Module 10.1: nag_polynom_eqn
Roots of Polynomials

`nag_polynom_eqn` provides a procedure for computing the roots of a polynomial with real or complex coefficients.

Contents

Procedure: nag_polynom_roots

1 Description

nag_polynom_roots is a procedure for finding all the roots of the nth degree real or complex polynomial equation

$$P(z) = a_n z^n + a_{n-1} z^{n-1} + a_{n-2} z^{n-2} + \cdots + a_1 z + a_0 = 0,$$

using a variant of Laguerre's Method.

2 Usage

```
USE nag_polynom_eqn
CALL nag_polynom_roots(a, z  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$n \geq 1$ — the degree of the polynomial

3.1 Mandatory Arguments

a($n+1$) — real(kind=*wp*) / complex(kind=*wp*), intent(in)

Input: the coefficients of the polynomial. If a is declared with bounds $(0 : n)$, then a(i) must contain a_i (i.e., the coefficient of z^i) for $i = 0, 1, \ldots, n$.

Constraints: the coefficient of z^n, a($n+1$) $\neq 0.0$.

z(n) — complex(kind=*wp*), intent(out)

Output: the roots of the polynomial, stored in z(i), for $i = 1, 2, \ldots, n$.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

scale — logical, intent(in), optional

Input: indicates whether or not the polynomial is to be scaled.

Default: scale = .true..

Note: see Section 6.2 for advice on when it may be preferable to set scale = .false. and for a description of the scaling strategy.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
303	Array arguments have inconsistent shapes.

Failures (error%level = 2):

error%code	Description
201	The iterative procedure has failed to converge.
	For this procedure this failure should not normally occur; please contact NAG.
202	Overflow/underflow prevents the evaluation of $P(z)$ near some of its zeros.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 1 and 2 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The roots are located using a modified form of Laguerre's Method, originally proposed by Smith [4].

The method of Laguerre (see Wilkinson [5]) can be described by the iterative scheme

$$L(z_k) = z_{k+1} - z_k = \frac{-nP(z_k)}{P'(z_k) \pm \sqrt{H(z_k)}},$$

where $H(z_k) = (n-1)[(n-1)(P'(z_k))^2 - nP(z_k)P''(z_k)]$, and z_0 is specified.

The sign in the denominator is chosen so that the modulus of the Laguerre step at z_k, namely $|L(z_k)|$, is as small as possible. The method can be shown to be cubically convergent for isolated roots (real or complex) and linearly convergent for multiple roots.

The procedure generates a sequence of iterates z_1, z_2, z_3,..., such that $|P(z_{k+1})| < |P(z_k)|$ and ensures that $z_{k+1} + L(z_{k+1})$ 'roughly' lies inside a circular region of radius $|F|$ about z_k known to contain a zero of $P(z)$; that is, $|L(z_{k+1})| \leq |F|$, where F denotes the Féjer bound (see Marden [2]) at the point z_k. Following Smith [4], F is taken to be $\min(B, 1.445nR)$, where B is an upper bound for the magnitude of the smallest root, given by

$$B = 1.0001 \times \min(\sqrt{n} \times L(z_k), |r_1|, |a_0/a_n|^{1/n}),$$

where r_1 is the zero, X, of smaller magnitude of the quadratic equation

$$(P''(z_k)/(n(n-1)))X^2 + 2(P'(z_k)/n)X + P(z_k) = 0$$

and the Cauchy lower bound R for the smallest root is computed (using Newton's Method) as the positive root of the polynomial equation

$$|a_n|z^n + |a_{n-1}|z^{n-1} + |a_{n-2}|z^{n-2} + \cdots + |a_1|z - |a_0| = 0.$$

Starting from the origin, successive iterates are generated according to the rule $z_{k+1} = z_k + L(z_k)$ for $k = 1, 2, 3, \ldots$, and $L(z_k)$ is 'adjusted' so that $|P(z_{k+1})| < |P(z_k)|$ and $|L(z_{k+1})| \leq |F|$. The iterative procedure terminates if $P(z_{k+1})$ is smaller in absolute value than the bound on the rounding error in $P(z_{k+1})$ and the current iterate $z_p = z_{k+1}$ is taken to be a root of $P(z)$ (as is its conjugate $\bar{z}_p$ if z_p is complex when $P(z)$ has real coefficients). A deflated polynomial, $\tilde{P}(z)$, is then formed as follows.

- $\tilde{P}(z) = P(z)/(z - z_p)$, degree $(n-1)$, if z_p is real and $P(z)$ has real coefficients;
- $\tilde{P}(z) = P(z)/\left((z - z_p)(z - \bar{z}_p)\right)$, degree $(n-2)$, if z_p is complex and $P(z)$ has real coefficients;
- $\tilde{P}(z) = P(z)/(z - z_p)$, degree $(n-1)$, if $P(z)$ has complex coefficients.

The above procedure is then repeated on the deflated polynomial until $n < 3$, whereupon the remaining roots are obtained via the 'standard' closed formulae for a linear ($n = 1$) or quadratic ($n = 2$) equation.

6.2 Scaling

If `scale` = `.true.` (the default), then a scaling factor for the coefficients is chosen as a power of the base b of the machine so that the largest coefficient in magnitude approaches $thresh = b^{e_{\max}-p}$, where b = `RADIX(1.0_wp)`, p = `DIGITS(1.0_wp)`, and $e_{\max}$ = `MAXEXPONENT(1.0_wp)`.

Note that no scaling is performed if the largest coefficient in magnitude exceeds $thresh$, even if `scale` = `.true.`.

However, with `scale` = `.true.`, overflow may be encountered when the input coefficients $a_0, a_1, a_2, \ldots, a_n$ vary widely in magnitude, particularly on those machines for which $b^{(4p)}$ overflows. In such cases, `scale` should be set to `.false.` and the coefficients scaled so that the largest coefficient in magnitude does not exceed $b^{(e_{\max}-2p)}$.

Even so, the scaling strategy used in this procedure is sometimes insufficient to avoid overflow and/or underflow conditions. In such cases, you are recommended to scale the independent variable (z) so that the disparity between the largest and smallest coefficient in magnitude is reduced. That is, use the procedure to locate the zeros of the polynomial $d \times P(cz)$ for some suitable values of c and d. For example, if the original polynomial was $P(z) = 2^{-100}i + 2^{100}z^{20}$, then choosing $c = 2^{-10}$ and $d = 2^{100}$, for instance, would yield the scaled polynomial $i + z^{20}$, which is well behaved relative to overflow and underflow and has zeros which are 2^{10} times those of $P(z)$.

Example 1: Roots of a polynomial with real coefficients

The following example program can be used to find the roots of an nth degree real polynomial. It is used to find the roots of the 5th degree polynomial $z^5 + 2z^4 + 3z^3 + 4z^2 + 5z + 6 = 0$.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_polynom_eqn_ex01

  ! Example Program Text for nag_polynom_eqn
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_polynom_eqn, ONLY : nag_polynom_roots
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, n
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: a(:)
  COMPLEX (wp), ALLOCATABLE :: z(:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_polynom_eqn_ex01'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) n

  ALLOCATE (a(0:n),z(n))       ! Allocate storage

  ! Read the polynomial coefficients
  READ (nag_std_in,*) a

  ! Find the roots

  CALL nag_polynom_roots(a,z)

  WRITE (nag_std_out,'(/,1X,A,I4,/)') 'Degree of Polynomial =', n

  DO i = 1, n
    WRITE (nag_std_out,'(1X,A,"(",E12.4,",",E12.4,")")') 'z = ', z(i)
  END DO

  DEALLOCATE (a,z)             ! Deallocate storage

END PROGRAM nag_polynom_eqn_ex01
```

2 Program Data

```
Example Program Data for nag_polynom_eqn_ex01
5                        : n ( Degree of Polynomial )
6.0 5.0 4.0 3.0 2.0 1.0  : a ( Vector of Coefficients )
```

3 Program Results

```
Example Program Results for nag_polynom_eqn_ex01

Degree of Polynomial =   5

z = ( -0.1492E+01,  0.0000E+00)
z = (  0.5517E+00,  0.1253E+01)
z = (  0.5517E+00, -0.1253E+01)
z = ( -0.8058E+00,  0.1223E+01)
z = ( -0.8058E+00, -0.1223E+01)
```

Example 2: Roots of a polynomial with complex coefficients

The following example program can be used to find the roots of an nth degree complex polynomial. It is used to find the roots of the polynomial $a_5z^5 + a_4z^4 + a_3z^3 + a_2z^2 + a_1z + a_0 = 0$, where $a_5 = 5 + 6i$, $a_4 = 30 + 20i$, $a_3 = -0.2 - 6i$, $a_2 = 50 + 100000i$, $a_1 = -2 + 40i$ and $a_0 = 10 + i$.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_polynom_eqn_ex02

  ! Example Program Text for nag_polynom_eqn
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_polynom_eqn, ONLY : nag_polynom_roots
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, n
  ! .. Local Arrays ..
  COMPLEX (wp), ALLOCATABLE :: a(:), z(:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_polynom_eqn_ex02'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) n

  ALLOCATE (a(0:n),z(n))       ! Allocate storage

  ! Read the polynomial coefficients
  READ (nag_std_in,*) a

  ! Find the roots

  CALL nag_polynom_roots(a,z)

  WRITE (nag_std_out,'(/,1X,A,I4,/)') 'Degree of Polynomial =', n

  DO i = 1, n
    WRITE (nag_std_out,'(1X,A,"(",E12.4,",",E12.4,")")') 'z = ', z(i)
  END DO

  DEALLOCATE (a,z)             ! Deallocate storage

END PROGRAM nag_polynom_eqn_ex02
```

2 Program Data

```
Example Program Data for nag_polynom_eqn_ex02
5                          : n ( Degree of Polynomial)
   ( 10.0  ,      1.0 )
   ( -2.0  ,     40.0 )
   ( 50.0  , 100000.0 )
   ( -0.2  ,     -6.0 )
   ( 30.0  ,     20.0 )
   (  5.0  ,      6.0 )   : End of a ( Vector of Coefficients )
```

3 Program Results

```
 Example Program Results for nag_polynom_eqn_ex02

 Degree of Polynomial =   5

 z = ( -0.2433E+02, -0.4855E+01)
 z = (  0.5249E+01,  0.2274E+02)
 z = (  0.1465E+02, -0.1657E+02)
 z = ( -0.6926E-02, -0.7443E-02)
 z = (  0.6526E-02,  0.7423E-02)
```

Mathematical Background

1 Introduction

Let

$$f(z) \equiv a_n z^n + a_{n-1} z^{n-1} + a_{n-2} z^{n-2} + \cdots + a_1 z + a_0, \quad a_n \neq 0$$

be a polynomial of degree n with complex coefficients a_i. A complex number z_1 is called a *zero* of $f(z)$ (or equivalently a *root* of the *equation* $f(z) = 0$), if $f(z_1) = 0$.

If z_1 is a zero, then $f(z)$ can be divided by a factor $(z - z_1)$:

$$f(z) = (z - z_1) f_1(z) \tag{1}$$

where $f_1(z)$ is a polynomial of degree $n - 1$. By the Fundamental Theorem of Algebra, a polynomial $f(z)$ has at least one zero, and so the process of dividing out factors $(z - z_i)$ can be continued until we have a complete *factorization* of $f(z)$:

$$f(z) \equiv a_n (z - z_1)(z - z_2) \cdots (z - z_n).$$

Here the complex numbers $z_1, z_2, \ldots, z_n$ are the zeros of $f(z)$; they may not all be distinct, so it is sometimes more convenient to write:

$$f(z) \equiv a_n (z - z_1)^{m_1} (z - z_2)^{m_2} \cdots (z - z_k)^{m_k}, \quad k \leq n,$$

with distinct zeros $z_1, z_2, \ldots, z_k$ and multiplicities $m_i \geq 1$. If $m_i = 1$, z_i is called a *single* zero; if $m_i > 1$, z_i is called a *multiple* or *repeated* zero; a multiple zero is also a zero of the derivative of $f(z)$.

If the coefficients of $f(z)$ are all real, then the zeros of $f(z)$ are either real or else occur as pairs of conjugate complex numbers $a + ib$ and $a - ib$. A pair of complex conjugate zeros are the zeros of a quadratic factor of $f(z)$, $(z^2 + rz + s)$, with real coefficients r and s.

Although polynomials are generally regarded as simple functions, the problem of numerically *computing* the zeros of an arbitrary polynomial is far from simple. A great variety of algorithms have been proposed, of which a number have been widely used in practice; for a fairly comprehensive survey, see Householder [1]. All general algorithms are iterative. Most converge to one zero at a time; the corresponding factor can then be divided out as in (1) above (this process is called deflation or, loosely, dividing out the zero) and the algorithm can be applied again to the polynomial $f_1(z)$. A pair of complex conjugate zeros can be divided out together; this corresponds to dividing $f(z)$ by a quadratic factor.

Whatever the theoretical basis of the algorithm, a number of *practical problems* arise; for a thorough discussion of some of them see Peters and Wilkinson [3]. The most elementary point is that, even if z_1 is mathematically an exact zero of $f(z)$, because of the fundamental limitations of computer arithmetic the *computed* value of $f(z_1)$ will not necessarily be exactly 0.0. In practice there is usually a small region of values of z about the exact zero at which the computed value of $f(z)$ becomes dominated by rounding errors. Moreover in many algorithms this inaccuracy in the computed value of $f(z)$ results in a similar inaccuracy in the computed step from one iterate to the next. This limits the precision with which any zero can be computed. Deflation is another potential cause of trouble, since, in the notation of (1), the computed coefficients of $f_1(z)$ will not be completely accurate, especially if z_1 is not an exact zero of $f(z)$; so the zeros of the computed $f_1(z)$ will deviate from the zeros of $f(z)$.

A zero is called *ill conditioned* if it is sensitive to small changes in the coefficients of the polynomial. An ill conditioned zero is likewise sensitive to the computational inaccuracies just mentioned. Conversely a zero is called *well conditioned* if it is comparatively insensitive to such perturbations. Roughly speaking a zero which is well separated from other zeros is well conditioned, while zeros which are close together are ill conditioned, but in talking about 'closeness' the decisive factor is not the absolute distance between neighbouring zeros but their *ratio*: if the ratio is close to 1 the zeros are ill conditioned. In particular, multiple zeros are ill conditioned. A multiple zero is usually split into a cluster of zeros by perturbations in the polynomial or computational inaccuracies.

References

[1] Householder A S (1970) *The Numerical Treatment of a Single Nonlinear Equation* McGraw-Hill

[2] Marden M (1966) Geometry of polynomials *Mathematical Surveys* **3** American Mathematical Society, Providence, RI

[3] Peters G and Wilkinson J H (1971) Practical problems arising in the solution of polynomial equations *J. Inst. Maths. Applics.* **8** 16–35

[4] Smith B T (1967) ZERPOL: A zero finding algorithm for polynomials using Laguerre's method *Technical Report* Department of Computer Science, University of Toronto, Canada

[5] Wilkinson J H (1965) *The Algebraic Eigenvalue Problem* Oxford University Press, London