The "mroots.sci" program : multiple roots of a real polynomial.

1 Principle, initialization of the algorithm

I tried with this program to determine the multiplicities of the roots of a real polynomial $P(x)$ of degree n, and to initialize the values of its roots using the `roots.sci` and `bezout.sci` programs offered by Scilab, when the polynomial has multiple roots.

To find out if a polynomial has multiple roots I use the Scilab `bezout.sci` function : if the gcd of $P(x)$ and $\frac{dP(x)}{dx}$ is 1, then the roots are simple and then the program `roots.sci` will give them to me.

Prerequisites :

1. In the proposed program, I want to use Scilab functions already programmed and preferably compiled functions (in FORTRAN, C, or C ++) for speed of execution.

2. **Results of the theory**.
   We know the following result : when a polynomial $P(x)$ has multiple roots, if x_k is one of its roots, then x_k is also a root of $\frac{dP(x)}{dx}$ and possibly other derivatives. By taking a monic polynomial $P(x) = \prod_{k=1}^{k=K}(x-x_k)^{m_k}$ which has multiple roots (and/or simple ones), then the gcd of $\frac{dP(x)}{dx}$ and $P(x)$ is :

$$g(x) = gcd(P(x), \frac{dP(x)}{dx}) = \prod_{k=1}^{k=K}(x-x_k)^{m_k-1}$$

$$U(x) = \frac{P(x)}{g(x)} = \prod_{k=1}^{k=K}(x-x_k) \; and \; V(x) = \frac{\frac{dP(x)}{dx}}{g(x)}$$

3. $U(x)$ is the decomposition without square (see the algorithm of Yun or Tobey-Horowitz) of $P(x)$, the roots of $U(x)$ are then simple roots.

4. Now let us form the ratio $R(x) = \frac{V(x)}{U(x)}$ this ratio is : $R(x) = \sum_{k=1}^{k=K}\frac{m_k}{x-x_k}$ .The roots x_k are the simple roots of $U(x)$ and therefore the desired roots and the integers (the multiplicities) are worth $m_k = ((x-x_k)\frac{V(x)}{U(x)})_{x=x_k}$ or $m_k = \frac{V(x_k)}{(\frac{dU}{dx})_{x=x_k}}$. This result is classic and constitutes an exercise often asked in the undergraduate math curriculum. $R(x) = \frac{V(x)}{U(x)} = \sum_{k=1}^{k=K}\frac{m_k}{x-x_k}$ is also the breakdown into simple elements of the report $\frac{dP(x)}{dx}$.

By analyzing the Bezout equation given in Scilab, `[g,U]=bezout(P,dP)` where `P` is the starting unit polynomial and `dP` its first derivative, we can easily see, that the term `U(2,2) = - P/g = - Ux` and `U(1,2) = dP/g = Vx,` moreover `det(U) = -1`. So finding the simple and/or multiple roots of $P(x)$ is to find the simple roots of `U(2,2)`, and the multiplicities are obtained by a calculation on `U(1,2)` via $R(x) = \frac{V(x)}{U(x)} = \sum_{k=1}^{k=K} \frac{m_k}{x-x_k}$. This is so from these considerations I propose initializing the program Scilab `mroots.sci`.
Note : We can also find the multiplicities from $R(x)$: if we put $R(x)$ in the form of partial fraction decomposition using the program Scilab `pfss(Vx/Ux)` we find the coefficients m_k in the numerators of rational components.

By brutally applying the proposed theory, one can have, for some polynomials, problems : if $P(x)$ has only simple roots then the program `[g, U] = bezout (P, dP)` must theoretically give $g == 1$ but it is not always the case, especially on the example offered by Wilkinson, polynomial of degree 20, (see help on the `roots.sci` program), we got my first version of the `mroots` program, which brutally applied the theory, a multiple root of multiplicity 20, located at the isobarycenter of the roots, (an example analog proposed by Samuel Gougeon gave this result with Scilab-Windows but not with Scilab-Linux).
To avoid this problem I propose a test on the veracity of the roots and their respective multiplicities, by calculating the sum and the product of the roots and by checking, with a given precision, that this sum and this product are respectively the second and last coefficient of the polynomial.
Notes :
As I just said if the term $g(x)$ given by the Bezout equation is worth 1 then $P(x)$ has only simple roots and the `mroots` program uses the `roots.sci` program only.

Similarly in the proposed program, I first determine the zero roots from the coefficients of the polynomial so as not to "pollute" the programs `bezout.sci and roots(U (2,2))` by these null roots and decrease the degree of the polynomial of departure. The rest of the program is the presentation of the solution.

Of course we use the `roots.sci` program to calculate the simple roots of $U(x)$: this is where accuracy issues can arise in addition to previous remarks. Let us take an example where the polynomial $P(x)$ is made up of the factors : $s^3, (s-2)^3, (s+1), (s^2+s+1)^4, (s+2)^4, (s+6), (s^2+1.5s+1)^3, (s^2+0.9s+0.5)^4, (s+2)^5$ the degree of this polynomial is 44 and using the initialization of my program, we find the multiplicities well and the roots (reals and complex conjugate roots) are calculated with an error of order $1.e-4$. With the Scilab program `nearly_multiples.sci` with an accuracy of 0.1 one finds the real roots quite well but the multiplicities are equal to 24, as for the imaginary roots (not pure) they are all separate.

2 improvement of the solution : search for roots, given multiplicity, by least squares method

In many cases, the initial solution obtained seems good and gives results significantly higher than those obtained by the `roots.sci` program alone (in the case of multiple roots). Similarly, the multiplicity is correct and gives for many examples tested, a result clearly more reliable than programs which seek to homogenize the roots in groups ("cluster"), and to average over each of these groups : `nearly_multiples.sci` program, for example from Scilab, that we can test on the previous example.

Let monic polynomial $P(x) = \prod_{k=1}^{k=K}(x - x_k)^{m_k}$ (degree n) which has K roots x_k each of multiplicity multiplicity m_k. At this stage, we therefore have the polynomial of degree n,$P(x)$, the vector of multiplicities $mu = [m_1, ..., m_k, ..., m_K]^T$ such that $\sum_1^K m_k = n$, which we will consider as correct and the vector of the corresponding roots $Z = [z_1, ..., z_k, ..., z_K]^T$ first initialization of the root vector. With these roots (vector Z) and this multiplicity (vector mu), we construct the approximate polynomial $P^*(x) = \prod_{k=1}^{k=K}(x - z_k)^{m_k}$. If we calculate this expression we recover the coefficients of the approximate polynomial which are also the elementary symmetric functions of the roots found.

Now the problem is a optimisation one, from the expressions of the elementary symmetric functions calculated from roots, we have to solve the mathematical problem : find the K values of the roots z_k minimizing the quadratic difference between the real elementary symmetric functions (given by the coefficients of $P(x)$) and those calculated from the roots and this for a given multiplicity vector.

One will find in the literature, a publication which justifies in a theoretical way and implements this principle with Matlab software (Author Z. ZENG, "COMPUTING MULTIPLE ROOTS OF INEXACT POLYNOMIALS ", Review "Mathematics of computation" Volume 74, July 22, 2004); so I'm not exposing root optimization by the use of elementary symmetric functions, (given by the coefficients of $P(x)$) and those calculated from the roots and this for a given multiplicity vector.

The program I propose uses the following programmed Scilab functions : `bezout.sci`, `roots.sci` and least squares optimization and does not ask to make specific macros to initialize the problem.

Notes : When we program the optimization problem, we build a difference vector DcW between the elementary symmetric functions calculated at each iteration and the "true" vector which is the vector of the coefficients of the studied polynomial (except for the sign). This vector difference is of dimension n, the degree of the starting polynomial, and depends on the K roots, with $K < n$: it is a transformation of $C_K \rightarrow C_n$. Then we calculate the jacobian matrix $JacW$ of DcW which is a rectangle matrix (n, K) (because the starting polynomial has multiple roots). See https: `//en.wikipedia.org/wiki/jacobian_matrix_and_determinant`.

On this subject, one will find on the internet exercises proposed to the French aggregation of maths, dealing with elementary symmetric functions and the calculation of the Jacobian matrix of these functions : these examples and the previously cited article were the source of my reasoning, because as I mentioned at the beginning of

my presentation, I am looking at all costs to use preprogrammed Scilab functions.

The syntax of this function is :

```
[rm, bkerr, pjcnd, fkerr] = mroots (P [, tol [, iter [, flag]]])
```

2.1 Example

Consider a polynomial of degree 5 having a triple root at -1 and two complex conjugate roots $-\frac{1}{2} \pm \frac{\sqrt{3}}{2}i$. This polynomial is written : $p(s) = 1 + 4s + 7s^2 + 7s^3 + 4s^4 + s^5$.

Let's create the programs : `r=roots(p)` then `rinit=mroots(p,"y")` to obtain on the one hand the roots of this polynomial by Scilab, and on the other hand the roots and the initial multiplicities by the program `mroots( p,"y")`, flag "y" in the `mroots` instruction : we do not do any optimization.

```
--> format(20)
--> r=roots(p)
r =
-0.49999999999999978 + 0.86602540378443393i
-0.49999999999999978 - 0.86602540378443393i
-1.00000505953435703 + 0.00000876385524891i
-1.00000505953435703 - 0.00000876385524891i
-0.99998988093129559 + 0.i
```

We have four complex roots and one real one. Now with the program `mroots(p,"y")` we obtain :

```
--> rinit=mroots(p,"y")//The program mroots whithout optimisation.
rinit =
-0.999999999999995 + 0.i                          3. + 0.i
-0.49999999999999989 + 0.86602540378443682i       1. + 0.i
-0.49999999999999989 - 0.86602540378443682i       1. + 0.i
```

Now we use `mroots.sci` with optimisation.

```
--> [rfinal,bkerr,pjcnd,fkerr]=mroots(p,1.e-14,3)
//Three iterations only.
rfinal =
-1.                  + 0.i                          3. + 0.i
-0.50000000000000011 + 0.86602540378443849i         1. + 0.i
-0.50000000000000011 - 0.86602540378443849i         1. + 0.i
bkerr =
0.
pjcnd =
4.13072440814959396
fkerr =
0.
```

2.2 More complex problem

Let's take the example close to the polynomial from section 1 :

The polynomial P has degree 34 and is composed of the elementary terms $s^3, (s-2)^3, (s+1), (s^2+s+1)^4, (s+2)^4$, $(s+6)$, $(s^2+1.5s+1)^3$, $(s^2+0.9s+0.5)^4$. To find the roots of P we put the polynomial in the form of its coefficients : this (inexact) result polynomial is P1 close to P, then program :

```
--> P1=s^3*(s-2)^3*(s+1)*(s*s+s+1)^4*(s+6)*(s+2)^4*..
( s*s+1.5*s+1)^3*(s*s+0.9*s+0.5)^4 ;
--> RAc=roots(P1)
RAc =
-6. + 0.i
2.0000032 + 0.0000056i
2.0000032 - 0.0000056i
.......
```

It's pretty much anything!!

```
--> format(16)
--> RACI=mroots(P1,"y") //Initialisation without optimisation.
RACI =
0. + 0.i                              3. + 0.i
-5.9999999999998 + 0.i                 1. + 0.i
2.              + 0.i                 3. + 0.i
-2.0000000041839 + 0.i                 4. + 0.i
-1.0000229796092 + 0.i                 1. + 0.i
-0.4999999770785 + 0.8660246420269i    4. + 0.i
-0.4999999770785 - 0.8660246420269i    4. + 0.i
-0.7499976077684 + 0.6614453600487i    3. + 0.i
-0.7499976077684 - 0.6614453600487i    3. + 0.i
-0.4500016559398 + 0.5454416934457i    4. + 0.i
-0.4500016559398 - 0.5454416934457i    4. + 0.i

--> [RACFINAL, bkerr, pjcdn, fkerr] = mroots(P1)//With optimisation
RACFINAL =
0. + 0.i                              3. + 0.i
-6. + 0.i                              1. + 0.i
2. + 0.i                              3. + 0.i
-2. + 0.i                              4. + 0.i
-1.0000000000007 + 0.i                 1. + 0.i
-0.5 + 0.8660254037844i                4. + 0.i
-0.5 - 0.8660254037844i                4. + 0.i
-0.7499999999998 + 0.6614378277661i    3. + 0.i
-0.7499999999998 - 0.6614378277661i    3. + 0.i
-0.45 + 0.5454356057318i               4. + 0.i
-0.45 - 0.5454356057318i               4. + 0.i

bkerr =
```

```
4.436358977D-14
pjcdn =
4616.6313404856
fkerr =
0.0000000004096
```

2.3 Bibliography :

1. en.wikipedia.org/wiki/jacobian_matrix_and_determinant.
2. Z. Zeng, Computing multiple roots of inexact polynomials, Revue « Mathematics of computation » july 2004.
3. Madina Hasan : The computation of multiple roots of a polynomial using structure preserving matrix methods. PhD thesis Sheffield University England July 2011.
4. S.Gratton, A.S.Lawless, N.K.Nichols, Approximative Gauss-Newton methods for non linear least squares problems. Numerical Analysis report 9/04 .