Numerical Computation of Polynomial Roots Using MPSolve Version 2.2

Dario Andrea Bini and Giuseppe Fiorentino
Dipartimento di Matematica, Università di Pisa
Via Buonarroti 2, 56127 Pisa.
(bini@dm.unipi.it, fiorent@di.unipi.it)

January 2000

Abstract

We describe the implementation and the use of the package MPSolve (Multiprecision Polynomial Solver) for the approximation of the roots of a univariate polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$.

The package relies on an algorithm, based on simultaneous approximation techniques, that generates a sequence $\mathcal{A}_1 \supset \mathcal{A}_2 \supset \ldots \supset \mathcal{A}_k$ of nested sets containing the root neighborhood of $p(x)$ (defined by the precision of the coefficients) and outputs Newton-isolated approximations of the roots. The algorithm, particularly suited to deal with sparse polynomials or polynomials defined by straight line programs, adaptively adjusts the working precision to the conditioning of each root. In this way, only the leading digits of the input coefficients sufficient to recover the requested information on the roots are actually involved in the computation. This feature makes our algorithm particularly suited to deal with polynomials arising from certain symbolic computations where the coefficients are typically integers with hundreds or thousands digits.

The present release of the software may perform different computations such as counting, isolating or approximating (to any number of digits) the roots belonging to a given subset of the complex plane. Automatic detection of multiplicities and/or of real/imaginary roots can be also performed. Polynomials with approximately known coefficients are also allowed.

1 Introduction

MPSolve is a software tool for the numerical computation of the roots of a univariate polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$. The executable program of the package is called `unisolve` (univariate polynomial solver).

In the present release the following features are available.

Input:

- The input polynomial can be provided either by means of the degree n and the coefficients a_i, $i = 0, \dots, n$ of the univariate polynomial $p(x)$, or as a C program for the evaluation of the values of $p(x)$, of $p'(x)$ and of an upper bound of $|p(x) - \mathrm{fl}(p(x))|$, where $\mathrm{fl}(a)$ denotes the value of the expression a computed in floating point arithmetic.
- The user can define both the input precision d_{in} and the desired output precision d_{out} expressed as decimal digits. The input precision defines the *polynomial neighborhood* of $p(x)$, i.e., the set of all the polynomials with coefficients having d_{in} common digits with the corresponding coefficients of $p(x)$, and the *root neighborhood*, i.e., the set of the roots of all the polynomials in the polynomial neighborhood of $p(x)$. For polynomials having coefficients with infinite precision (say, integer or rational) d_{in} must be set to zero.

Output:
In order to provide the user as much information as possible, MPsolve outputs its approximations according to the following rules:

- For infinite precision input, i.e., for $d_{in} = 0$, the program delivers a list of n complex numbers represented with at most d_{out} digits. For each number of this list, the displayed digits coincide with the digits of the corresponding root of $p(x)$. The number of displayed digits reaches (or even may slightly exceed) the maximum value d_{out} only for those sets of roots that have at least d_{out} common digits. Otherwise it is displayed a number of digits sufficient to provide Newton-isolated approximations of the roots. That is, the number of digits is large enough to separate each root from the others and to guarantee the quadratic convergence of Newton's iteration right from the start when applied to the output approximation. For instance, for the polynomial having the following 5 roots 1.11111111, 1.12222222, 1.1233333, 1/3, 1/3, and for the input $d_{in} = 0$, $d_{out} = 30$ the program would output the numbers (1.111, 0.0) (1.1222, 0.0) (1.12333, 0.0) (0.333333333333333333333333333333, 0.0) (0.333333333333333333333333333333, 0.0).
- For input with a finite precision ($d_{in} > 0$) the program delivers a list of n complex numbers having at most d_{out} digits as in the case of infinite precision. For each number z in this list there exists a polynomial $q(x)$ in the neighborhood of $p(x)$ such that $q(z) = 0$. Each root of $p(x)$ has its corresponding approximation in the list. Moreover, any polynomial $q(x)$ in the neighborhood of $p(x)$ has a root that coincides with z in the leading displayed digits.
- If the output format -Of is chosen, the program delivers the numerical approximations to the roots with no filtering of the correct digits,

a guaranteed a posteriori error bound, a string of three characters describing the status of the approximation. This string has the following meaning: the first character may take the following values:

  – m: multiple root
  – i: isolated root
  – a: approximated single root (relative error less than $10^{-d_{out}}$)
  – o: approximated cluster of roots (relative error less than $10^{-d_{out}}$)
  – c: cluster of roots not yet approximated (relative error greater than $10^{-d_{out}}$)

  The second character may take the following values

  – R: real root
  – r: non-real root
  – I: imaginary root
  – i: non-imaginary root
  – w: uncertain real/imaginary root
  – z: non-real and non-imaginary root

  The third character may take the following values

  – i: root in $\mathcal{S}$
  – o: root out of $\mathcal{S}$
  – u: root uncertain

- In certain cases it may happen that the relative error bound of the computed approximations is greater than 1, i.e., there is no correct bit in the output. This situation typically occurs, when the imaginary (or real) part of a nonzero root is zero or is such that its ratio with the real (imaginary) part has modulus less than $10^{-d_{out}}$. In this case the program outputs 0.0exx, where xx is the exponent of the approximation of this number. Therefore, an output like (1.234, 0.0e-1000) means that the imaginary part, if nonzero, has a modulus less than 10^{-1000}. This lack of information is typical in numerical computation when we have to deal with 0. Moreover, for polynomials having coefficients with infinite precision, it is easy to overcome this lack of information by using the reality or integrality of the coefficients. This is performed automatically with the option `-Dx`, where '`x`' may be `r` (real detect) `i` (imaginary detect) or `b` (both real and imaginary detect). See the next section about the available options.

  The uncertainty about the zero output is present also in the case of polynomials $p(x)$ having approximate input coefficients where the root neighborhood of $p(x)$ intersects zero. In this case the program outputs 0.0 for those approximations belonging to the connected component of the root neighborhood that intersects zero. However, these (nonzero) approximations are reported integrally in the standard error if the switch `-d` (debug) is set or the output format `-Of` is chosen (see the next section for more details).

Goals:

The program has been designed to deal with different goals. In particular besides the goal "Newton isolation" described above (default option `-Gi`), the program allows the further goals "approximation" (option `-Ga`) and "count roots" (option `-Gc`). Moreover, by giving some command line options, the user is enabled to restrict the search of the roots to specific subsets of the complex plane (option `-S`x), say, unit disk, right half-plane, etc.), also the automatic detection of multiple roots (option `-M`) and/or of real and imaginary roots (option `-D`) is fully implemented.

The program, based on the technique of simultaneous approximation of the roots, makes use of a variant of the algorithms described in [1], [2]. Essentially, the program constructs a sequence of sets $\mathcal{A}_1 \supset \mathcal{A}_2 \supset \mathcal{A}_3 \supset \ldots \supset \mathcal{A}_k$, where each set $\mathcal{A}_i$ is the union of n disks, contains the root neighborhood of $p(x)$, and each disk contains at least one root. Moreover, each connected component of $\mathcal{A}_i$ contains as many roots of any polynomial in the neighborhood of $p(x)$ as the number of disks that make up the component. Each set $\mathcal{A}_i$ is obtained by performing the computation with increasing levels of working precision w_1, $w_2, \ldots$, starting with the standard machine precision ($w_1 = 53$ bits for the IEEE standard) and doubling the precision at each stage, i.e. $w_{i+1} = 2w_i$, until either all the disks are Newton–isolated or the output precision has been reached, or the working precision becomes larger than the input precision, i.e., $w_i > 4nd_{in} \log_2 10$.

The main engine for shrinking the disks is Aberth's iteration. In order to accelerate the shrinking speed, a suitable strategy for the detection and analysis of clusters (connected components of $\mathcal{A}_i$) is used together with the application of the restarting criterion of [1] for choosing new centers for the disks in the clusters by means of the computation of a convex hull.

2 Outline of the algorithm

The program tries to perform the most part of the computation in the lowest working precision starting with the standard machine precision and switching to multiple precision *only if needed.* In this way, for polynomials having only few ill conditioned roots the (more expensive) mp version of the algorithm is applied just for the ill conditioned roots with a substantial saving of the computation time. This is made possible since our algorithm does use the "clean" information given by the coefficients of $p(x)$ at each step of the computation, in fact, *it does not compute the explicit factorization* of the polynomial. Algorithms based on explicit factorizations need a high precision computation of the coefficients of the factors obtained just at the first splitting stage even though the factors have only few ill conditioned roots.

This feature of using at each stage the information of the coefficients of $p(x)$ is particularly useful for sparse polynomials where the computation of $p(x)$ becomes very cheap. In fact, we have implemented a sparse version of the Ruffini-Horner rule for the computation of $p(x)$ and of $p'(x)$. The higher speed of this sparse version can be exploited at each level of our algorithm for polynomial roots.

Moreover, the program allows also to use polynomials defined by straight line programs. For instance the polynomial p_k of degree $n = 2^k - 1$ defined as

$$\begin{array}{l} p_0(x) = 1 \\ p_i(x) = x(p_{i-1}(x))^2 + 1, \quad i = 1, 2, \ldots, k, \end{array} \tag{1}$$

having roots strictly related to the Mandelbrot set, can be computed with just $3k$ arithmetic operations and the computation is more accurate.

Another feature of our algorithm is that it allows to use only those digits of the input coefficients that are strictly needed for the requested approximation. More specifically, it is typical in symbolic computation to encounter polynomials with integer coefficients having hundreds or even thousands digits. Very often, in order to separate or to count the (real) roots of such polynomials only few leading digits of the coefficients are enough. MPSolve, starts the computation with using few leading digits of the coefficients, and doubles the number of them at each step of the computation until the requested goal is reached. This leads to a substantial saving of the time cost in many concrete situations with respect to other algorithms based on explicit factorization. Typically, for the latter class of algorithms, the useful information that initially is confined in the leading digits of the coefficients, is scrambled in all the (many) digits of the coefficients of the factors after just the first factorization step.

3 Notes on the implementation

The program is written in ANSI C, makes use of the GMP multiprecision package, and of the extended types package MT (More Types) written by G. Fiorentino. Besides the type `double`, and `cplx` (implementing complex numbers having standard C `double` real and imaginary parts) the package uses the `rdpe` and the `cdpe` types consisting in real and complex numbers where the real components are represented as a pair (mantissa, exponent), with a C `double` mantissa and a `long int` exponent. The latter types are needed to perform computation within the precision (and with a slightly higher cost) of that of `double`, but allow a much wider range that avoids overflow and underflow situations. The notation “dpe”, which means Double Precision and Exponent, is borrowed by the MPFun package by D. Bailey. For higher precision computation MPSolve uses the multiprecision types of GMP and the multiprecision complex type `mpc` implemented by G. Fiorentino.

In order to achieve a high speed of computation, almost all the sub-algorithms of the package have been implemented in three slightly different versions: the “float” version, the “dpe” version and the “mp” version.

4 Using MPSolve

The executable program of the package MPSolve is called unisolve and can be used as:

`unisolve` *`options input_file`*

If the *`input_file`* is missing then unisolve will read from the standard input stream (typically the keyboard).

Options and directives

The program unisolve allows many *`options`* to change its default behavior, here is the list:

`-G`*x*: goal of the computation. Three goals are possible:

- `i`: isolate (default)
- `a`: approximate
- `c`: count

For instance, `unisolve -Gi` isolates the roots of the polynomial

`-S`*x*: set where the program looks for the roots. The available sets are

- `a`: all the complex plane (default)
- `l`: left half plane $\{z \in \mathbf{C} : \quad Re(z) < 0\}$
- `r`: right half plane $\{z \in \mathbf{C} : \quad Re(z) > 0\}$
- `u`: upper half plane $\{z \in \mathbf{C} : \quad Im(z) > 0\}$
- `d`: (down) lower half plane $\{z \in \mathbf{C} : \quad Im(z) < 0\}$
- `i`: inside the unit disk $\{z \in \mathbf{C} : \quad |z| < 1\}$
- `o`: outside the unit disk $\{z \in \mathbf{C} : \quad |z| > 1\}$
- `R`: Real line $\{z \in \mathbf{C} : \quad Im(z) = 0\}$
- `I`: Imaginary line $\{z \in \mathbf{C} : \quad Re(z) = 0\}$

For instance, `unisolve -Gc -SR` counts the real roots of the polynomial, `unisolve -Gi -So` isolates the roots out of the unit disk.

`-D`*x*: the program detects if the root is real/imaginary. For a detected real root the imaginary part is written as '0', similarly, the real part of a detected imaginary root is written as '0'. The options are:

- `r`: detect real
- `i`: detect imaginary
- `b`: detect both real and imaginary roots.
- `n`: do not detect (default)

`-M`*x*: the program detects if there are multiple roots there are two options detect `-M+`, do not detect `-M-` (default).

`-d`: if this switch is set then the program writes, in the standard error, information about the execution of the program. By using the modifier `-d1` all output goes to the standard output stream.

`-i`*x*: input precision in digits. If, say, the option `-i100` is set, then it is assumed that the coefficients of the input polynomial have 100 correct digits. This value overwrites the precision assigned in the input file. The option `-i0` (default) means infinite precision.

`-o`*x*: output precision. If, say, the option `-o100` is set, then the program will isolate/approximate/count the roots of the polynomial up to the precision of 100 digits. The default value is 30 digits. The detection of multiplicities and of real/imaginary roots, that are based on the computation of the Mahler bound, are affected by this value.

`-O`*x*: output format. The options for x are (here re and im denote real and imaginary parts of the root, respectively):

`c`: compact form (default): (re, im)

`b`: bare format: re \tab im

`g`: "Gnuplot", low precision format: re im, well suited as gnuplot input

`v`: verbose: Root(i) = re ± I im

`f`: full information: (re, im), a posteriori error bound, status of the approximation.

With the option `-Of` the real and imaginary parts are written integrally with no filtering of the correct digits. The status of the approximation is a triple of characters with the following meaning:

- First character: (cluster, isolated, approximated, multiplicity)

  m: multiple root

  i: isolated root

  a: approximated single root (relative error less than ϵ_{out})

  o: approximated cluster of roots (relative error less than ϵ_{out})

  c: cluster of roots not yet approximated (relative error greater than ϵ_{out})

- Second character: (real/imaginary roots)

  R: real root

  r: non-real root

  I: imaginary root

  i: non-imaginary root

  w: uncertain real/imaginary root

z: non-real and non-imaginary root

- Third character (set $\mathcal{S}$)

  i: root in $\mathcal{S}$

  o: root out of $\mathcal{S}$

  u: root uncertain

`-Lp`*x*: maximum number of iterations per packet (default 10). The Ehrlich-Aberth iterations are grouped into packets. The number of iterations per packet can be modified.

`-Li`*x*: maximum number of global iterations (default 100). The number of packets of the Ehrlich-Aberth iteration can be modified. Increasing this number may be needed for dealing with exceptionally hard polynomials.

The input file

The ***`input_file`*** must contain:

- some optional lines of comments, introduced by the character '`!`'.
- a string of three characters `abc` where the first character of the string specifies the way the polynomial is assigned, the second and the third specify the type of the coefficients.
- the input precision of the coefficients, in digits (0 means infinite precision).
- degree and coefficients of the polynomial coded according to the values of `abc`.

More precisely:

```
a='s' means sparse polynomial;
a='d' means dense polynomial;
a='u' means polynomial provided by the user by
      means of a c program in the file mps_usr.c
      (a sample of this file is provided for the
      Mandelbrot polynomial);

b='r' means real coefficients;
b='c' means complex coefficients;

c='i' means integer real/imaginary parts;
c='q' means rational real/imaginary parts;
c='b' means bigfloat real/imaginary parts;
c='f' means float real/imaginary parts.
```

For `a`='s' the file contains:

```
n: the degree of the polynomial;
n_coeff: the number of the nonzero coefficients;
i:    indices of the generic nonzero coefficient;
a_i:  the coefficient of x^i;
...   the order of this list of coefficients is
      not relevant.
```

For a=‘d’ the file contains:

```
n: the degree of the polynomial
a_0 constant coefficient
a_1 coefficient of x
...
```

For b=‘r’ the coefficients are stored as single number for c $\neq$ ‘q’ while for c=‘q’, they are stored as

```
num: numerator
den: denominator
```

For b=‘c’ the coefficient are stored as pairs provided that c $\neq$ ‘q’, i.e.,

```
re: real_part
im:  imaginary_part
```

otherwise, for c=‘q’ they are stored as quadruples:

```
real_numer: numerator of the real part
real_denom: denominator of the real part
imag_numer: numerator of the imaginary part
imag_denom: denominator of the imaginary  part
```

For instance the following data defines the Kameny polynomial

```
! Kameny polynomial, sparse, complex, integer.
! p(x)=ic^3 x^7+ c^4 x^2- 6c^2x+9, c=10^6
sci
0
7
4
0
 9
 0
1
 -6000000000000
 0
2
 1000000000000000000000010
 0
7
 0
 1000000000000000000
```

The following data define the Wilkinson polynomial of degree 20.

```
! Wilkinson polynomial n=20: p(x)=\prod_{i=0}^n (x-i)
dri
0
20
2432902008176640000
-8752948036761600000
13803759753640704000
-12870931245150988800
8037811822645051776
-3599979517947607200
1206647803780373360
-311333643161390640
63030812099294896
-10142299865511450
1307535010540395
-135585182899530
11310276995381
-756111184500
40171771630
-1672280820
53327946
-1256850
20615
-210
1
```

Finally, the data

```
uri
0
511
```

define the user polynomial (by default it is the Mandelbrot polynomial) of degree 511.

The program writes to the standard output stream, and, if the `-d` switch is set, also to the standard error stream. The information written in the standard output are: the approximations of the roots, where each root is written as `(real_part, imaginary_part)`. The information written to the standard error stream concern the execution of the program, moreover, also the approximations of the roots are reported, where the roots are not replaced by 0.0 if the relative error is greater than 1. Also the absolute error bound of each approximation is reported. This information may be useful in the case of polynomials with approximate input.

5 The test suite

Some input test polynomials are provided in the directory `Data`; the directory `Results` contains output files for some of the test polynomials.

The test polynomials have the following features:

- The polynomials from `spiral10` to `spiral30` are defined by

  $$p(x) = (x+1)(x+1+a)(x+1+a+a^2)...(x+1+a+a^2+...+a^n),$$
  $$a = \mathbf{i}/1000, \quad \mathbf{i}^2 = -1, \quad n = 10, 15, 20, 25, 30.$$

  The polynomials from `geom1_10` to `geom1_40` and from `geom2_10` to `geom2_40` are defined by $p(x) = (x+1)(x+a)(x+a^2)...(x+a^{n-1})$, with $a = 100\mathbf{i}$ and $a = \mathbf{i}/100$. respectively. The polynomials from `geom3_10` to `geom3_80` are defined by $4^{n(n+1)/2}\prod_{i=1}^{n}(x - 1/4^i)$, for $n = 10, 20, 40, 80$. The polynomials from `geom4_10` to `geom4_80` are defined by $\prod_{i=1}^{n}(x - 4^i)$ for $n = 10, 20, 40, 80$.

- The polynomials from `easy100` to `easy6400` are defined by $p(x) = \sum_{i=0}^{n}(i+1)x^i$, for $100, 200, 400, 800, 1600, 3200, 6400$.

- The polynomials from `mandel31` to `mandel1023` are the Mandel polynomials of degrees from 31 to 1023, defined in section 1 and assigned in terms of their coefficients.

- The polynomials from `umand31` to `umand2047` are the Mandel polynomials of degrees from 31 to 2047, assigned in terms of a straight line program (file `mps_usr.c`).

- The polynomials from `exp50` to `exp400` are the truncation of the exponential series to the degree $n = 50, 100, 200, 400$, respectively, i.e., $\sum_{i=0}^{n} x^i/i!$.

- Curz polynomials of degree 20 40 80 160 `curz20`–`curz160` defined by: $p_{j+1}(x) = x\sum_{i=0}^{j-1} p_{j-i}(-1)^i/(i+1) + (-1)^j/(j+1)$, $j = 1, 2, \ldots$, for $p_1 = 1$.

- The polynomials `kam1_1`, `kam1_2`, `kam1_3` belong to the class

  $$(x - 3c^2)^2 + icx^7,$$

  for $c = 10^{-6}, 10^{-20}, 10^{-70}$, respectively. The polynomials have two simple complex roots with extremely small real separation (33, 113, 385, common digits respectively) and imaginary parts close to zero (of modulus about $10^{-44}, 10^{-149}, 10^{-524}$, respectively). The remaining roots are well separated.

- The polynomials `kam2_1`, `kam2_2`, `kam2_3` belong to the class

  $$(c^2x^2 - 3)^2 + ic^2x^9,$$

for $c = 10^6, 10^{20}, 10^{70}$, respectively. The polynomials have four simple complex roots with small imaginary parts ($10^{-27}, 10^{-90}, 10^{-315}$, respectively,) in two close sets (21, 70, 247 common digits). The remaining roots are well separated.

- The polynomials `kam3_1`, `kam3_2`, `kam3_2` belong to the class

  $$(c^2x^2 - 3)^2 + c^2x^9,$$

  for $c = 10^6, 10^{20}, 10^{70}$, respectively. The polynomials have two extremely close real roots (20, 70, 247 common digits) plus a complex pair with extremely small imaginary parts
  ($10^{-27}, 10^{-90}, 10^{-315}$) The remaining roots are well separated, one of them is real.

- The polynomial `kam4` is defined as follows.

  $$x^{14} + 2 \cdot 10^{24} \cdot x^{11} + 10^{48} \cdot x^8 + 4 \cdot x^7 - 4 \cdot 10^{24} \cdot x^4 + 4$$

  The polynomial has a cluster of 8 roots of modulus $\mathtt{1.89} \cdot \mathtt{10}^{-6}$, a cluster of 6 roots having modulus $\mathtt{10}^{8}$. The cluster made up by 8 roots has: two real roots having 21 common digits two pairs of complex roots having small real parts, and imaginary parts with 21 common digits The cluster made up by 6 roots has two very close real roots matching in their first 28 digits, and 2 very close complex pairs whose real and imaginary parts match in their first 27 digits.

- The polynomials from `mig1_20`, to `mig1_500`, and from `mig1_50_1` to `mig1_500_1` are Mignotte-like polynomials in the class

  $$x^n + (ax + 1)^m, \quad n >> m > 1, \ |a| > 1,$$

  These polynomials have m very close roots around $-1/a$. The polynomials have been chosen by setting $m = 3$ and $m = 31$.

- The polynomial `mig2` is defined by

  $$(x^{n-k} + (ax + 1)^m)(ax + 1)^k, \quad n >> m > 1, \ n >> k > 1, \ |a| > 1,$$

  for $n = 20, m = 3, k = 3, a = 100i$. The polynomial has a cluster of m very close roots around $-1/a$ where $-1/a$ is also a multiple root of multiplicity k.

- The polynomials `lar1` and `lar1_200` are defined as

  $$x^n + 10^{300}x^{14} + x^5 + 1,$$

  have a coefficient with modulus close to the largest number representable as double in the IEEE standard. Overflow conditions/failure are likely for programs that do not use extended arithmetic. The polynomials are obtained with $a = 1$ and $n = 20, 200$, respectively.

- The polynomial `lar2` is defined by

  $$x^n + x^{11} + 10^{300}x + 10^{-300}, n = 20.$$

  The polynomial has coefficients that have modulus close to the largest and to the smallest numbers representable as double in the IEEE standard. Overflow conditions/failure are likely for programs that do not use extended arithmetic. The polynomial, even if its coefficients are representable as double, has a root which is too small in order to be represented as a nonzero double.

- the polynomial `lar3` is defined by

  $$10^{-200}x^n + 10^{200}x^{n-1} + 10^{200}, \quad n = 20$$

  The polynomial cannot be normalized in the standard IEEE arithmetic without generating an overflow condition. The polynomial, even if its coefficients are representable as double, has a root which cannot be represented as a double without generating an overflow condition. Failure is likely for programs that do not use extended arithmetic.

- The polynomial `lar4` is defined by $10^{20} + 10^{2000}x^{19} + 10^{-1600}x^{23}$.

- The polynomial `lar5` is defined by $10^{2000} + 10^{2000}x^{19} + 10x^{20}$.

- Polynomial `lsr1`

  $1 + 30000x + 300000000x^2 + 1000000000000x^3 + x^{200} + 400000000x^{298}+$
  $12000000040000x^{299} + 120000001200000001x^{300}+$
  $400000012000000030000x^{301} + 400000000300000000x^{302}+$
  $1000000000000x^{303} + 400000000x^{498}+$
  $40000x^{499} + x^{500}$

  The polynomial has a cluster of two very close roots of large moduli and a cluster of three very close roots of small moduli.

- Polynomial `lsr2` given by

  $$x^{500} + 100^{200}x^{499} + 10^{500}x^{495} + 10^{499}x^{480} + 10^3x^3 + 3 \cdot 10^2x^2 + 30x + 1$$

  The polynomial has roots with large and small moduli.

- Polynomial `lsr3` defined by $1+x*30+x^2*300+x^3*1000+x^{480}*10^{495}+x^{495}*10^{500}+x^{499}*10^{200}+x^{500}$.

- Polynomials `lsr4_1`,`lsr4_2`, `lsr4_3` are given by $(x^{50}+1)(x^2+ax+a^{-1})$, $a = 10^{10}, 10^{20}, 10^{40}$, respectively.

- Polynomial `lsr_200` $(x^{12} - (10^{20}x - 1)^4)(1 + (10^{20} + x)^4x^8)(x^{200} + 1)$. There are 4 roots of modulus 10^{-20} matching in the first 60 digits and 4 roots of modulus 10^{20} matching in the first 60 digits.

- Polynomial `lsr_24` $(x^{12}-(10^{20}x-1)^4)(1+(10^{20}+x)^4x^8)$.
  There are 4 roots of modulus 10^{-20} matching in the first 60 digits and 4 roots of modulus 10^{20} matching in the first 60 digits.

- The polynomials from `nroots50` to `nroots1600` are of the kind z^n-1, $n=50,100,200,400,800,1600$.

- The polynomials from `wilk20` to `wilk320` are the Wilkinson polynomial of degree 20,40,80,160,320, defined by

$$\prod_{i=1}^{n}(x-i), \quad (n=20).$$

  Coefficients are very large and the roots are very ill-conditioned already for $n=20$. The condition number of the roots, for $n=20$ ranges from 420 to 10^{14}.

- The polynomial `wilk_mod1` is a modified Wilkinson polynomial defined as follows

$$((x^{10}/10^{20}+(x-10)^2)\prod_{i=1}^{20}(x-i).$$

  The polynomial is obtained by multiplying Wilkinson's polynomial of degree 20 with a Mignotte-like polynomial having two roots close to 10. Besides the ill conditioning of the integer roots, there is a cluster of two roots close to 10. The condition number of the root 10 is about 10^{26}.

- The polynomial `wilk_mod2` is a modified Wilkinson polynomial defined as

$$\prod_{i=1}^{20}(x-i)(x-20)^2.$$

  Besides the ill conditioning of the roots, there is a multiple root.

- The polynomials `kir1_10`, `kir1_20`, `kir1_40`, proposed by Peter Kirrinnis, are defined as follow $(z^4-1/16)^n(z^4-(1/2+eps)^4)16^n/eps^4$, $eps=1/4096$, $n=10,20,40$ and have degree $4n+4$. They have multiple roots with high multiplicity and some of them clustered.

- The polynomials `kir1_10_mod`, `kir1_20_mod`, `kir1_40_mod`, are obtained by adding z^{4n+4} to the Kirrinnis polynomials. In this way they are defined as follow $(z^4-1/16)^n(z^4-(1/2+eps)^4)16^n/eps^4+z^{4n+4}$, $eps=1/4096$, $n=10,20,40$ and have degree $4n+4$. They have simple clustered roots.

- The polynomial `nektarios`, provided by Nektarios Psycaris, has degree 648 and arises in the study of certain problems in physics.

- The polynomial `trv_m`, provided by Carlo Traverso, arises from the symbolic processing of a system of polynomial equations. It has multiple roots.

- The polynomial `katsura8` arises from the symbolic preprocessing of a system of polynomial equations.
- The polynomials `chrm*` are chromatic polynomials.

The HTML file `bench.htm` in the `Doc` directory reports the timings, in seconds, of the programs NSolve, fsolve, polroots and unisolve of the packages Mathematica, Maple, Pari and MPSolve respectively. All computations have been performed on a Pentium 200 MMX with 64 Mb of physical RAM.

We performed computations with several goals, like isolating all the roots, approximating all of them with 50 or 1000 digits, counting the real roots or those in the unit disk or those in the right half plane, approximating only real roots, etc.

Failures, denote with an 'F' in the table, occurred in Maple for internal errors like '*unable to compute the norm*', or for unsuccessful computation of the roots. In Pari we had failures only due to stack overflows, while in Mathematica we had unsuccessful computation of the roots. We point out that the Mathematica command NSolve, when used with the d digits of working precision, say `NSolve[p[x]==0,x,100]` for $d = 100$, does not guarantee d correct output digits. In fact the number of correct output digits depends on the conditioning of the root. In particular, in the case of a root of multiplicity m, in order to have at least k correct digits we had to choose $d = km$, i.e., to type `NSolve[p[x]==0, x, k*m]`.

Timings are complete only for the package MPSolve.

6 User defined polynomials.

The file `mps_user.c` contains the "user defined polynomial", more precisely, the programs `fnewton_usr, dnewton_usr, mnewton_usr` that are the float, dpe, mp versions, respectively of the same computation. Actually, the float version is not used by `unisolve` (indeed, the float version cannot be as robust as the dpe version for overflow reasons) and has been inserted as a more readable example.

Given on input the value of x, each program in the file `mps_user.c` must compute:

- The value of $p(x)$
- The value of $p'(x)$
- An upper bound σ to the number $|p(x) - fl(p(x))|$, where $fl(\cdot)$ denotes the floating point computation of the expression between parentheses,

and output the following values:

- the Newton correction `corr`$= p(x)/p'(x)$;
- the inclusion radius `rad`$= n(|p(x)| + \sigma)/|p'(x)|$;

- the Boolean value `again`$= (|fl(p(x))| > \sigma)$, where if `again` is true then the Newton iteration is continued. Observe that `again` true means that the computed value is affected by a relative roundoff error less than 1, i.e., $|fl(p(x)) - p(x)|/|fl(p(x))| < \sigma/fl(p(x)) < 1$. Therefore it brings still information.

Indeed, computing $p(x)$ and $p'(x)$ is almost straightforward if the formal relations defining the polynomial are known. For instance, for the Mandelbrot polynomial defined in (1), it is immediate to write a program computing $p(x)$. For the computation of $p'(x)$ we may use the following relation obtained by differentiating (1).

$$\begin{aligned}&p'_0 = 0\\&p'_i = p_{i-1}^2 + 2 \cdot x \cdot p_{i-1} \cdot p'_{i-1}\end{aligned}$$

In order to implement the stop condition and to give a guaranteed a-posteriori error bound, a rounding error analysis of the computation must be performed.

Let us show this for the Mandelbrot polynomial. Let $\widetilde{p} = fl(p)$ denote the value obtained by applying (1) in floating point arithmetic with machine precision μ and recall the following relations (compare [1]) $fl(a*b) = a*b(1+\epsilon)$ and $fl(a+b) = (a+b)(1+\delta)$, where $|\epsilon| < 2\sqrt{2}\mu$ and $|\delta| < \mu$, which hold for complex values a, b that do not generate overflow/underflow in the computation of $a+b$ and $a*b$ (here μ is the machine precision).

By applying the above relations to (1) we easily obtain

$$\begin{aligned}&\widetilde{p}_i = x\widetilde{p}_{i-1}^2(1+\theta_i) + (1+\sigma_i)\\&|\theta_i| < 6.7\mu + O(\mu^2),\ |\sigma_i| < \mu + O(\mu^2)\end{aligned}$$

Subtracting both the members of the above relation and of (1), ignoring $O(\mu^2)$ terms yields the following expressions that allow us to compute a bound to the absolute error $e_i = |p_i - \widetilde{p}_i|$

$$\begin{aligned}&e_0 = 0\\&e_i < 2|x| \cdot e_{i-1}|\widetilde{p}_{i-1}| + 7.7(|x| \cdot |\widetilde{p}_{i-1}| + 1)\mu\end{aligned}$$

The rounding error analysis can be performed in a rough way obtaining very weak bounds, or in a very accurate way obtaining sharp bounds.

The more accurate is the roundoff error bound, the more efficient is the stop criterion.

References

[1] D. A. Bini and G. Fiorentino, Design, Analysis, and Implementation of a Multiprecision Polynomial Rootfinder, Numerical Algorithms, 23, 2000, 127–173.

[2] D. A. Bini, Numerical computation of polynomial zeros by means of Aberth's method, Numerical Algorithms, 13, 1996, 179–200.

[3] D. A. Bini and G. Fiorentino, A multiprecision implementation of a poly-algorithm for univariate polynomial zeros, Proc. of The POSSO Workshop on Software, Paris, 1995, J.C. Faugère, J. Marchand, R. Rioboo editors.
[4] F. Rouillier, RUR, FRISCO Report 3.3.2.3.1.