NAG Toolbox

nag_zeros_poly_complex (c02af)

1 Purpose

nag_zeros_poly_complex (c02af) finds all the roots of a complex polynomial equation, using a variant of Laguerre's method.

2 Syntax

```
[z, ifail] = nag_zeros_poly_complex(a, n, 'scal', scal)
[z, ifail] = c02af(a, n, 'scal', scal)
```

3 Description

nag_zeros_poly_complex (c02af) attempts to find all the roots of the nth degree complex polynomial equation

$$P(z) = a_0 z^n + a_1 z^{n-1} + a_2 z^{n-2} + \cdots + a_{n-1} z + a_n = 0.$$

The roots are located using a modified form of Laguerre's method, originally proposed by Smith (1967).

The method of Laguerre (see Wilkinson (1965)) 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 $H(z_k) = (n-1)\Big[(n-1)(P'(z_k))^2 - nP(z_k)P''(z_k)\Big]$ and z_0 is specified.

The sign in the denominator is chosen so that the modulus of the Laguerre step at z_k, viz. $|L(z_k)|$, 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 function generates a sequence of iterates $z_1, z_2, z_3, \ldots$, 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 Fejér bound (see Marden (1966)) at the point z_k. Following Smith (1967), F is taken to be $\min(B, 1.445nR)$, where B is an upper bound for the magnitude of the smallest zero given by

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

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

$$\frac{P''(z_k)}{2n(n-1)}X^2 + \frac{P'(z_k)}{n}X + \tfrac{1}{2}P(z_k) = 0$$

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

$$|a_0|z^n + |a_1|z^{n-1} + |a_2|z^{n-2} + \cdots + |a_{n-1}|z - |a_n| = 0.$$

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 zero of $P(z)$. The deflated polynomial $\tilde{P}(z) = P(z)/\big(z - z_p\big)$ of degree $n-1$ is then formed, and the above procedure is 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.

Note that nag_zeros_quadratic_complex (c02ah), nag_zeros_cubic_complex (c02am) and nag_zeros_quartic_complex (c02an) can be used to obtain the roots of a quadratic, cubic ($n = 3$) and quartic ($n = 4$) polynomial, respectively.

4 References

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

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

Thompson K W (1991) Error analysis for polynomial solvers *Fortran Journal (Volume 3)* **3** 10–13

Wilkinson J H (1965) *The Algebraic Eigenvalue Problem* Oxford University Press, Oxford

5 Parameters

5.1 Compulsory Input Parameters

1: **a**(**2**, **n** + **1**) – REAL (KIND=nag_wp) array

a$(1, i+1)$ and **a**$(2, i+1)$ must contain the real and imaginary parts of a_i (i.e., the coefficient of z^{n-i}), for $i = 0, 1, \ldots, n$.

Constraint: **a**$(1,1) \neq 0.0$ or **a**$(2,1) \neq 0.0$.

2: **n** – INTEGER

n, the degree of the polynomial.

Constraint: **n** ≥ 1.

5.2 Optional Input Parameters

1: **scal** – LOGICAL

Suggested value: **scal** = *true*.

Default: *true*

Indicates whether or not the polynomial is to be scaled. See Section 9 for advice on when it may be preferable to set **scal** = *false* and for a description of the scaling strategy.

5.3 Output Parameters

1: **z**(**2**, **n**) – REAL (KIND=nag_wp) array

The real and imaginary parts of the roots are stored in **z**$(1, i)$ and **z**$(2, i)$ respectively, for $i = 1, 2, \ldots, n$.

2: **ifail** – INTEGER

ifail $= 0$ unless the function detects an error (see Section 5).

6 Error Indicators and Warnings

Errors or warnings detected by the function:

ifail $= 1$

On entry, **a**$(1,1) = 0.0$ and **a**$(2,1) = 0.0$,
or **n** < 1.

ifail $= 2$

The iterative procedure has failed to converge. This error is very unlikely to occur. If it does, please contact NAG, as some basic assumption for the arithmetic has been violated. See also Section 9.

ifail $= 3$

Either overflow or underflow prevents the evaluation of $P(z)$ near some of its zeros. This error is very unlikely to occur. If it does, please contact NAG. See also Section 9.

ifail $= -99$

An unexpected error has been triggered by this routine. Please contact NAG.

ifail $= -399$

Your licence key may have expired or may not have been installed correctly.

ifail $= -999$

Dynamic memory allocation failed.

7 Accuracy

All roots are evaluated as accurately as possible, but because of the inherent nature of the problem complete accuracy cannot be guaranteed. See also Section 10.

8 Further Comments

If **scal** $= true$, 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 $thresh = b^{e_{\max}-p}$. You should note that no scaling is performed if the largest coefficient in magnitude exceeds $thresh$, even if **scal** $= true$. (b, $e_{\max}$ and p are defined in Chapter X02.)

However, with **scal** $= true$, 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, **scal** 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 by nag_zeros_poly_complex (c02af) 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 function to locate the zeros of the polynomial $dP(cz)$ 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 $P(z)$.

If the function fails with **ifail** $= 2$ or 3, then the real and imaginary parts of any roots obtained before the failure occurred are stored in **z** in the reverse order in which they were found. Let n_R denote the number of roots found before the failure occurred. Then $\mathbf{z}(1, n)$ and $\mathbf{z}(2, n)$ contain the real and imaginary parts of the first root found, $\mathbf{z}(1, n-1)$ and $\mathbf{z}(2, n-1)$ contain the real and imaginary parts of the second root found, $\ldots, \mathbf{z}(1, n-n_R+1)$ and $\mathbf{z}(2, n-n_R+1)$ contain the real and imaginary parts of the n_Rth root found. After the failure has occurred, the remaining $2 \times (n - n_R)$ elements of **z** contain a large negative number (equal to $-1/\big(\text{x02am}() \times \sqrt{2}\big)$).

9 Example

For this function two examples are presented. There is a single example program for nag_zeros_poly_complex (c02af), with a main program and the code to solve the two example problems given in the functions EX1 and EX2.

Example 1 (EX1)

This example finds the roots of the polynomial

$$a_0 z^5 + a_1 z^4 + a_2 z^3 + a_3 z^2 + a_4 z + a_5 = 0,$$

where $a_0 = (5.0 + 6.0i)$, $a_1 = (30.0 + 20.0i)$, $a_2 = -(0.2 + 6.0i)$, $a_3 = (50.0 + 100000.0i)$, $a_4 = -(2.0 - 40.0i)$ and $a_5 = (10.0 + 1.0i)$.

Example 2 (EX2)

This example solves the same problem as function EX1, but in addition attempts to estimate the accuracy of the computed roots using a perturbation analysis. Further details can be found in Thompson (1991).

9.1 Program Text

```
     function c02af_example

fprintf('c02af example results\n\n');

n = nag_int(5);
a = [5, 30, -0.2,     50, -2, 10;
     6, 20, -6.0, 100000, 40,  1];

ex1(n,a);

ex2(n,a);

function ex1(n, a)
  % Compute the roots of the polynomial
  [z, ifail] = c02af(a, n);

  disp('Computed roots of polynomial:');
  cz = z(1,:) + i*z(2,:);
  disp(cz');

function ex2(n, a)
  % Compute the roots of the polynomial
  [z, ifail] = c02af(a, n);

  % Estimate the accuracy of the computed roots using a perturbation analysis

  % Form the coefficients of the perturbed polynomial
  abar = a;
  epsbar = 3*x02aj;
  for j=1:2:n+1
    abar(:,j) = (1+epsbar)*abar(:,j);
  end
  for j=2:2:n+1
    abar(:,j) = (1-epsbar)*abar(:,j);
  end

  % Compute the roots of the perturbed polynomial
  [zbar, ifail] = c02af(abar, nag_int(n));

  % Perform error analysis
  % Initialise markers to 0 (unmarked)
  m = zeros(n, 1);
  r = zeros(n, 1);
  rmax = x02al;

  cz = z(1,:) + i*z(2,:);
  czbar = zbar(1,:) + i*zbar(2,:);
  for j=1:n
    deltai = rmax;
    r1 = abs(cz(j));

    % Loop over all perturbed roots (stored in zbar)
    for k=1:n
```

```
      % Compare the current unperturbed root to all unmarked perturbed roots.
      if m(k) == 0
        r2 = abs(czbar(k));
        deltac = abs(r1-r2);

        if deltac < deltai
          deltai = deltac;
          jmin = k;
        end
      end
    end

    % Mark the selected perturbed root
    m(jmin) = 1;

    % Compute the relative error
    if r1 ~= 0
      r3 = abs(czbar(jmin));
      di = min(r1,r3);
      r(j) = max(deltai/max(di,deltai/rmax),x02aj);
    end

  end

  fprintf('\nComputed Roots of Polynomial   Error Estimate\n');
  fprintf('                                (machine dependent)\n');
  for j=1:n
    if (z(2,j)<0)
      fprintf('%9.4f - %7.4fi           %8.2e\n',z(1,j),-z(2,j),r(j));
    else
      fprintf('%9.4f + %7.4fi           %8.2e\n',z(1,j),z(2,j),r(j));
    end
  end
```

9.2 Program Results

```
     c02af example results

Computed roots of polynomial:
 -24.3278 + 4.8555i
   5.2487 -22.7359i
  14.6533 +16.5689i
  -0.0069 + 0.0074i
   0.0065 - 0.0074i

Computed Roots of Polynomial   Error Estimate
                              (machine dependent)
 -24.3278 -  4.8555i           1.43e-16
   5.2487 + 22.7359i           3.05e-16
  14.6533 - 16.5689i           3.21e-16
  -0.0069 -  0.0074i           1.71e-16
   0.0065 +  0.0074i           1.11e-16
```