x-30). \quad (13)$$

$$P_{mk}(x) = (x-1)^{4k}(x-2)^{3k}(x-3)^{2k}(x-4)^k \quad \text{for } k = 1, 2, 3, 4. \quad (14)$$

$$P_{m15}(x) = \left(x - \frac{10}{11}\right)^5 \left(x - \frac{20}{11}\right)^5 \left(x - \frac{30}{11}\right)^5 \quad (15)$$

Please note that the file `multroot.zip` has all the m-files and technical papers relating to this method.

4 SOLUTION — General Functions

```

%-----%
function [z,flag,itors] =
    Bisect(a,b,tol,Maxits,flag);
flag = inf;
z = inf;
itors = 0;
fa = FUN(a);
fb = FUN(b);
if sign(fa) == sign(fb)
    flag = -2
    return
end;
for iters = 1:Maxits
    m = a + 0.5*(b-a);
    fm = FUN(m);
    if (fm == 0.0)
        z = m;
        flag = 1;
        return
    end;
    if abs(a-b) <= tol+eps*abs(a)
        z = m;
        flag = 0;
        return
    end;
    if sign(fa) == sign(fm)
        a = m;
        fa = fm;
    else
        b = m;
        fb = fm;
    end;
end;
flag = -1; % run out of iterations

%-----%
function [zero,flag,itors] =
    ModFzero(x1,x2,tol,Maxits,flag);
flag = inf;
zero = inf;
itors = 0;
if sign(FUN(x1)) == sign(FUN(x2))
    flag = -2;
    return
end;
[zero, fval, exitflag,output]=
    FZERO(@FUN, [x1,x2]);
if exitflag > 0
    flag = fval == 0
end
itors = output.iterations;
if iters > Maxits
    flag = -1;
end
%-----%

%-----%
function [z,flag,itors] =
    Newton(xold,tol,Maxits,flag);
flag = inf;
z = inf;
itors = 0;
fold = FUN(xold);
for iters = 1:Maxits
    fpold = DFUN(xold);
    if fpold == 0.0
        flag = -1;
        return
    end;
    xnew = xold - fold/fpold;
    fnew = FUN(xnew);
    if (fnew == 0.0)
        z = xnew;
        flag = 1;
        return
    end;
    if abs(xnew- xold) <= tol+eps*abs(xnew)
        z = xnew;
        flag = 0;
        return
    end;
    xold = xnew;
    fold = fnew;
end;
flag = -1; % run out of iterations

%-----%
function [z,flag,itors] =
    Secant(x1,x2,tol,Maxits,flag);
flag = inf;
z = inf;
itors = 0;
f1 = FUN(x1);
f2 = FUN(x2);
for iters = 1:Maxits
    if f1 == f2
        flag = -1;
        return
    end;
    x3 = x2 - f2/((f2-f1)/(x2-x1));
    f3 = FUN(x3);
    if (f3 == 0.0)
        z = x3;
        flag = 1;
        return
    end;
    if abs(x3-x2) <= tol+eps*abs(x3)
        z = x3;
        flag = 0;
        return
    end;
    x1 = x2; f1 = f2;
    x2 = x3; f2 = f3;
end;
flag = -1; % run out of iterations
%-----%

```

4.1 Notes

There is nothing very fancy about these implementations of the standard zero-finding methods. They use the rules-of-thumb given in Chapter 4 of the notes.

1. Bisection and Modified Zero check for wrong signs.
2. Each method uses the same f - and x - convergence tests.
3. Newton needs to check for $f'(x) = 0$. Secant checks for $f(x_1) = f(x_2)$.
4. The *maxits* flag is set just after the end of the main loop. It does not make sense to set it anywhere else.
5. The structure of each implementation is roughly the same : an initialization part, f - convergence part, x -convergence part, *maxits* reached part.

4.2 Results

```
-----
Function No 1
-----
Method      zero      f(zero)      flag  iters  No Evals
Bisect  7.3908513321516067e-001  0.0000000000000000e+000  1    66    69
Newton  7.3908513321516067e-001  0.0000000000000000e+000  1     5    12
Secant  7.3908513321516067e-001  0.0000000000000000e+000  1     6     9
Matlab  7.3908513321516056e-001 -1.1102230246251565e-016  0     8    14
-----
```

```
-----
Function No 2
-----
Method      zero      f(zero)      flag  iters  No Evals
Bisect -2.0199999999998552e-003  0.0000000000000000e+000  1    54    57
Newton -2.0199999999998106e-003  0.0000000000000000e+000  1     9    20
Secant -2.0199999999999481e-003  0.0000000000000000e+000  1    14    17
Matlab -2.0199999999998657e-003  0.0000000000000000e+000  1     6    12
-----
```

```
-----
Function No 3
-----
Method      zero      f(zero)      flag  iters  No Evals
Bisect  1.0000000034006220e+000  1.5735113434445225e-008  0    54    57
Newton  2.0000000000000001e-001  1.4597150777202392e+000  0     1     4
Secant  9.999999999999634e-001  2.8865798640254070e-015  0    12    15
Matlab  1.0000000056189062e+000 -3.5165610423604221e-009  0    28    34
-----
```

```
-----
Function No 4
-----
Method      zero      f(zero)      flag  iters  No Evals
Bisect  1.0000000000000000e+000  0.0000000000000000e+000  1    53    56
Newton  1.0000000000000000e+000  0.0000000000000000e+000  1    13    28
Secant  1.0000000000000000e+000  0.0000000000000000e+000  1    25    28
Matlab  1.0000000000000000e+000  0.0000000000000000e+000  1    13    19
-----
```

```
-----
Function No 5
-----
Method      zero      f(zero)      flag  iters  No Evals
Bisect  1.1875774162579775e+000 -3.9898639947466563e-016  0    54    57
Newton  1.5000000000000000e+000  8.4133566024300077e-001  0     1     4
Secant  1.1875774162579777e+000  2.6020852139652106e-017  0     7    10
Matlab  1.1875774162579777e+000  2.6020852139652106e-017  0     9    15
-----
```

Function No 6

Method	zero	f(zero)	flag	iters	No Evals
Bisect	4.1010000000000001e-008	0.0000000000000000e+000	1	77	80
Newton	4.1010000000000001e-008	0.0000000000000000e+000	1	3	8
Secant	4.1010000000000001e-008	0.0000000000000000e+000	1	67	70
Matlab	4.1009999999982842e-008	-4.6330939085222309e-010	0	56	62

Function No 7

Method	zero	f(zero)	flag	iters	No Evals
Bisect	1.0999999940395355e+000	0.0000000000000000e+000	1	26	29
Newton	1.0999999882596938e+000	0.0000000000000000e+000	1	25	52
Secant	1.1000000198573399e+000	0.0000000000000000e+000	1	37	40
Matlab	1.0999999822043613e+000	0.0000000000000000e+000	1	70	76

Function No 8

Method	zero	f(zero)	flag	iters	No Evals
Bisect	Inf	Inf	-2	0	3
Newton	9.1000000000000068e+000	2.1486017101577638e-064	0	123	248
Secant	9.1000000000000139e+000	4.8617306858290170e-063	0	171	174
Matlab	Inf	Inf	Inf	0	3

Function No 9

Method	zero	f(zero)	flag	iters	No Evals
Bisect	Inf	Inf	-2	0	3
Newton	Inf	Inf	-1	300	602
Secant	Inf	Inf	-1	300	303
Matlab	Inf	Inf	Inf	0	3

Function No 10

Method	zero	f(zero)	flag	iters	No Evals
Bisect	1.5625000000000000e-002	0.0000000000000000e+000	1	6	9
Newton	Inf	Inf	-1	300	602
Secant	Inf	Inf	-1	300	303
Matlab	1.5419318344364441e-002	0.0000000000000000e+000	1	16	22

Function No 11

Method	zero	f(zero)	flag	iters	No Evals
Bisect	-7.1778349147983768e+005	0.0000000000000000e+000	1	180	183
Newton	-1.0000010000000001e+006	-1.3552527156068805e-020	0	52	106
Secant	-1.0000010000000003e+006	-1.2197274440461925e-019	0	70	73
Matlab	-5.0060069415714656e+005	0.0000000000000000e+000	1	3	9

Function No 12

Method	zero	f(zero)	flag	iters	No Evals
Bisect	-1.2272733663244316e-091	-1.2272733663244316e-091	-1	300	303
Newton	0.0000000000000000e+000	0.0000000000000000e+000	1	39	80
Secant	0.0000000000000000e+000	0.0000000000000000e+000	1	46	49
Matlab	1.0804238957906978e-024	1.0804238957906978e-024	0	9	15

4.3 Comments

1. Most of the functions were designed to be nasty in order to test various aspects of the methods under test. Bisect and ModFZero are directly comparable because ModFZero uses the bisection method when it senses that the function is in a difficult region. The results show that while both are fairly robust, MATLAB's FZero is much more efficient, taking fewer iterations than Bisect in general. Note that I increased maxits = 300. My Bisect would not converge in 200 iterations for $f_{12}(x) = \tan^{-1}(x)$, despite the fact that it acts like $f(x) = x$ around 0. This shows that my convergence test is not as robust as I thought it was.
2. Notice that most methods were stopped by the f -convergence test, i.e, flag = 1.
3. The starting values for all methods can have a great effect on the convergence behavior. One example should be enough to make this point. Function 6 is shown in Figure 1.

$$f_6(x) = (y - 4.0) \times (y + 2.0) \times (y + 41.0) \quad \text{where } y = (x - 1.01 \times 10^{-9}) \times 10^8$$

We can see that Newton-like methods cannot reach the middle zero unless it starts somewhere between (x_{\max}, x_{\min}) , the values where this function's derivative $f'(x) = 0$.

Note how closely-spaced the zeros are. This function was originally designed to test single-precision zero-finders. What changes should be made for double-precision testing?

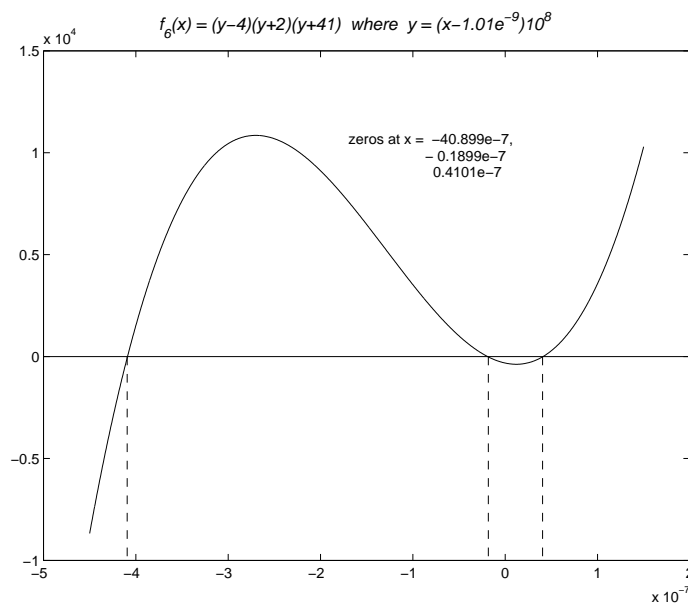


Figure 1: Closely-spaced Zeros

Question 1. Under what starting conditions would the Bisect method reach the middle zero of $f_6(x)$? In general, is it necessary to start Bisect with $a < b$? If the answer is 'yes', how does the Bisect method maintain this condition?

5 SOLUTION – Polynomials

The functions used to test and compare MATLAB's `roots(p)` and Zeng's `multroot(p,tol)` were chosen to have well-spaced zeros.

$$P_{w30}(x) = (x - 1)(x - 2) \cdots (x - 30) \text{ has zeros at } 1, 2, \dots, 30$$

$$P_{mk}(x) = (x - 1)^{4k}(x - 2)^{3k}(x - 3)^{2k}(x - 4)^k \text{ for } k = 1, 2, 3, 4 \text{ has zeros at } 1, 2, 3, 4.$$

$$P_{m15}(x) = \left(x - \frac{10}{11}\right)^5 \left(x - \frac{20}{11}\right)^5 \left(x - \frac{30}{11}\right)^5 \text{ has zeros at } \frac{10}{11}, \frac{20}{11}, \frac{30}{11}.$$

The MATLAB code for these is in `... \multroot \testsuit` folder of the unzipped `multroot.zip`.

For testing purposes we need to know the zeros of the polynomials because we have no idea what junk the polynomial root-finders may produce.

Zeng's `multroot(p,tol)` would not work properly on $P_{w30}(x)$ until I set `tol := 10-4` which is much higher than the default of `10-10`. MATLAB's error messages were vaguely helpful, indicating that the error was occurring in a function that was called by `multroot` (which is just a shell for calling the functions that really do the work). Looking at the code for these functions showed that it would take 4 months to understand it, so I guessed that a less stringent tolerance might help. It did. If the tolerance had been 'hardwired' into the code rather than allowing the user to set it then I would never have got it to work. This is an important rule of good numerical software writing : critical parameters should be under the control of the user.¹

5.1 Wilkinson's Polynomial

Figure 2 shows plots of the zeros produced by MATLAB's `roots(p)` and Zeng's `multroot(p,tol)` for the polynomial $P_{w30}(x)$. The circles are the exact zeros. This is a better way to compare complex-valued numbers than putting them in a table. As we can see, both methods produce identical results — garbage. Zeng's method automatically switches to MATLAB's method when the polynomial has simple zeros. The results for $P_{w20}(x)$ are plotted in blue. These are correct.

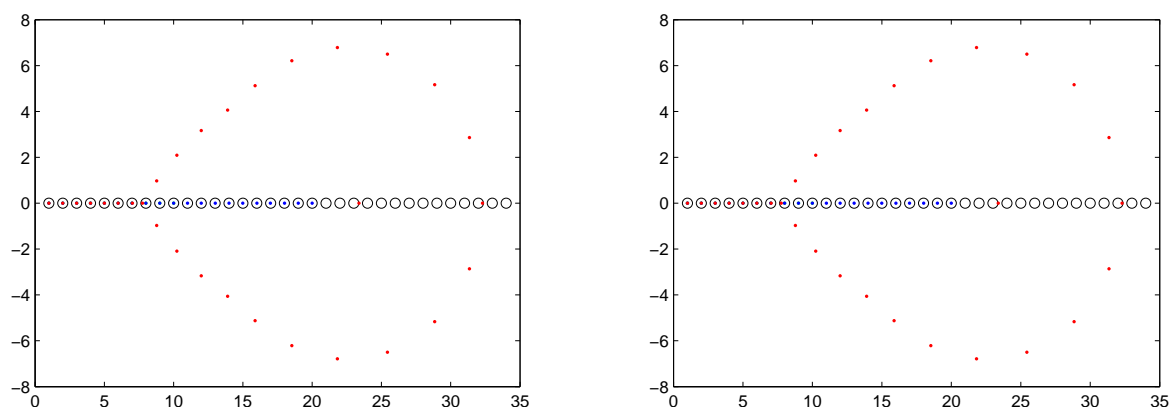


Figure 2: MATLAB and Zeng on Wilkinson's Polynomial $P_{w30}(x)$

¹This may be good for expert users but it is bad to ask an amateur to set parameters.

5.2 4 Multiple Zeros of $P_{mk}(x)$

The results of these test are shown in Figure 3. MATLAB's roots(p) performs badly but Zeng's mult-root(p) is perfect. Zeng's method also gives the multiplicities of the zeros it has found — not an easy task.

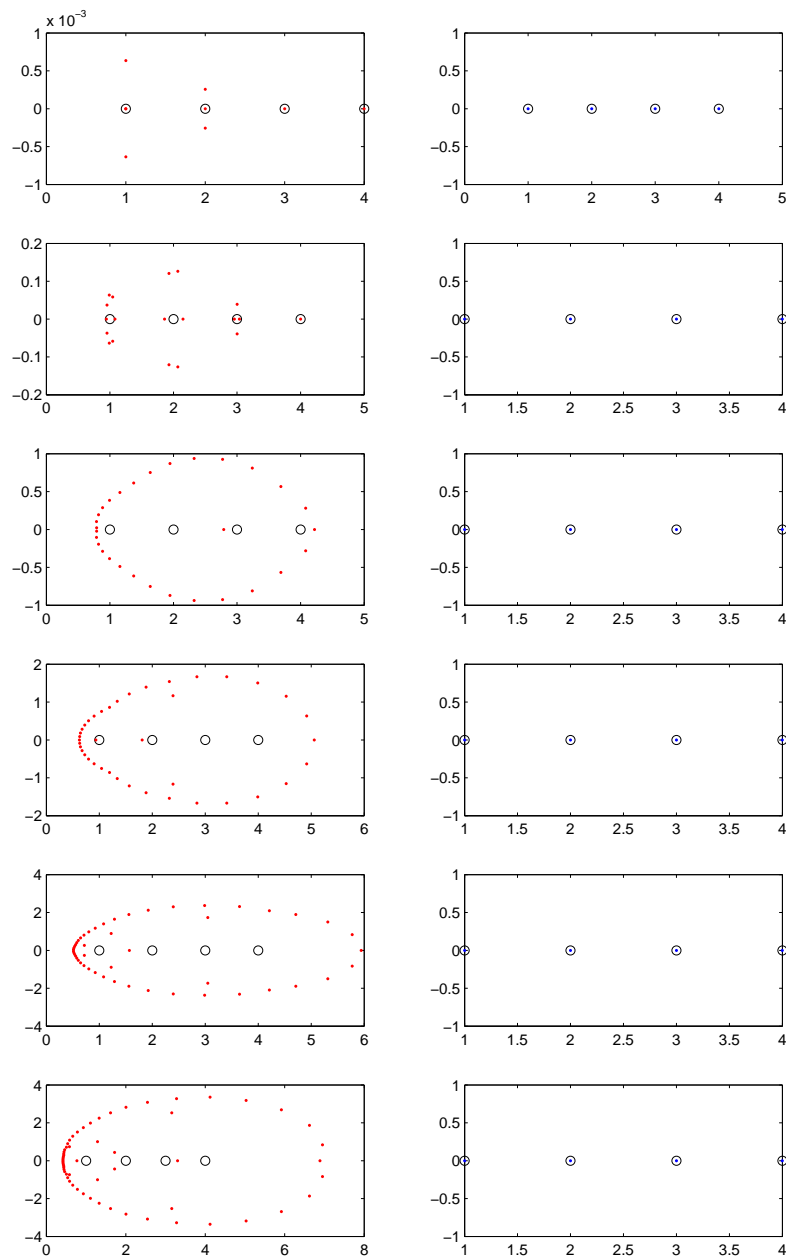


Figure 3: MATLAB and Zeng on 4 Multiple Zeros of $P_{mk}(x)$

5.3 3 Multiple Zeros of $P_{m15}(x)$

Figure 4 shows that Zeng’s method gives perfect results for this function, giving both the correct values of the zeros and their multiplicities. MATLAB gets only 3 out of the 15 zeros correct. These are the red dots in or near the circles. Notice however that MATLAB gets the real parts of the zeros close to the exact values. The spurious imaginary parts contaminate the results.

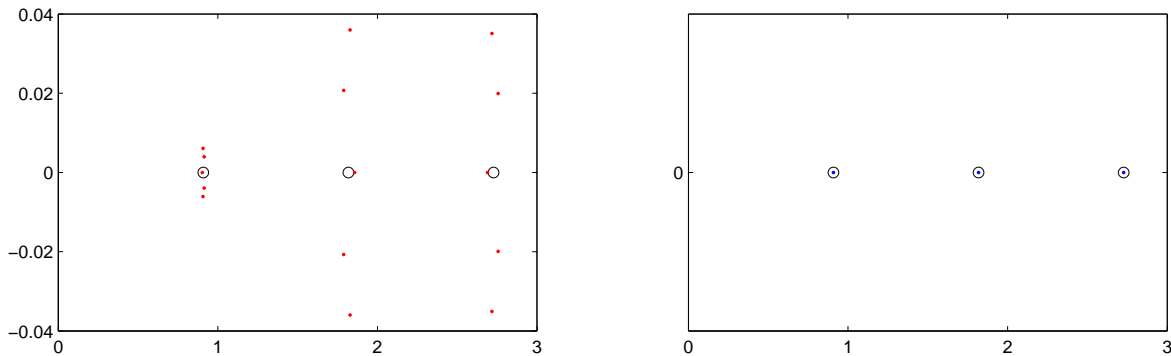


Figure 4: MATLAB and Zeng on 3 Zeros of Multiplicity 5 of $P_{m15}(x)$

5.4 Comments

Finding the zeros of polynomials can be very difficult. High degree polynomials with simple zeros can be very ill-conditions, as Wilkinson’s polynomials show. Polynomials with zeros of high multiplicity are even more difficult. Zeng’s method works as follows :²

Given a polynomial $p_n(x) = a_0 + a_1x^1 + a_2x^2 + \dots + a_nx^n$, multroot(p) calls 2 functions

- GcdRoot(p) decomposes $p_n(x)$ into $a_0(x - x_1)^{m_1}(x - x_2)^{m_2} \dots (x - x_k)^{m_k}$, returns m_1, m_2, \dots, m_k along with initial approximations to the zeros x_1, x_2, \dots, x_k .
- PejRoot uses this information to refine the zeros to an optimal accuracy, which depends on the *pejorative condition number* of $p_n(x)$.

This work won the Distinguished Paper Award at ACM ISSAC ’03

²This description is from zrootpack.pdf