Jacques POITEVINEAU & Bruno LECOUTRE

STATISTICAL DISTRIBUTIONS FOR BAYESIAN EXPERIMENTAL DATA ANALYSIS

FORTRAN FUNCTIONS
1. CONTINUOUS DISTRIBUTIONS

2010

ERIS ♢♢♢ eris62.eu

PRELIMINARY

Download the free FMLIB package of David M. Smith:

`dmsmith.lmu.build/`

The FM package performs multiple-precision real, complex, and integer arithmetic. It provides the intrinsic Fortran numerical functions, as well as many special functions that are not included in the Fortran standard.

In addition to these three basic arithmetic types, multiple-precision exact rational arithmetic and interval arithmetic are also available.

One of the primary uses of the package is to study the accuracy and stability of numerical algorithms by comparing results computed with several different levels of precision.

SOME REFERENCES

Lecoutre B., Guigues J.-L., Poitevineau J. (1992) - Distribution of quadratic forms of multivariate Student variables. *Applied Statistics*, *41*, 617-627.
Lecoutre B. (1999) - Two useful distributions for Bayesian predictive procedures under normal models. *Journal of Statistical Planning and Inference*, *77*, 93-105.
Poitevineau J., Lecoutre B. (2010) - Implementing Bayesian predictive procedures: The K-prime and K-square distributions. *Computational Statistics & Data Analysis*, *54*, 723-730.

Contents

Part I
CUMULATIVE DISTRIBUTION FUNCTIONS

Chapter 1
Beta distribution

Function betacdf(x, p, q, ier)
File : Betacdf.f90

```
function betacdf( x, p, q, ier )
!----------------------------------------------------------------------
!
!    Returns the incomplete beta function
!
!    X     - Input . Value of the variable (0 <= X <= 1)     - Real
!    P     - Input . First parameter  (P > 0)                - Real
!    Q     - Input . Second parameter (Q > 0)                - Real
!    IER   - Output. Return code :                           - Integer
!                    0 = normal
!                    1 = invalid input argument
!                    3 = result out of limits
!                        (betacdf < 0 or betacdf > 1)
!
!    Uses subroutine bratio:
!    Armido DiDinato, Alfred Morris.
!    Algorithm 708: Significant Digit Computation of the
!    Incomplete Beta Function Ratios.
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
!----------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: betacdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, p, q
   integer, intent(out) :: ier

   ! Local declarations
   ! ------------------

   real(kind=8), parameter :: zero=0.0_8, one=1.0_8

   real(kind=8) :: w, w1, y

   !----------------------------------------------------------------------

   ! Test for valid input arguments

   if ( p <= zero .or. q <= zero ) then
      ier  = 1
      betacdf = -one
      return
   else if ( x < zero ) then
      ier  = 1
```

```
      betacdf = zero
      return
    else if ( x > one ) then
      ier  = 1
      betacdf = one
      return
    end if

    y = one - x
    call bratio( p, q, x, y, w, w1, ier )
    betacdf = w

    !  Final check
    if ( ier /= 0 ) then
      ier = 1
    else if ( betacdf < zero .or. betacdf > one ) then
      ier = 3
    end if

  end function betacdf
  subroutine bratio( a, b, x, y, w, w1, ierr )

  !*****************************************************************************80
  !
  !! BRATIO evaluates the incomplete beta function Ix(A,B).
  !
  !  Discussion:
  !
  !    It is assumed that X <= 1
  !    and Y = 1 - X.  BRATIO assigns W and W1 the values
  !
  !                  W  = ix(a,b)
  !                  W1 = 1 - ix(a,b)
  !
  !    ierr is a variable that reports the status of the results.
  !    if no input errors are detected then ierr is set to 0 and
  !    w and w1 are computed. otherwise, if an error is detected,
  !    then w and w1 are assigned the value 0 and ierr is set to
  !    one of the following values ...
  !
  !       ierr = 1  if a or b is negative
  !       ierr = 2  if a = b = 0
  !       ierr = 3  if x < 0 or x .gt. 1
  !       ierr = 4  if y < 0 or y .gt. 1
  !       ierr = 5  if x + y /= 1
  !       ierr = 6  if x = a = 0
  !       ierr = 7  if y = b = 0
  !
  !  Modified:
  !
  !    17 May 2007
  !
```

```
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
!  Parameters:
!
!    Input, real A, B, the parameters of the function.  A and B should
!    be nonnegative.
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8

  real(kind=8) a
  real(kind=8) a0
  real(kind=8) apser
  real(kind=8) b
  real(kind=8) b0
  real(kind=8) basym
  real(kind=8) bfrac
  real(kind=8) bpser
  real(kind=8) bup
  real(kind=8) eps
  real(kind=8) fpser
  integer ierr
  integer ierr1
  integer ind
  real(kind=8) lambda
  integer n
  real(kind=8) t
  real(kind=8) w
  real(kind=8) w1
  real(kind=8) x
  real(kind=8) x0
  real(kind=8) y
  real(kind=8) y0
  real(kind=8) z

  eps = epsilon ( one )

  w = zero
  w1 = zero
  if (a < zero .or. b < zero) go to 300
```

```
      if (a == zero .and. b == zero) go to 310
      if (x < zero .or. x .gt. one) go to 320
          if (y < zero .or. y .gt. one) go to 330

      z = ((x + y) - half) - half

      if ( abs(z) .gt. 3.0_8*eps) then
        ierr = 5
        return
      end if

      ierr = 0

      if (x == zero) go to 200
      if (y == zero) go to 210
      if (a == zero) go to 211
      if (b == zero) go to 201

          eps = max ( eps, 1.e-15_8 )
          if ( max ( a, b ) < 1.e-3_8*eps) go to 230

          ind = 0
          a0 = a
          b0 = b
          x0 = x
          y0 = y
          if ( min ( a0, b0 ) .gt. one) go to 30
  !
  !  procedure for a0 <= 1 or b0 <= 1
  !
          if (x <= half) go to 10
          ind = 1
          a0 = b
          b0 = a
          x0 = y
          y0 = x

       10 if (b0 < min ( eps, eps * a0 ) ) go to 80
          if (a0 < min ( eps, eps * b0 ) .and. b0*x0 <= one) go to 90
          if ( max ( a0, b0 ) .gt. one) go to 20
          if (a0 .ge. min ( 0.2_8, b0 ) ) go to 100
          if (x0**a0 <= 0.9_8) go to 100
          if (x0 .ge. 0.3_8) go to 110
          n = 20
          go to 130

       20 if (b0 <= one) go to 100
          if (x0 .ge. 0.3_8) go to 110
          if (x0 .ge. 0.1_8) go to 21
          if ((x0*b0)**a0 <= 0.7_8) go to 100
       21 if (b0 .gt. 15.0_8) go to 131
          n = 20
          go to 130
```

```
!
!  procedure for a0 .gt. 1 and b0 .gt. 1
!
   30 if (a .gt. b) go to 31
         lambda = a - (a + b)*x
         go to 32
   31 lambda = (a + b)*y - b

   32 if (lambda .ge. zero) go to 40
      ind = 1
      a0 = b
      b0 = a
      x0 = y
      y0 = x
      lambda = abs(lambda)

   40 if (b0 < 40.0_8 .and. b0*x0 <= 0.7_8) go to 100
      if (b0 < 40.0_8) go to 140
      if (a0 .gt. b0) go to 50
         if (a0 <= 100.0_8) go to 120
         if (lambda .gt. 0.03_8*a0) go to 120
         go to 180
   50 if (b0 <= 100.0_8) go to 120
      if (lambda .gt. 0.03_8*b0) go to 120
      go to 180
!
!  evaluation of the appropriate algorithm
!
   80 continue
      w = fpser(a0, b0, x0, eps)
      w1 = half + (half - w)
      go to 220

   90 continue
      w1 = apser(a0, b0, x0, eps)
      w = half + (half - w1)
      go to 220

  100 continue
      w = bpser(a0, b0, x0, eps)
      w1 = half + (half - w)
      go to 220

  110 continue
      w1 = bpser(b0, a0, y0, eps)
      w = half + (half - w1)
      go to 220

  120 continue
      w = bfrac(a0, b0, x0, y0, lambda, 15.0_8*eps)
      w1 = half + (half - w)
      go to 220
```

```
  130 continue
      w1 = bup(b0, a0, y0, x0, n, eps)
      b0 = b0 + n
  131 call bgrat ( b0, a0, y0, x0, w1, 15.0_8*eps, ierr1 )
      w = half + (half - w1)
      go to 220

  140 n = b0
      b0 = b0 - n

      if (b0 == zero) then
        n = n - 1
        b0 = one
      end if

  141 w = bup(b0, a0, y0, x0, n, eps)
      if (x0 .gt. 0.7_8) go to 150
      w = w + bpser(a0, b0, x0, eps)
      w1 = half + (half - w)
      go to 220

  150 if ( 15.0_8 < a0 ) go to 151
        n = 20
        w = w + bup(a0, b0, x0, y0, n, eps)
        a0 = a0 + n
  151 call bgrat ( a0, b0, x0, y0, w, 15.0_8*eps, ierr1 )
      w1 = half + (half - w)
      go to 220

  180 w = basym(a0, b0, lambda, 100.0_8*eps)
      w1 = half + (half - w)
      go to 220
!
!  termination of the procedure
!
  200 if (a == zero) go to 350
  201 w = zero
      w1 = one
      return

  210 if (b == zero) go to 360
  211 w = one
      w1 = zero
      return

  220 if (ind == 0) return
      t = w
      w = w1
      w1 = t
      return
!
!  procedure for a and b < 1.e-3*eps
```

```
!
  230 w = b/(a + b)
      w1 = a/(a + b)
      return
!
!  Error return
!
  300 ierr = 1
      return
  310 ierr = 2
      return
  320 ierr = 3
      return
  330 ierr = 4
      return

  350 ierr = 6
      return
  360 ierr = 7

  return
end subroutine bratio
function algdiv ( a, b )

!**********************************************************************80
!
!! ALGDIV computes ln(gamma(b)/gamma(a+b)) when 8 <= B.
!
!  Discussion:
!
!    In this algorithm, del(x) is the function defined by
!    ln(gamma(x)) = (x - 0.5)*ln(x) - x + 0.5*ln(2*pi) + del(x).
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=8), parameter :: half=0.5_8, one=1.0_8
```

```
  real(kind=8) a
  real(kind=8) algdiv
  real(kind=8) alnrel
  real(kind=8) b
  real(kind=8) c
  real(kind=8), parameter :: c0 = 0.833333333333333e-01_8
  real(kind=8), parameter :: c1 = -0.277777777760991e-02_8
  real(kind=8), parameter :: c2 = 0.793650666825390e-03_8
  real(kind=8), parameter :: c3 = -0.595202931351870e-03_8
  real(kind=8), parameter :: c4 = 0.837308034031215e-03_8
  real(kind=8), parameter :: c5 = -0.165322962780713e-02_8
  real(kind=8) d
  real(kind=8) h
  real(kind=8) s11
  real(kind=8) s3
  real(kind=8) s5
  real(kind=8) s7
  real(kind=8) s9
  real(kind=8) t
  real(kind=8) u
  real(kind=8) v
  real(kind=8) w
  real(kind=8) x
  real(kind=8) x2

  if ( b < a ) then
    h = b / a
    c = one / (one + h)
    x = h / (one + h)
    d = a + (b - half)
  else
    h = a / b
    c = h / ( one + h )
    x = one / ( one + h )
    d = b + ( a - half )
  end if
!
!  set sn = (1 - x**n) / ( 1 - x )
!
  x2 = x * x
  s3 = one + ( x + x2)
  s5 = one + ( x + x2 * s3 )
  s7 = one + ( x + x2 * s5 )
  s9 = one + ( x + x2 * s7 )
  s11 = one + ( x + x2 * s9 )
!
!  Set w = del(b) - del(a + b)
!
  t = (one/b)**2
  w = ((((c5*s11*t + c4*s9)*t + c3*s7)*t + c2*s5)*t + c1*s3)*t + c0
  w = w * ( c / b )
!
```

```
  !  Combine the results.
  !
    u = d * alnrel ( a / b )
    v = a*( log ( b ) - one )

    if ( v < u ) then
      algdiv = ( w - v ) - u
    else
      algdiv = ( w - u ) - v
    end if

    return
  end
  function alnrel ( a )

  !*****************************************************************************80
  !
  !! ALNREL evaluates the function ln(1 + a).
  !
  !  Modified:
  !
  !    17 May 2007
  !
  !  Author:
  !
  !    Armido DiDinato, Alfred Morris
  !
  !  Reference:
  !
  !    Armido DiDinato, Alfred Morris,
  !    Algorithm 708:
  !    Significant Digit Computation of the
  !    Incomplete Beta Function Ratios,
  !    ACM Transactions on Mathematical Software,
  !    Volume 18, Number 3, September 1993, pages 360-373.
  !
    implicit none

     real(kind=16), parameter :: onedble=1.0_16
     real(kind=8), parameter :: one=1.0_8, two=2.0_8

    real(kind=8) a
    real(kind=8) alnrel
    real(kind=8), parameter :: p1 = -0.129418923021993e+01_8
    real(kind=8), parameter :: p2 = 0.405303492862024_8
    real(kind=8), parameter :: p3 = -0.178874546012214e-01_8
    real(kind=8), parameter :: q1 = -0.162752256355323e+01_8
    real(kind=8), parameter :: q2 = 0.747811014037616_8
    real(kind=8), parameter :: q3 = -0.845104217945565e-01_8
    real(kind=8) t
    real(kind=8) t2
    real(kind=8) w
```

```
  real(kind=8) x

  if ( abs ( a ) <= 0.375_8 ) then

    t = a / (a + two)
    t2 = t * t
    w = (((p3*t2 + p2)*t2 + p1)*t2 + one) / &
        (((q3*t2 + q2)*t2 + q1)*t2 + one)
    alnrel = two * t * w

  else

    x = onedble + real ( a, kind(onedble) )
    alnrel = log ( x )

  end if

  return
end
function apser ( a, b, x, eps )

!*******************************************************************************80
!
!! APSER yields the incomplete beta ratio i(sub(1-x))(b,a) for
!     a <= min(eps,eps*b), b*x <= 1, and x <= 0.5. used when
!     a is very small. use only if above inequalities are satisfied.
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8, one=1.0_8

  real(kind=8) a
  real(kind=8) aj
  real(kind=8) apser
  real(kind=8) b
  real(kind=8) bx
```

```
  real(kind=8) c
  real(kind=8) eps
  real(kind=8), parameter :: g = 0.577215664901533_8
  real(kind=8) j
  real(kind=8) psi
  real(kind=8) s
  real(kind=8) t
  real(kind=8) tol
  real(kind=8) x

  bx = b * x
  t = x - bx

  if ( b * eps <= 2.e-2_8 ) then
    c = log(x) + psi(b) + g + t
  else
    c = log(bx) + g + t
  end if

  tol = 5.0_8*eps*abs(c)
  j = one
  s = zero

  do

    j = j + one
    t = t*(x - bx/j)
    aj = t/j
    s = s + aj

    if ( abs ( aj ) <= tol ) then
      exit
    end if

  end do

  apser = -a*(c + s)

  return
end
function basym ( a, b, lambda, eps )

!*******************************************************************************80
!
!! BASYM uses an asymptotic expansion for Ix(A,B) for large A and B.
!
!    lambda = (a + b)*y - b  and eps is the tolerance used.
!    it is assumed that lambda is nonnegative and that
!    a and b are greater than or equal to 15.
!
!  Modified:
!
!    17 May 2007
```

```
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8, two=2.0_8

  real(kind=8) a
  real(kind=8) a0(21)
  real(kind=8) b
  real(kind=8) b0(21)
  real(kind=8) bcorr
  real(kind=8) basym
  real(kind=8) bsum
  real(kind=8) c(21)
  real(kind=8) d(21)
  real(kind=8) dsum
  real(kind=8), parameter :: e0 = 1.12837916709551_8
  real(kind=8), parameter :: e1 = 0.353553390593274_8
  real(kind=8) eps
!!real(kind=8) erfc1
  real(kind=8) f
  real(kind=8) h
  real(kind=8) h2
  real(kind=8) hn
  integer i
  integer im1
  integer imj
  integer j
  real(kind=8) j0
  real(kind=8) j1
  real(kind=8) lambda
  integer m
  integer mm1
  integer mmj
  integer n
  integer np1
  integer, parameter :: num = 20
  real(kind=8) r
  real(kind=8) r0
  real(kind=8) r1
```

```
  real(kind=8) rlog1
  real(kind=8) s
  real(kind=8) sum2
  real(kind=8) t
  real(kind=8) t0
  real(kind=8) t1
  real(kind=8) u
  real(kind=8) w
  real(kind=8) w0
  real(kind=8) z
  real(kind=8) z0
  real(kind=8) z2
  real(kind=8) zn
  real(kind=8) znm1
!
!  num is the maximum value that n can take in the do loop
!  ending at statement 50. it is required that num be even.
!  the arrays a0, b0, c, d have dimension num + 1.
!
!-----------------------
!     e0 = 2/sqrt(pi)
!     e1 = 2**(-3/2)
!-----------------------

  basym = zero

  if ( a < b ) then
    h = a/b
    r0 = one/(one + h)
    r1 = (b - a)/b
    w0 = one/sqrt(a*(one + h))
  else
    h = b/a
    r0 = one/(one + h)
    r1 = (b - a)/a
    w0 = one/sqrt(b*(one + h))
  end if

  f = a*rlog1(-lambda/a) + b*rlog1(lambda/b)
  t = exp(-f)

  if ( t == zero ) then
    return
  end if

  z0 = sqrt(f)
  z = half*(z0/e1)
  z2 = f + f

  a0(1) = (two/3.0_8)*r1
  c(1) = - half*a0(1)
  d(1) = - c(1)
```

```
    j0 = (half/e0)*exp(z0*z0)*erfc(z0)
    j1 = e1
    sum2 = j0 + d(1)*w0*j1

    s = one
    h2 = h*h
    hn = one
    w = w0
    znm1 = z
    zn = z2

    do n = 2, num, 2

      hn = h2*hn
      a0(n) = two*r0*(one + h*hn)/(n + two)
      np1 = n + 1
      s = s + hn
      a0(np1) = two*r1*s/(n + 3.0_8)

      do i = n, np1

        r = -half*(i + one)
        b0(1) = r*a0(1)

        do m = 2, i

          bsum = zero
          mm1 = m - 1

          do j = 1, mm1
            mmj = m - j
            bsum = bsum + (j*r - mmj)*a0(j)*b0(mmj)
          end do

          b0(m) = r*a0(m) + bsum/m

        end do

        c(i) = b0(i)/(i + one)

        dsum = zero
        im1 = i - 1

        do j = 1, im1
          imj = i - j
          dsum = dsum + d(imj)*c(j)
        end do

        d(i) = -(dsum + c(i))

      end do

      j0 = e1*znm1 + (n - one)*j0
```

```
    j1 = e1*zn + n*j1
    znm1 = z2*znm1
    zn = z2*zn
    w = w0*w
    t0 = d(n)*w*j0
    w = w0*w
    t1 = d(np1)*w*j1
    sum2 = sum2 + (t0 + t1)

    if ((abs(t0) + abs(t1)) <= eps * sum2 ) then
      exit
    end if

  end do

  u = exp ( -bcorr(a,b) )
  basym = e0 * t * u * sum2

  return
end
function bcorr ( a0, b0 )

!*****************************************************************************80
!
!! BCORR evaluates  del(a0) + del(b0) - del(a0 + b0)  where
!     ln(gamma(a)) = (a - 0.5)*ln(a) - a + 0.5*ln(2*pi) + del(a).
!     it is assumed that a0 .ge. 8 and b0 .ge. 8.
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=8), parameter :: one=1.0_8

  real(kind=8) a
  real(kind=8) a0
  real(kind=8) b
```

```
  real(kind=8) b0
  real(kind=8) bcorr
  real(kind=8) c
  real(kind=8), parameter :: c0 = 0.833333333333333e-01_8
  real(kind=8), parameter :: c1 = -0.277777777760991e-02_8
  real(kind=8), parameter :: c2 = 0.793650666825390e-03_8
  real(kind=8), parameter :: c3 = -0.595202931351870e-03_8
  real(kind=8), parameter :: c4 = 0.837308034031215e-03_8
  real(kind=8), parameter :: c5 = -0.165322962780713e-02_8
  real(kind=8) h
  real(kind=8) s11
  real(kind=8) s3
  real(kind=8) s5
  real(kind=8) s7
  real(kind=8) s9
  real(kind=8) t
  real(kind=8) w
  real(kind=8) x
  real(kind=8) x2

  a = min ( a0, b0 )
  b = max ( a0, b0 )
  h = a/b
  c = h/(one + h)
  x = one/(one + h)
  x2 = x*x
!
!  Set sn = (1 - x**n)/(1 - x)
!
  s3 = one + (x + x2)
  s5 = one + (x + x2*s3)
  s7 = one + (x + x2*s5)
  s9 = one + (x + x2*s7)
  s11 = one + (x + x2*s9)
!
!  Set w = del(b) - del(a + b)
!
  t = (one/b)**2
  w = ((((c5*s11*t + c4*s9)*t + c3*s7)*t + c2*s5)*t + c1*s3)*t + c0
  w = w*(c/b)
!
!  Compute  del(a) + w
!
  t = (one/a)**2
  bcorr = (((((c5*t + c4)*t + c3)*t + c2)*t + c1)*t + c0)/a + w

  return
end
function betaln ( a0, b0 )

!*****************************************************************************80
!
```

```
!! BETALN evaluates the logarithm of the Beta function.
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
!  Local Parameters:
!
!    Local, real E, the value of Log ( 2 * PI ) / 2.
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8, two=2.0_8

  real(kind=8) a
  real(kind=8) a0
  real(kind=8) algdiv
  real(kind=8) alnrel
  real(kind=8) b
  real(kind=8) b0
  real(kind=8) bcorr
  real(kind=8) betaln
  real(kind=8) c
  real(kind=8), parameter :: e = 0.918938533204673_8
!!real(kind=8) dlgama
  real(kind=8) gsumln
  real(kind=8) h
  integer i
  integer n
  real(kind=8) u
  real(kind=8) v
  real(kind=8) w
  real(kind=8) z

  a = min ( a0, b0 )
  b = max ( a0, b0 )
  if (a .ge. 8.0_8) go to 60
  if (a .ge. one) go to 20
!
```

```
!  a < 1
!
   if ( b < 8.0_8 ) then
     betaln = dlgama ( a ) + ( dlgama ( b ) - dlgama ( a + b ) )
     return
   else
     betaln = dlgama ( a ) + algdiv ( a, b )
     return
   end if
!
!  procedure when 1 <= a < 8
!
    20 if (a .gt. two) go to 30
       if (b .gt. two) go to 21
          betaln = dlgama(a) + dlgama(b) - gsumln(a,b)
          return
    21 w = zero
       if (b < 8.0_8) go to 40
          betaln = dlgama(a) + algdiv(a,b)
          return
!
! reduction of a when b <= 1000
!
    30 if (b .gt. 1000.0_8) go to 50
       n = a - one
       w = one
       do i = 1,n
          a = a - one
          h = a/b
          w = w * (h/(one + h))
       end do
       w = log ( w )
       if (b < 8.0_8) go to 40
       betaln = w + dlgama(a) + algdiv(a,b)
       return
!
!  Reduction of b when b < 8
!
    40 n = b - one
       z = one
       do i = 1,n
          b = b - one
          z = z * (b/(a + b))
       end do
       betaln = w + log ( z ) + (dlgama(a) + (dlgama(b) - gsumln(a,b)))
       return
!
!  reduction of a when b .gt. 1000
!
    50 n = a - one
       w = one
       do i = 1,n
```

```
        a = a - one
        w = w * (a/(one + a/b))
      end do
      betaln = ( log ( w ) - n* log ( b ) ) + (dlgama(a) + algdiv(a,b))
      return
!
!  procedure when a .ge. 8
!
   60 w = bcorr(a,b)
      h = a/b
      c = h/(one + h)
      u = -(a - half) * log ( c )
      v = b*alnrel(h)
      if (u <= v) go to 61
        betaln = (((-half* log ( b ) + e) + w) - v) - u
        return
   61 betaln = (((-half* log ( b ) + e) + w) - u) - v

  return
end
function bfrac ( a, b, x, y, lambda, eps )

!*******************************************************************************80
!
!! BFRAC uses a continued fraction expansion for ix(a,b) when a,b .gt. 1.
!
!    it is assumed that  lambda = (a + b)*y - b.
!
!  Modified:
!
!    28 August 2004
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8, one=1.0_8, two=2.0_8

  real(kind=8) a
  real(kind=8) alpha
  real(kind=8) an
```

```
    real(kind=8) anp1
    real(kind=8) b
    real(kind=8) beta
    real(kind=8) bfrac
    real(kind=8) bn
    real(kind=8) bnp1
    real(kind=8) brcomp
    real(kind=8) c
    real(kind=8) c0
    real(kind=8) c1
    real(kind=8) e
    real(kind=8) eps
    real(kind=8) lambda
    real(kind=8) n
    real(kind=8) p
    real(kind=8) r
    real(kind=8) r0
    real(kind=8) s
    real(kind=8) t
    real(kind=8) w
    real(kind=8) x
    real(kind=8) y
    real(kind=8) yp1

    bfrac = brcomp ( a, b, x, y )

    if ( bfrac == zero ) then
      return
    end if

    c = one + lambda
    c0 = b / a
    c1 = one + one / a
    yp1 = y + one

    n = zero
    p = one
    s = a + one
    an = zero
    bn = one
    anp1 = one
    bnp1 = c / c1
    r = c1 / c
!
!  Continued fraction calculation.
!
    do

      n = n + one
      t = n / a
      w = n * ( b - n ) * x
      e = a / s
```

```
    alpha = ( p * ( p + c0 ) * e * e ) * ( w * x )
    e = ( one + t ) / ( c1 + t + t )
    beta = n + w / s + e * ( c + n * yp1 )
    p = one + t
    s = s + two
  !
  !  Update AN, BN, ANP1, and BNP1.
  !
    t = alpha * an + beta * anp1
    an = anp1
    anp1 = t
    t = alpha * bn + beta * bnp1
    bn = bnp1
    bnp1 = t
    r0 = r
    r = anp1 / bnp1

    if ( abs ( r - r0 ) <= eps * r ) then
      exit
    end if
  !
  !  Rescale AN, BN, ANP1, and BNP1.
  !
    an = an / bnp1
    bn = bn / bnp1
    anp1 = r
    bnp1 = one

  end do
  !
  !  Termination.
  !
  bfrac = bfrac * r

  return
end
subroutine bgrat ( a, b, x, y, w, eps, ierr )

!*****************************************************************************80
!
!! BGRAT uses an asymptotic expansion for Ix(a,b) when A is larger than B.
!
!  Discussion:
!
!    The result of the expansion is added to w.  It is assumed
!    that a .ge. 15 and b <= 1.  eps is the tolerance used.
!    ierr is a variable that reports the status of the results.
!
!  Modified:
!
!    17 May 2007
!
```

```
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8, two=2.0_8

  real(kind=8) a
  real(kind=8) algdiv
  real(kind=8) alnrel
  real(kind=8) b
  real(kind=8) bm1
  real(kind=8) bp2n
  real(kind=8) c(30)
  real(kind=8) cn
  real(kind=8) coef
  real(kind=8) d(30)
  real(kind=8) dj
  real(kind=8) eps
  real(kind=8) gam1
  integer i
  integer ierr
  real(kind=8) j
  real(kind=8) l
  real(kind=8) lnx
  integer n
  real(kind=8) n2
  integer nm1
  real(kind=8) nu
  real(kind=8) p
  real(kind=8) q
  real(kind=8) r
  real(kind=8) s
  real(kind=8) sum1
  real(kind=8) t
  real(kind=8) t2
  real(kind=8) u
  real(kind=8) v
  real(kind=8) w
  real(kind=8) x
  real(kind=8) y
  real(kind=8) z
```

```
      bm1 = (b - half) - half
      nu = a + half*bm1
      if (y .gt. 0.375_8) go to 10
      lnx = alnrel(-y)
      go to 11
   10 continue
      lnx = log ( x )
   11 continue
      z = -nu*lnx
      if (b*z == zero) go to 100
!
!  computation of the expansion
!  set r = exp(-z)*z**b/gamma(b)
!
          r = b*(one + gam1(b))*exp ( b * log ( z ) )
          r = r*exp(a*lnx)*exp(half*bm1*lnx)
          u = algdiv(b,a) + b* log ( nu )
          u = r*exp(-u)
          if (u == zero) go to 100
          call grat1(b,z,r,p,q,eps)

          v = 0.25_8*(one/nu)**2
          t2 = 0.25_8*lnx*lnx
          l = w/u
          j = q/r
          sum1 = j
          t = one
          cn = one
          n2 = zero
          do n = 1,30
            bp2n = b + n2
            j = (bp2n*(bp2n + one)*j + (z + bp2n + one)*t)*v
            n2 = n2 + two
            t = t*t2
            cn = cn/(n2*(n2 + one))
            c(n) = cn
            s = zero
            if (n == 1) go to 21
              nm1 = n - 1
              coef = b - n
              do i = 1,nm1
                s = s + coef*c(i)*d(n-i)
                coef = coef + b
              end do
     21     d(n) = bm1*cn + s/n
            dj = d(n)*j
            sum1 = sum1 + dj

            if ( sum1 <= zero ) then
              go to 100
            end if
```

```
          if (abs(dj) <= eps*(sum1 + l)) then
            go to 30
          end if

        end do
!
!  Add the results to w
!
    30 ierr = 0
        w = w + u * sum1
        return
!
!  the expansion cannot be computed
!
   100 ierr = 1

   return
end
function bpser ( a, b, x, eps )

!*****************************************************************************80
!
!! BPSER uses the power series expansion for evaluating ix(a,b) when b <= 1
!     or b*x <= 0.7.  eps is the tolerance used.
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
   implicit none

    real(kind=16), parameter :: onedble=1.0_16
    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8

   real(kind=8) a
   real(kind=8) a0
   real(kind=8) algdiv
   real(kind=8) apb
   real(kind=8) b
```

```
  real(kind=8) b0
  real(kind=8) betaln
  real(kind=8) bpser
  real(kind=8) c
  real(kind=8) eps
  real(kind=8) gam1
  real(kind=8) gamln1
  integer i
  integer m
  real(kind=8) n
  real(kind=8) sum1
  real(kind=8) t
  real(kind=8) tol
  real(kind=8) u
  real(kind=8) w
  real(kind=8) x
  real(kind=8) z

  bpser = zero
  if ( x == zero ) then
    return
  end if
!
!  compute the factor x**a/(a*beta(a,b))
!
      a0 = min ( a, b )
      if (a0 < one) go to 10
         z = a* log ( x ) - betaln(a,b)
         bpser = exp(z)/a
         go to 70
   10 b0 = max ( a, b )
      if (b0 .ge. 8.0_8) go to 60
      if (b0 .gt. one) go to 40
!
!  procedure for a0 < 1 and b0 <= 1
!
      bpser = x**a
      if (bpser == zero) return

      apb = a + b
      if (apb .gt. one) go to 20
         z = one + gam1(apb)
         go to 30
   20 u = real(a,kind(onedble)) + real(b,kind(onedble)) - onedble
      z = (one + gam1(u))/apb

   30 c = (one + gam1(a))*(one + gam1(b))/z
      bpser = bpser*c*(b/apb)
      go to 70
!
!  procedure for a0 < 1 and 1 < b0 < 8
!
```

```
    40 u = gamln1(a0)
       m = b0 - one
       if (m < 1) go to 50
       c = one
       do i = 1,m
          b0 = b0 - one
          c = c*(b0/(a0 + b0))
       end do
       u = log ( c ) + u

    50 z = a* log ( x ) - u
       b0 = b0 - one
       apb = a0 + b0
       if (apb .gt. one) go to 51
          t = one + gam1(apb)
          go to 52
    51 u = real(a0,kind(onedble)) + real(b0,kind(onedble)) - onedble
       t = (one + gam1(u))/apb
    52 bpser = exp(z)*(a0/a)*(one + gam1(b0))/t
       go to 70
!
!  procedure for a0 < 1 and b0 .ge. 8
!
    60 u = gamln1(a0) + algdiv(a0,b0)
       z = a* log ( x ) - u
       bpser = (a0/a)*exp(z)
    70 if (bpser == zero .or. a <= 0.1_8*eps) return
!
!  Compute the series
!
   sum1 = zero
   n = zero
   c = one
   tol = eps/a

   do
     n = n + one
     c = c*(half + (half - b/n))*x
     w = c/(a + n)
     sum1 = sum1 + w
     if ( abs ( w ) <= tol ) then
       exit
     end if
   end do

   bpser = bpser*(one + a*sum1)

   return
 end
 function brcmp1 ( mu, a, b, x, y )

 !*******************************************************************************80
```

```
!
!! BRCMP1 evaluates exp(mu) * (x**a*y**b/beta(a,b)).
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

    real(kind=16), parameter :: onedble=1.0_16
    real(kind=8), parameter :: zero=0.0_8, one=1.0_8

  real(kind=8) a
  real(kind=8) a0
  real(kind=8) algdiv
  real(kind=8) alnrel
  real(kind=8) apb
  real(kind=8) b
  real(kind=8) b0
  real(kind=8) bcorr
  real(kind=8) betaln
  real(kind=8) brcmp1
  real(kind=8) c
  real(kind=8), parameter :: const = 0.398942280401433_8
  real(kind=8) e
  real(kind=8) esum
  real(kind=8) gam1
  real(kind=8) gamln1
  real(kind=8) h
  integer i
  real(kind=8) lambda
  real(kind=8) lnx
  real(kind=8) lny
  integer mu
  integer n
  real(kind=8) rlog1
  real(kind=8) t
  real(kind=8) u
  real(kind=8) v
```

```
      real(kind=8) x
      real(kind=8) x0
      real(kind=8) y
      real(kind=8) y0
      real(kind=8) z

      a0 = min ( a, b )
         if (a0 .ge. 8.0_8) go to 100

         if (x .gt. 0.375_8) go to 10
            lnx = log ( x )
            lny = alnrel(-x)
            go to 20
     10 if (y .gt. 0.375_8) go to 11
            lnx = alnrel(-y)
            lny = log ( y )
            go to 20
     11 lnx = log ( x )
         lny = log ( y )

     20 z = a*lnx + b*lny
         if (a0 < one) go to 30
         z = z - betaln(a,b)
         brcmp1 = esum(mu,z)
         return
!-----------------------------------------------------------------------
!  procedure for a < 1 or b < 1
!-----------------------------------------------------------------------
     30 b0 = max ( a, b )
         if (b0 .ge. 8.0_8) go to 80
         if (b0 .gt. one) go to 60
!
!  algorithm for b0 <= 1
!
         brcmp1 = esum(mu,z)
         if (brcmp1 == zero) return

         apb = a + b
         if ( one < apb ) go to 40
            z = one + gam1(apb)
            go to 50
     40 u = real(a,kind(onedble)) + real(b,kind(onedble)) - onedble
         z = (one + gam1(u))/apb

     50 c = (one + gam1(a))*(one + gam1(b))/z
         brcmp1 = brcmp1*(a0*c)/(one + a0/b0)
         return
!
!  algorithm for 1 < b0 < 8
!
     60 u = gamln1(a0)
         n = b0 - one
```

```
      if (n < 1) go to 70
      c = one
      do 61 i = 1,n
         b0 = b0 - one
         c = c*(b0/(a0 + b0))
   61 continue
      u = log ( c ) + u

   70 z = z - u
      b0 = b0 - one
      apb = a0 + b0
      if (apb .gt. one) go to 71
         t = one + gam1(apb)
         go to 72
   71 u = real(a0,kind(onedble)) + real(b0,kind(onedble)) - onedble
      t = (one + gam1(u))/apb
   72 brcmp1 = a0*esum(mu,z)*(one + gam1(b0))/t
      return
!
!  algorithm for b0 .ge. 8
!
   80 u = gamln1(a0) + algdiv(a0,b0)
      brcmp1 = a0*esum(mu,z - u)
      return
!-----------------------------------------------------------------------
!  procedure for a .ge. 8 and b .ge. 8
!-----------------------------------------------------------------------
  100 if (a .gt. b) go to 101
         h = a/b
         x0 = h/(one + h)
         y0 = one/(one + h)
         lambda = a - (a + b)*x
         go to 110
  101 h = b/a
      x0 = one/(one + h)
      y0 = h/(one + h)
      lambda = (a + b)*y - b

  110 e = -lambda/a
      if (abs(e) .gt. 0.6_8) go to 111
         u = rlog1(e)
         go to 120
  111 u = e - log ( x / x0 )

  120 e = lambda/b
      if (abs(e) .gt. 0.6_8) go to 121
         v = rlog1(e)
         go to 130
  121 v = e - log ( y / y0 )

  130 z = esum(mu,-(a*u + b*v))
      brcmp1 = const*sqrt(b*x0)*z*exp(-bcorr(a,b))
```

```
  return
end
function brcomp ( a, b, x, y )

!**********************************************************************80
!
!! BRCOMP evaluates X**a * y**b / beta(a,b).
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=16), parameter :: onedble=1.0_16
   real(kind=8), parameter :: zero=0.0_8, one=1.0_8

  real(kind=8) a
  real(kind=8) a0
  real(kind=8) algdiv
  real(kind=8) alnrel
  real(kind=8) apb
  real(kind=8) b
  real(kind=8) b0
  real(kind=8) bcorr
  real(kind=8) betaln
  real(kind=8) brcomp
  real(kind=8) c
  real(kind=8), parameter :: const = 0.398942280401433_8
  real(kind=8) e
  real(kind=8) gam1
  real(kind=8) gamln1
  real(kind=8) h
  integer i
  real(kind=8) lambda
  real(kind=8) lnx
  real(kind=8) lny
  integer n
  real(kind=8) rlog1
```

```
    real(kind=8) t
    real(kind=8) u
    real(kind=8) v
    real(kind=8) x
    real(kind=8) x0
    real(kind=8) y
    real(kind=8) y0
    real(kind=8) z

    brcomp = zero

    if ( x == zero .or. y == zero ) then
      return
    end if

    a0 = min ( a, b )

        if ( 8.0 <= a0 ) go to 100

        if (x .gt. 0.375_8) go to 10
           lnx = log ( x )
           lny = alnrel(-x)
           go to 20
     10 if (y .gt. 0.375_8) go to 11
           lnx = alnrel(-y)
           lny = log ( y )
           go to 20
     11 lnx = log ( x )
        lny = log ( y )

     20 z = a*lnx + b*lny

        if ( one <= a ) then
          z = z - betaln(a,b)
          brcomp = exp(z)
          return
        end if
!-----------------------------------------------------------------------
!  procedure for a < 1 or b < 1
!-----------------------------------------------------------------------
     30 b0 = max ( a, b )
        if (b0 .ge. 8.0_8) go to 80
        if ( one < b0 ) go to 60
!
!  algorithm for b0 <= 1
!
        brcomp = exp(z)
        if (brcomp == zero) return

        apb = a + b
        if (apb .gt. one) go to 40
           z = one + gam1(apb)
           go to 50
```

```
     40 u = real(a,kind(onedble)) + real(b,kind(onedble)) - onedble
        z = (one + gam1(u))/apb

     50 c = (one + gam1(a))*(one + gam1(b))/z
        brcomp = brcomp*(a0*c)/(one + a0/b0)
        return
  !
  ! algorithm for 1 < b0 < 8
  !
     60 u = gamln1(a0)
        n = b0 - one
        if (n < 1) go to 70
        c = one
        do i = 1,n
           b0 = b0 - one
           c = c*(b0/(a0 + b0))
        end do
        u = log ( c ) + u

     70 z = z - u
        b0 = b0 - one
        apb = a0 + b0
        if (apb .gt. one) go to 71
           t = one + gam1(apb)
           go to 72
     71 u = real(a0,kind(onedble)) + real(b0,kind(onedble)) - onedble
        t = (one + gam1(u))/apb
     72 brcomp = a0*exp(z)*(one + gam1(b0))/t
        return
  !
  ! algorithm for b0 .ge. 8
  !
     80 u = gamln1(a0) + algdiv(a0,b0)
        brcomp = a0*exp(z - u)
        return
  !------------------------------------------------------------------------
  ! procedure for a .ge. 8 and b .ge. 8
  !------------------------------------------------------------------------
    100 if (a .gt. b) go to 101
           h = a/b
           x0 = h/(one + h)
           y0 = one/(one + h)
           lambda = a - (a + b)*x
           go to 110
    101 h = b/a
        x0 = one/(one + h)
        y0 = h/(one + h)
        lambda = (a + b)*y - b

    110 e = -lambda/a
        if (abs(e) .gt. 0.6_8) go to 111
           u = rlog1(e)
```

```
        go to 120
  111 u = e - log ( x / x0 )

  120 e = lambda/b
      if (abs(e) .gt. 0.6_8) go to 121
        v = rlog1(e)
        go to 130
  121 v = e - log ( y / y0 )

  130 z = exp(-(a*u + b*v))
      brcomp = const*sqrt(b*x0)*z*exp(-bcorr(a,b))

  return
end
function bup ( a, b, x, y, n, eps )

!*******************************************************************************80
!
!! BUP evaluates ix(a,b) - ix(a+n,b) where n is a positive integer.
!
!     eps is the tolerance used.
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8, one=1.0_8

  real(kind=8) a
  real(kind=8) ap1
  real(kind=8) apb
  real(kind=8) b
  real(kind=8) brcmp1
  real(kind=8) bup
  real(kind=8) d
  real(kind=8) eps
  real(kind=8) exparg
```

```
      integer i
      integer k
      integer kp1
      real(kind=8) l
      integer mu
      integer n
      integer nm1
      real(kind=8) r
      real(kind=8) t
      real(kind=8) w
      real(kind=8) x
      real(kind=8) y
!
!  obtain the scaling factor exp(-mu) and
!  exp(mu)*(x**a*y**b/beta(a,b))/a
!
      apb = a + b
      ap1 = a + one
      mu = 0
      d = one
      if (n == 1 .or. a < one) go to 10
         if (apb < 1.1_8*ap1) go to 10
            mu = abs(exparg(1))
            k = exparg(0)
            if (k < mu) mu = k
            t = mu
            d = exp(-t)

   10 continue

      bup = brcmp1(mu,a,b,x,y)/a
         if (n == 1 .or. bup == zero) return
         nm1 = n - 1
         w = d
!
!  let k be the index of the maximum term
!
         k = 0
         if (b <= one) go to 40
         if ( 1.e-4_8 < y ) go to 20
            k = nm1
            go to 30
      20 r = (b - one)*x/y - a
         if (r < one) go to 40
         k = nm1
         t = nm1
         if (r < t) k = r
!
!  add the increasing terms of the series
!
      30 do i = 1,k
            l = i - 1
```

```
        d = ((apb + l)/(ap1 + l))*x*d
        w = w + d
      end do
      if (k == nm1) go to 50
!
!  add the remaining terms of the series
!
   40 kp1 = k + 1
      do i = kp1,nm1
        l = i - 1
        d = ((apb + l)/(ap1 + l))*x*d
        w = w + d
        if (d <= eps*w) go to 50
      end do
!
!  terminate the procedure
!
   50 bup = bup*w

  return
end
function esum ( mu, x )

!*******************************************************************************80
!
!! ESUM evaluates exp(mu + x).
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8

  real(kind=8) esum
  integer mu
  real(kind=8) w
  real(kind=8) x
```

```
  if (x .gt. zero) go to 10

     if (mu < 0) go to 20
        w = mu + x
        if (w .gt. zero) go to 20
        esum = exp(w)
        return

   10 if (mu .gt. 0) go to 20
        w = mu + x
        if (w < zero) go to 20
        esum = exp(w)
        return

   20 w = mu

  esum = exp(w)*exp(x)

  return
end
function exparg ( l )

!*****************************************************************************80
!
!! EXPARG reports the largest safe arguments for EXP(X).
!
!    if l = 0 then  exparg(l) = the largest positive w for which
!    exp(w) can be computed.
!
!    if l is nonzero then  exparg(l) = the largest negative w for
!    which the computed value of exp(w) is nonzero.
!
!    Only an approximate value for exparg(l) is needed.
!
  implicit none

  real(kind=8) exparg
  integer l

   if (l == 0) then
      exparg =  706.893_8   ! log(huge(real*8)), approx.
   else
      exparg = -706.893_8   ! log(tiny(real*8)), approx.
   end if

end
function fpser ( a, b, x, eps )

!*****************************************************************************80
!
!! FPSER evaluates Ix(A,B) for small B and X <= 0.5.
!
!          for b < min(eps,eps*a) and x <= 0.5.
```

```
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8, one=1.0_8

  real(kind=8) a
  real(kind=8) an
  real(kind=8) b
  real(kind=8) c
  real(kind=8) eps
  real(kind=8) exparg
  real(kind=8) fpser
  real(kind=8) s
  real(kind=8) t
  real(kind=8) tol
  real(kind=8) x
!
!  Set FPSER = X**A.
!
  fpser = one

  if ( 1.0E-03_8 * eps < a ) then
    fpser = zero
    t = a * log ( x )
    if ( t < exparg ( 1 ) ) then
      return
    end if
    fpser = exp ( t )
  end if
!
!  Note that 1/b(a,b) = b
!
  fpser = ( b / a ) * fpser
  tol = eps/a
  an = a + one
```

```
    t = x
    s = t/an

    do

      an = an + one
      t = x * t
      c = t / an
      s = s + c

      if ( abs ( c ) <= tol ) then
        exit
      end if

    end do

    fpser = fpser * ( one + a * s )

    return
  end
  function gam1 ( a )

  !*****************************************************************************80
  !
  !! GAM1 computes 1/gamma(a+1) - 1  for -0.5 <= a <= 1.5
  !
  !  Modified:
  !
  !    17 May 2007
  !
  !  Author:
  !
  !    Armido DiDinato, Alfred Morris
  !
  !  Reference:
  !
  !    Armido DiDinato, Alfred Morris,
  !    Algorithm 708:
  !    Significant Digit Computation of the
  !    Incomplete Beta Function Ratios,
  !    ACM Transactions on Mathematical Software,
  !    Volume 18, Number 3, September 1993, pages 360-373.
  !
    implicit none

     real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8

    real(kind=8) a
    real(kind=8) bot
    real(kind=8) d
    real(kind=8) gam1
    real(kind=8) p(7)
    real(kind=8) q(5)
```

```
      real(kind=8) r(9)
      real(kind=8) s1
      real(kind=8) s2
      real(kind=8) t
      real(kind=8) top
      real(kind=8) w

          data p(1)/ .577215664901533_8/, p(2)/-.409078193005776_8/, &
              p(3)/-.230975380857675_8/, p(4)/ .597275330452234e-01_8/, &
              p(5)/ .766968181649490e-02_8/, p(6)/-.514889771323592e-02_8/, &
              p(7)/ .589597428611429e-03_8/

          data q(1)/ .100000000000000e+01_8/, q(2)/ .427569613095214_8/, &
              q(3)/ .158451672430138_8/, q(4)/ .261132021441447e-01_8/, &
              q(5)/ .423244297896961e-02_8/

          data r(1)/-.422784335098468_8/, r(2)/-.771330383816272_8/, &
              r(3)/-.244757765222226_8/, r(4)/ .118378989872749_8/, &
              r(5)/ .930357293360349e-03_8/, r(6)/-.118290993445146e-01_8/, &
              r(7)/ .223047661158249e-02_8/, r(8)/ .266505979058923e-03_8/, &
              r(9)/-.132674909766242e-03_8/

          data s1  / .273076135303957_8/, s2    / .559398236957378e-01_8/

          t = a
          d = a - half
          if (d .gt. zero) t = d - half
          if (t) 30,10,20

       10 gam1 = zero
          return

       20 top = (((((p(7)*t + p(6))*t + p(5))*t + p(4))*t + p(3))*t &
                         + p(2))*t + p(1)
          bot = (((q(5)*t + q(4))*t + q(3))*t + q(2))*t + one
          w = top/bot
          if (d .gt. zero) go to 21
             gam1 = a*w
             return
       21 gam1 = (t/a)*((w - half) - half)
          return

       30 top = (((((((r(9)*t + r(8))*t + r(7))*t + r(6))*t + r(5))*t &
                           + r(4))*t + r(3))*t + r(2))*t + r(1)
          bot = (s2*t + s1)*t + one
          w = top/bot
          if ( zero < d ) go to 31
             gam1 = a*((w + half) + half)
             return
       31 gam1 = t*w/a

       return
      end
```

```
  function gamln1 ( a )

  !*****************************************************************************80
  !
  !! GAMLN1 evaluates ln(gamma(1 + a)) for -0.2 <= A <= 1.25
  !
  !  Modified:
  !
  !    17 May 2007
  !
  !  Author:
  !
  !    Armido DiDinato, Alfred Morris
  !
  !  Reference:
  !
  !    Armido DiDinato, Alfred Morris,
  !    Algorithm 708:
  !    Significant Digit Computation of the
  !    Incomplete Beta Function Ratios,
  !    ACM Transactions on Mathematical Software,
  !    Volume 18, Number 3, September 1993, pages 360-373.
  !
    implicit none

     real(kind=8), parameter :: half=0.5_8, one=1.0_8

    real(kind=8) a
    real(kind=8) gamln1
    real(kind=8) p0
    real(kind=8) p1
    real(kind=8) p2
    real(kind=8) p3
    real(kind=8) p4
    real(kind=8) p5
    real(kind=8) p6
    real(kind=8) q1
    real(kind=8) q2
    real(kind=8) q3
    real(kind=8) q4
    real(kind=8) q5
    real(kind=8) q6
    real(kind=8) r0
    real(kind=8) r1
    real(kind=8) r2
    real(kind=8) r3
    real(kind=8) r4
    real(kind=8) r5
    real(kind=8) s1
    real(kind=8) s2
    real(kind=8) s3
    real(kind=8) s4
```

```
  real(kind=8) s5
  real(kind=8) w
  real(kind=8) x

     data p0/ .577215664901533_8/, p1/ .844203922187225_8/, &
          p2/-.168860593646662_8/, p3/-.780427615533591_8/, &
          p4/-.402055799310489_8/, p5/-.673562214325671e-01_8/, &
          p6/-.271935708322958e-02_8/
     data q1/ .288743195473681e+01_8/, q2/ .312755088914843e+01_8/, &
          q3/ .156875193295039e+01_8/, q4/ .361951990101499_8/, &
          q5/ .325038868253937e-01_8/, q6/ .667465618796164e-03_8/

     data r0/.422784335098467_8/,    r1/.848044614534529_8/, &
          r2/.565221050691933_8/,    r3/.156513060486551_8/, &
          r4/.170502484022650e-01_8/, r5/.497958207639485e-03_8/
     data s1/.124313399877507e+01_8/, s2/.548042109832463_8/, &
          s3/.101552187439830_8/,    s4/.713309612391000e-02_8/, &
          s5/.116165475989616e-03_8/

  if ( a < 0.6 ) then
     w = ((((((p6*a + p5)*a + p4)*a + p3)*a + p2)*a + p1)*a + p0)/ &
         ((((((q6*a + q5)*a + q4)*a + q3)*a + q2)*a + q1)*a + one)
     gamln1 = -a*w
  else
    x = (a - half) - half
     w = (((((r5*x + r4)*x + r3)*x + r2)*x + r1)*x + r0)/ &
         (((((s5*x + s4)*x + s3)*x + s2)*x + s1)*x + one)
     gamln1 = x*w
  end if

  return
end
subroutine grat1 ( a, x, r, p, q, eps )

!**********************************************************************80
!
!! GRAT1 evaluates the incomplete Gamma ratio functions P(A,X) and Q(A,X).
!
!  Discussion:
!
!    It is assumed that a <= 1.  eps is the tolerance to be used.
!    the input argument r has the value e**(-x)*x**a/gamma(a).
!
!  Modified:
!
!    28 August 2004
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
```

```
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
      implicit none

       real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8, two=2.0_8

      real(kind=8) a
      real(kind=8) a2n
      real(kind=8) a2nm1
      real(kind=8) am0
      real(kind=8) an
      real(kind=8) an0
      real(kind=8) b2n
      real(kind=8) b2nm1
      real(kind=8) c
      real(kind=8) cma
      real(kind=8) eps
    !!real(kind=8) erf
    !!real(kind=8) erfc1
      real(kind=8) g
      real(kind=8) gam1
      real(kind=8) h
      real(kind=8) j
      real(kind=8) l
      real(kind=8) p
      real(kind=8) q
      real(kind=8) r
      real(kind=8) rexp
      real(kind=8) sum2
      real(kind=8) t
      real(kind=8) tol
      real(kind=8) w
      real(kind=8) x
      real(kind=8) z

      if (a*x == zero) go to 130
      if (a == half) go to 120
      if (x < 1.1) go to 10
      go to 50
!
!  Taylor series for p(a,x)/x**a
!
   10 continue

         an = 3.0_8
         c = x
```

```
      sum2 = x/(a + 3.0_8)
      tol = 0.1_8*eps/(a + one)

      do
        an = an + one
        c = -c*(x/an)
        t = c/(a + an)
        sum2 = sum2 + t
        if ( abs ( t ) <= tol ) then
          exit
        end if
      end do

      j = a*x*((sum2/6.0_8 - half/(a + two))*x + one/(a + one))

      z = a * log ( x )
      h = gam1(a)
      g = one + h
      if (x < 0.25_8) go to 20
        if (a < x/2.59_8) go to 40
        go to 30
   20 if (z .gt. -0.13394_8) go to 40

   30 w = exp(z)
      p = w*g*(half + (half - j))
      q = half + (half - p)
      return

   40 l = rexp(z)
      w = half + (half + l)
      q = (w*j - l)*g - h
      if (q < zero) go to 110
      p = half + (half - q)
      return
!
! continued fraction expansion
!
   50 a2nm1 = one
      a2n = one
      b2nm1 = x
      b2n = x + (one - a)
      c = one
   51    a2nm1 = x*a2n + c*a2nm1
         b2nm1 = x*b2n + c*b2nm1
         am0 = a2nm1/b2nm1
         c = c + one
         cma = c - a
         a2n = a2nm1 + cma*a2n
         b2n = b2nm1 + cma*b2n
         an0 = a2n/b2n
         if (abs(an0 - am0) .ge. eps*an0) go to 51
      q = r*an0
```

```
      p = half + (half - q)
      return
!
!  special cases
!
  100 p = zero
      q = one
      return

  110 p = one
      q = zero
      return

  120 if (x .ge. 0.25_8) go to 121
      p = erf(sqrt(x))
      q = half + (half - p)
      return
  121 q = erfc(sqrt(x))
      p = half + (half - q)
      return

  130 if (x <= a) go to 100
      go to 110

end
function gsumln ( a, b )

!*******************************************************************************80
!
!! GSUMLN evaluates the function Log ( Gamma ( A + B ) ) in a special range.
!
!         for 1 <= a <= 2  and  1 <= b <= 2
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=16), parameter :: twodble=2.0_16
```

```
    real(kind=8), parameter :: one=1.0_8

  real(kind=8) a
  real(kind=8) alnrel
  real(kind=8) b
  real(kind=8) gamln1
  real(kind=8) gsumln
  real(kind=8) x

  x = real ( a, kind(twodble) ) + real ( b, kind(twodble) ) - twodble

  if ( x <= 0.25_8 ) then
    gsumln = gamln1 ( one + x )
  else if (x <= 1.25_8 ) then
    gsumln = gamln1 ( x ) + alnrel ( x )
  else
    gsumln = gamln1 ( x - one ) + log ( x * ( one + x ) )
  end if

  return
end
function ipmpar ( i )

!*****************************************************************************80
!
!! IPMPAR provides the integer machine constants for the computer
!     that is used. it is assumed that the argument i is an integer
!     having one of the values 1-10. ipmpar(i) has the value ...
!
!  integers.
!
!     assume integers are represented in the n-digit, base-a form
!
!               sign ( x(n-1)*a**(n-1) + ... + x(1)*a + x(0) )
!
!               where 0 <= x(i) < a for i=0,...,n-1.
!
!     ipmpar(1) = a, the base.
!
!     ipmpar(2) = n, the number of base-a digits.
!
!     ipmpar(3) = a**n - 1, the largest magnitude.
!
!  floating-point numbers.
!
!     it is assumed that the single and double precision floating
!     point arithmetics have the same base, say b, and that the
!     nonzero numbers are represented in the form
!
!               sign (b**e) * (x(1)/b + ... + x(m)/b**m)
!
!               where x(i) = 0,1,...,b-1 for i=1,...,m,
```

```
!              x(1) .ge. 1, and emin <= e <= emax.
!
!     ipmpar(4) = b, the base.
!
!  single-precision
!
!     ipmpar(5) = m, the number of base-b digits.
!
!     ipmpar(6) = emin, the smallest exponent e.
!
!     ipmpar(7) = emax, the largest exponent e.
!
!  double-precision
!
!     ipmpar(8) = m, the number of base-b digits.
!
!     ipmpar(9) = emin, the smallest exponent e.
!
!     ipmpar(10) = emax, the largest exponent e.
!
!     to define this function for the computer being used, activate
!     the data statments for the computer by removing the c from
!     column 1. (all the other data statements should have c in
!     column 1.)
!
!     ipmpar is an adaptation of the function i1mach, written by
!     p.a. fox, a.d. hall, and n.l. schryer (bell laboratories).
!     ipmpar was formed by a.h. morris (nswc). the constants are
!     from bell laboratories, nswc, and other sources.
!
      integer imach(10)
  integer ipmpar
!
!     machine constants for the ibm pc.
!
!!    data imach( 1) /     2 /
!!    data imach( 2) /    31 /
!!    data imach( 3) / 2147483647 /
!!    data imach( 4) /     2 /
!!    data imach( 5) /    24 /
!!    data imach( 6) /  -125 /
!!    data imach( 7) /   128 /
!!    data imach( 8) /    53 /
!!    data imach( 9) / -1021 /
!!    data imach(10) /  1024 /

      imach( 1) = 2
      imach( 2) = digits(0)
      imach( 3) = huge(0)
      imach( 4) = 2
      imach( 5) = digits(0.0)
      imach( 6) = minexponent(0.0)
```

```
      imach( 7) = maxexponent(0.0)
      imach( 8) = digits(0.0_8)
      imach( 9) = minexponent(0.0_8)
      imach(10) = maxexponent(0.0_8)
!
!     machine constants for amdahl machines.
!
!     data imach( 1) /   2 /
!     data imach( 2) /  31 /
!     data imach( 3) / 2147483647 /
!     data imach( 4) /  16 /
!     data imach( 5) /   6 /
!     data imach( 6) / -64 /
!     data imach( 7) /  63 /
!     data imach( 8) /  14 /
!     data imach( 9) / -64 /
!     data imach(10) /  63 /
!
!     machine constants for the at&t 3b series, at&t
!     pc 7300, and at&t 6300.
!
!     data imach( 1) /     2 /
!     data imach( 2) /    31 /
!     data imach( 3) / 2147483647 /
!     data imach( 4) /     2 /
!     data imach( 5) /    24 /
!     data imach( 6) /  -125 /
!     data imach( 7) /   128 /
!     data imach( 8) /    53 /
!     data imach( 9) / -1021 /
!     data imach(10) /  1024 /
!
!     machine constants for the burroughs 1700 system.
!
!     data imach( 1) /    2 /
!     data imach( 2) /   33 /
!     data imach( 3) / 8589934591 /
!     data imach( 4) /    2 /
!     data imach( 5) /   24 /
!     data imach( 6) / -256 /
!     data imach( 7) /  255 /
!     data imach( 8) /   60 /
!     data imach( 9) / -256 /
!     data imach(10) /  255 /
!
!     machine constants for the burroughs 5700 system.
!
!     data imach( 1) /    2 /
!     data imach( 2) /   39 /
!     data imach( 3) / 549755813887 /
!     data imach( 4) /    8 /
!     data imach( 5) /   13 /
```

```
!     data imach( 6) /  -50 /
!     data imach( 7) /   76 /
!     data imach( 8) /   26 /
!     data imach( 9) /  -50 /
!     data imach(10) /   76 /
!
!     machine constants for the burroughs 6700/7700 systems.
!
!     data imach( 1) /      2 /
!     data imach( 2) /     39 /
!     data imach( 3) / 549755813887 /
!     data imach( 4) /      8 /
!     data imach( 5) /     13 /
!     data imach( 6) /    -50 /
!     data imach( 7) /     76 /
!     data imach( 8) /     26 /
!     data imach( 9) / -32754 /
!     data imach(10) /  32780 /
!
!     machine constants for the cdc 6000/7000 series
!     60 bit arithmetic, and the cdc cyber 995 64 bit
!     arithmetic (nos operating system).
!
!     data imach( 1) /    2 /
!     data imach( 2) /   48 /
!     data imach( 3) / 281474976710655 /
!     data imach( 4) /    2 /
!     data imach( 5) /   48 /
!     data imach( 6) / -974 /
!     data imach( 7) / 1070 /
!     data imach( 8) /   95 /
!     data imach( 9) / -926 /
!     data imach(10) / 1070 /
!
!     machine constants for the cdc cyber 995 64 bit
!     arithmetic (nos/ve operating system).
!
!     data imach( 1) /     2 /
!     data imach( 2) /    63 /
!     data imach( 3) / 9223372036854775807 /
!     data imach( 4) /     2 /
!     data imach( 5) /    48 /
!     data imach( 6) / -4096 /
!     data imach( 7) /  4095 /
!     data imach( 8) /    96 /
!     data imach( 9) / -4096 /
!     data imach(10) /  4095 /
!
!     machine constants for the cray 1, xmp, 2, and 3.
!
!     data imach( 1) /     2 /
!     data imach( 2) /    63 /
```

```
!     data imach( 3) / 9223372036854775807 /
!     data imach( 4) /     2 /
!     data imach( 5) /    47 /
!     data imach( 6) / -8189 /
!     data imach( 7) /  8190 /
!     data imach( 8) /    94 /
!     data imach( 9) / -8099 /
!     data imach(10) /  8190 /
!
!     machine constants for the data general eclipse s/200.
!
!     data imach( 1) /    2 /
!     data imach( 2) /   15 /
!     data imach( 3) / 32767 /
!     data imach( 4) /   16 /
!     data imach( 5) /    6 /
!     data imach( 6) /  -64 /
!     data imach( 7) /   63 /
!     data imach( 8) /   14 /
!     data imach( 9) /  -64 /
!     data imach(10) /   63 /
!
!     machine constants for the harris 220.
!
!     data imach( 1) /    2 /
!     data imach( 2) /   23 /
!     data imach( 3) / 8388607 /
!     data imach( 4) /    2 /
!     data imach( 5) /   23 /
!     data imach( 6) / -127 /
!     data imach( 7) /  127 /
!     data imach( 8) /   38 /
!     data imach( 9) / -127 /
!     data imach(10) /  127 /
!
!     machine constants for the honeywell 600/6000
!     and dps 8/70 series.
!
!     data imach( 1) /    2 /
!     data imach( 2) /   35 /
!     data imach( 3) / 34359738367 /
!     data imach( 4) /    2 /
!     data imach( 5) /   27 /
!     data imach( 6) / -127 /
!     data imach( 7) /  127 /
!     data imach( 8) /   63 /
!     data imach( 9) / -127 /
!     data imach(10) /  127 /
!
!     machine constants for the hp 2100
!     3 word double precision option with ftn4
!
```

```
!     data imach( 1) /    2 /
!     data imach( 2) /   15 /
!     data imach( 3) / 32767 /
!     data imach( 4) /    2 /
!     data imach( 5) /   23 /
!     data imach( 6) / -128 /
!     data imach( 7) /  127 /
!     data imach( 8) /   39 /
!     data imach( 9) / -128 /
!     data imach(10) /  127 /
!
!     machine constants for the hp 2100
!     4 word double precision option with ftn4
!
!     data imach( 1) /    2 /
!     data imach( 2) /   15 /
!     data imach( 3) / 32767 /
!     data imach( 4) /    2 /
!     data imach( 5) /   23 /
!     data imach( 6) / -128 /
!     data imach( 7) /  127 /
!     data imach( 8) /   55 /
!     data imach( 9) / -128 /
!     data imach(10) /  127 /
!
!     machine constants for the hp 9000.
!
!     data imach( 1) /     2 /
!     data imach( 2) /    31 /
!     data imach( 3) / 2147483647 /
!     data imach( 4) /     2 /
!     data imach( 5) /    24 /
!     data imach( 6) /  -126 /
!     data imach( 7) /   128 /
!     data imach( 8) /    53 /
!     data imach( 9) / -1021 /
!     data imach(10) /  1024 /
!
!     machine constants for the ibm 360/370 series,
!     the icl 2900, the itel as/6, the xerox sigma
!     5/7/9 and the sel systems 85/86.
!
!     data imach( 1) /    2 /
!     data imach( 2) /   31 /
!     data imach( 3) / 2147483647 /
!     data imach( 4) /   16 /
!     data imach( 5) /    6 /
!     data imach( 6) /  -64 /
!     data imach( 7) /   63 /
!     data imach( 8) /   14 /
!     data imach( 9) /  -64 /
!     data imach(10) /   63 /
```

```
!
!     machine constants for the macintosh ii - absoft
!     macfortran ii.
!
!     data imach( 1) /     2 /
!     data imach( 2) /    31 /
!     data imach( 3) / 2147483647 /
!     data imach( 4) /     2 /
!     data imach( 5) /    24 /
!     data imach( 6) /  -125 /
!     data imach( 7) /   128 /
!     data imach( 8) /    53 /
!     data imach( 9) / -1021 /
!     data imach(10) /  1024 /
!
!     machine constants for the microvax - vms fortran.
!
!     data imach( 1) /    2 /
!     data imach( 2) /   31 /
!     data imach( 3) / 2147483647 /
!     data imach( 4) /    2 /
!     data imach( 5) /   24 /
!     data imach( 6) / -127 /
!     data imach( 7) /  127 /
!     data imach( 8) /   56 /
!     data imach( 9) / -127 /
!     data imach(10) /  127 /
!
!     machine constants for the pdp-10 (ka processor).
!
!     data imach( 1) /    2 /
!     data imach( 2) /   35 /
!     data imach( 3) / 34359738367 /
!     data imach( 4) /    2 /
!     data imach( 5) /   27 /
!     data imach( 6) / -128 /
!     data imach( 7) /  127 /
!     data imach( 8) /   54 /
!     data imach( 9) / -101 /
!     data imach(10) /  127 /
!
!     machine constants for the pdp-10 (ki processor).
!
!     data imach( 1) /    2 /
!     data imach( 2) /   35 /
!     data imach( 3) / 34359738367 /
!     data imach( 4) /    2 /
!     data imach( 5) /   27 /
!     data imach( 6) / -128 /
!     data imach( 7) /  127 /
!     data imach( 8) /   62 /
!     data imach( 9) / -128 /
```

```
!     data imach(10) /  127 /
!
!     machine constants for the pdp-11 fortran supporting
!     32-bit integer arithmetic.
!
!     data imach( 1) /    2 /
!     data imach( 2) /   31 /
!     data imach( 3) / 2147483647 /
!     data imach( 4) /    2 /
!     data imach( 5) /   24 /
!     data imach( 6) / -127 /
!     data imach( 7) /  127 /
!     data imach( 8) /   56 /
!     data imach( 9) / -127 /
!     data imach(10) /  127 /
!
!     machine constants for the sequent balance 8000.
!
!     data imach( 1) /     2 /
!     data imach( 2) /    31 /
!     data imach( 3) / 2147483647 /
!     data imach( 4) /     2 /
!     data imach( 5) /    24 /
!     data imach( 6) /  -125 /
!     data imach( 7) /   128 /
!     data imach( 8) /    53 /
!     data imach( 9) / -1021 /
!     data imach(10) /  1024 /
!
!     machine constants for the silicon graphics iris-4d
!     series (mips r3000 processor).
!
!     data imach( 1) /     2 /
!     data imach( 2) /    31 /
!     data imach( 3) / 2147483647 /
!     data imach( 4) /     2 /
!     data imach( 5) /    24 /
!     data imach( 6) /  -125 /
!     data imach( 7) /   128 /
!     data imach( 8) /    53 /
!     data imach( 9) / -1021 /
!     data imach(10) /  1024 /
!
!     machine constants for the sun 3.
!
!     data imach( 1) /     2 /
!     data imach( 2) /    31 /
!     data imach( 3) / 2147483647 /
!     data imach( 4) /     2 /
!     data imach( 5) /    24 /
!     data imach( 6) /  -125 /
!     data imach( 7) /   128 /
```

```
!     data imach( 8) /    53 /
!     data imach( 9) / -1021 /
!     data imach(10) /  1024 /
!
!     machine constants for the univac 1100 series.
!
!     data imach( 1) /    2 /
!     data imach( 2) /   35 /
!     data imach( 3) / 34359738367 /
!     data imach( 4) /    2 /
!     data imach( 5) /   27 /
!     data imach( 6) / -128 /
!     data imach( 7) /  127 /
!     data imach( 8) /   60 /
!     data imach( 9) /-1024 /
!     data imach(10) / 1023 /
!
!     machine constants for the vax 11/780.
!
!     data imach( 1) /    2 /
!     data imach( 2) /   31 /
!     data imach( 3) / 2147483647 /
!     data imach( 4) /    2 /
!     data imach( 5) /   24 /
!     data imach( 6) / -127 /
!     data imach( 7) /  127 /
!     data imach( 8) /   56 /
!     data imach( 9) / -127 /
!     data imach(10) /  127 /
!
      ipmpar = imach(i)

  return
end
function psi ( xx )

!*******************************************************************************80
!
!! PSI evaluates the Psi or Digamma function.
!
!     psi(xx) is assigned the value 0 when the digamma function cannot
!     be computed.
!
!     the main computation involves evaluation of rational chebyshev
!     approximations published in math. comp. 27, 123-127(1973) by
!     cody, strecok and thacher.
!
!     psi was written at argonne national laboratory for the funpack
!     package of special function subroutines. psi was modified by
!     a.h. morris (nswc).
!
!  Modified:
```

```
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
!  Local parameters:
!
!    Local, double precision DX0, zero of PSI.
!
!    Local, real PIOV4 = pi/4.
!
!    Local, real XMAX1, the smallest positive floating point constant
!    with entirely integer representation.  Also used as negative of
!    lower bound on acceptable negative arguments and as the positive
!    argument beyond which PSI may be represented as log(x).
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8

  real(kind=8) aug
  real(kind=8) den
  real(kind=16), parameter :: dx0 = 1.461632144968362341262659542325721325_16
  integer i
  integer ipmpar
  integer m
  integer n
  integer nq
  real(kind=8) p1(7)
  real(kind=8) p2(4)
  real(kind=8), parameter :: piov4 = 0.785398163397448_8
  real(kind=8) psi
  real(kind=8) q1(6)
  real(kind=8) q2(4)
  real(kind=8) sgn
  real(kind=8) upper
  real(kind=8) w
  real(kind=8) x
  real(kind=8) xmax1
  real(kind=8) xmx0
  real(kind=8) xsmall
```

```
      real(kind=8) xx
      real(kind=8) z
!
!  coefficients for rational approximation of
!  psi(x) / (x - x0),  0.5 <= x <= 3.0
!
          data p1(1)/.895385022981970e-02_8/, p1(2)/.477762828042627e+01_8/, &
              p1(3)/.142441585084029e+03_8/, p1(4)/.118645200713425e+04_8/, &
              p1(5)/.363351846806499e+04_8/, p1(6)/.413810161269013e+04_8/, &
              p1(7)/.130560269827897e+04_8/
          data q1(1)/.448452573429826e+02_8/, q1(2)/.520752771467162e+03_8/, &
              q1(3)/.221000799247830e+04_8/, q1(4)/.364127349079381e+04_8/, &
              q1(5)/.190831076596300e+04_8/, q1(6)/.691091682714533e-05_8/
!
!  coefficients for rational approximation of
!  psi(x) - ln(x) + 1 / (2*x),  x .gt. 3.0
!
          data p2(1)/-.212940445131011e+01_8/, p2(2)/-.701677227766759e+01_8/, &
              p2(3)/-.448616543918019e+01_8/, p2(4)/-.648157123766197_8/
          data q2(1)/ .322703493791143e+02_8/, q2(2)/ .892920700481861e+02_8/, &
              q2(3)/ .546117738103215e+02_8/, q2(4)/ .777788548522962e+01_8/
!
!  machine dependent constants ...
!
!  xsmall = absolute argument below which pi*cotan(pi*x)
!  may be represented by 1/x.
!
          xmax1 = ipmpar(3)
          xmax1 =  min ( xmax1, one / epsilon ( xmax1 ) )
          xsmall = 1.e-9_8

          x = xx
          aug = zero
          if (x .ge. half) go to 200
!
!  x < 0.5,  use reflection formula
!  psi(1-x) = psi(x) + pi * cotan(pi*x)
!
          if ( xsmall < abs(x) ) go to 100
          if (x == zero) go to 400
!
!  0 < abs(x) <= xsmall.  use 1/x as a substitute
!  for  pi*cotan(pi*x)
!
          aug = -one / x
          go to 150
!
!  reduction of argument for cotan
!
      100 w = - x
          sgn = piov4
          if ( zero < w ) go to 120
```

```
      w = - w
      sgn = -sgn
!
!  Error exit if x <= -xmax1
!
  120 if (w .ge. xmax1) go to 400
      nq = int(w)
      w = w - float(nq)
      nq = int(w*4.0_8)
      w = 4.0_8 * (w - float(nq) * .25_8)
!
!  w is now related to the fractional part of  4.0 * x.
!  adjust argument to correspond to values in first
!  quadrant and determine sign
!
      n = nq / 2
      if ((n+n) /= nq) then
        w = one - w
      end if
      z = piov4 * w
      m = n / 2
      if ((m+m) /= n) then
        sgn = - sgn
      end if
!
!  Determine final value for  -pi*cotan(pi*x)
!
      n = (nq + 1) / 2
      m = n / 2
      m = m + m
      if (m /= n) go to 140
!
!  Check for singularity
!
      if (z == zero) go to 400
!
!  use cos/sin as a substitute for cotan, and
!  sin/cos as a substitute for tan
!
      aug = sgn * ((cos(z) / sin(z)) * 4.0_8)
      go to 150
  140 aug = sgn * ((sin(z) / cos(z)) * 4.0_8)
  150 x = one - x
  200 if (x .gt. 3.0_8) go to 300
!
!  0.5 <= x <= 3.0
!
      den = x
      upper = p1(1) * x

      do i = 1, 5
         den = (den + q1(i)) * x
```

```
        upper = (upper + p1(i+1)) * x
      end do

      den = (upper + p1(7)) / (den + q1(6))
      xmx0 = real(x,kind(dx0)) - dx0
      psi = den * xmx0 + aug
      return
!
!  if x .ge. xmax1, psi = ln(x)
!
  300 if (x .ge. xmax1) go to 350
!
!  3.0 < x < xmax1
!
      w = one / (x * x)
      den = w
      upper = p2(1) * w

      do i = 1, 3
        den = (den + q2(i)) * w
        upper = (upper + p2(i+1)) * w
      end do

      aug = upper / (den + q2(4)) - half / x + aug
  350 psi = aug + log ( x )
      return
!
!  error return
!
  400 psi = zero

  return
end
function rexp ( x )

!*******************************************************************************80
!
!! REXP evaluates the function Exp(X) - 1.
!
!  Modified:
!
!    17 May 2007
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
```

```
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8

  real(kind=8), parameter :: p1 = 0.914041914819518e-09_8
  real(kind=8), parameter :: p2 = 0.238082361044469e-01_8
  real(kind=8), parameter :: q1 = -0.499999999085958_8
  real(kind=8), parameter :: q2 = 0.107141568980644_8
  real(kind=8), parameter :: q3 = -0.119041179760821e-01_8
  real(kind=8), parameter :: q4 = 0.595130811860248e-03_8
  real(kind=8) rexp
  real(kind=8) w
  real(kind=8) x

  if ( abs ( x ) <= 0.15_8 ) then

    rexp = x * ((( &
              p2    &
         * x + p1 ) &
         * x + one ) &
       / ((((  q4    &
         * x + q3 ) &
         * x + q2 ) &
         * x + q1 ) &
         * x + one ))

  else if ( x < zero ) then

    w = exp ( x )
    rexp = ( w - half ) - half

  else

    w = exp ( x )
    rexp = w * ( half + ( half - one / w ) )

  end if

  return
end
function rlog1 ( x )

!*****************************************************************************80
!
!! RLOG1 evaluates the function X - Log ( 1 + X ).
!
!  Modified:
!
!    17 May 2007
```

```
!
!  Author:
!
!    Armido DiDinato, Alfred Morris
!
!  Reference:
!
!    Armido DiDinato, Alfred Morris,
!    Algorithm 708:
!    Significant Digit Computation of the
!    Incomplete Beta Function Ratios,
!    ACM Transactions on Mathematical Software,
!    Volume 18, Number 3, September 1993, pages 360-373.
!
  implicit none

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8, two=2.0_8

  real(kind=8), parameter :: a = 0.566749439387324E-01_8
  real(kind=8), parameter :: b = 0.456512608815524E-01_8
  real(kind=8) h
  real(kind=8), parameter :: p0 = 0.333333333333333_8
  real(kind=8), parameter :: p1 = -0.224696413112536_8
  real(kind=8), parameter :: p2 = 0.620886815375787E-02_8
  real(kind=8), parameter :: q1 = -0.127408923933623E+01_8
  real(kind=8), parameter :: q2 = 0.354508718369557_8
  real(kind=8) r
  real(kind=8) rlog1
  real(kind=8) t
  real(kind=8) w
  real(kind=8) w1
  real(kind=8) x

  if ( x < -0.39_8 ) then

    w = ( x + half ) + half
    rlog1 = x - log ( w )

  else if ( x < -0.18_8 ) then

    h = ( x + 0.3_8 ) / 0.7_8
    w1 = a - h * 0.3_8
    r = h / ( h + two )
    t = r * r
    w = ( ( p2 * t + p1 ) * t + p0 ) / ( ( q2 * t + q1 ) * t + one )
    rlog1 = two * t * ( one / ( one - r ) - r * w ) + w1

  else if ( x <= 0.18_8 ) then

    h = x
    w1 = zero
    r = h / ( h + two )
    t = r * r
```

```
      w = (( p2 * t + p1 ) * t + p0 ) / (( q2 * t + q1 ) * t + one )
      rlog1 = two * t * ( one / ( one - r ) - r * w ) + w1

    else if ( x <= 0.57_8 ) then

      h = 0.75_8 * x - 0.25_8
      w1 = b + h / 3.0_8
      r = h / ( h + two )
      t = r * r
      w = (( p2 * t + p1 ) * t + p0 ) / (( q2 * t + q1 ) * t + one )
      rlog1 = two * t * ( one / ( one - r ) - r * w ) + w1

    else

      w = ( x + half ) + half
      rlog1 = x - log ( w )

    end if

    return
  end
```

Chapter 2
Beta distribution (other)

Function betacdf63(x, p, q, pql, iap, ier)
File : Betacdf63.f90

```
!------------------------------------------------------------------------
!
!    Returns the incomplete beta function
!
!    X     - Input . Value of the variable (0 <= X <= 1)     - Real
!    P     - Input . First parameter  (P > 0)                - Real
!    Q     - Input . Second parameter (Q > 0)                - Real
!    PQL   - Input . Limit for P and Q over which the Beta   - Real
!                    distribution is approximated by a normal
!                    distribution (should be at least 1000.
!                    and should not exceed the greatest
!                    positive integer)
!    IAP   - Output. 0 if the Beta has not been approximated - Integer
!                    1 otherwise
!    IER   - Output. Return code :                           - Integer
!                    0 = normal
!                    1 = invalid input argument
!                    2 = maximum number of iterations reached
!                        (then betacdf = value at last iteration,
!                                    or zero)
!                    3 = result out of limits
!                        (betacdf < -EPS or betacdf > 1+EPS)
!
!    External functions called:
!      BAPN (and DLGAMA if not in Fortran library)
!
!    Fortran functions called:
!      ABS  EXP  LOG  MIN  (and DLGAMA if available)
!
!    Uses continued fraction evaluation: Algorithm AS63 by
!    Majumder & Bhattacharjee; Applied Statistics, 1973, 22, 409-411
!    (when P=1 or Q=1, direct computations)
!
!    NOTE. When Beta is approximated the absolute error should not
!    be less than 0.001; otherwise it is supposed to be minimal.
!    Approximation according to:
!      D.B. Peizer, J.W. Pratt; J.A.S.A., 1968, 63, 1416-1456
!
!------------------------------------------------------------------------

  implicit none

  ! Function
  ! --------

  real(kind=8) :: betacdf63

  ! Arguments
  ! ---------

  real(kind=8), intent(in) :: x, p, q, pql
  integer, intent(out) :: iap, ier
```

```
  ! Local declarations
  ! ------------------

!! real(kind=8), external :: dlgama
  real(kind=8), external :: bapn

  real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
  real(kind=8), parameter :: pqmin=18.0_8
  real(kind=8), parameter :: adjerr=1.0e-3_8  ! Adjusted error
  real(kind=8), parameter :: eps=0.223e-15_8, tinyr=1.0e-307_8, explower=-706.893_8
  real(kind=8), parameter :: fpmin=1.0e-300_8
  integer, parameter :: maxit=5000

  !     pqmin = lower limit for min(p,q) such that the approximation
  !             is still satisfactory (pqmin must be > 0.5). note that
  !             18 was chosen empirically, on the basis of a few trials
  !     These constants are machine dependent:
  !     fpmin = a real nearby the smallest positive real
  !     eps   = machine epsilon
  !             (the smallest real such that 1.0 + eps > 1.0)
  !     tinyr = the smallest positive real
  !     explower = minimum valid argument for the exponential function
  !             (i.e. log(tinyr))

  real(kind=8) :: a, aa, b, bt, c, cx, d, del, erb, h, qab, qam,  &
                  qap, rx, temp, term, xx
  integer :: m, m2
  logical :: lindx

  !-----------------------------------------------------------------------

  betacdf63 = zero
  iap  = 0
  ier  = 0

  ! Test for valid input arguments

  if ( x < zero .or. x > one .or. p <= zero .or. q <= zero ) then
    ier  = 1
    betacdf63 = -one
    return
  end if

  ! x close to 0 or 1

  if ( x <= tinyr ) return
  if ( x+eps >= one ) then
    betacdf63 = one
    return
  end if

  erb = eps        ! erb is error used for final check
```

```
!  p or q close to 1

if ( abs(p-one) <= eps ) then
   if ( abs(q-one) <= eps ) then
      betacdf63 = x
      return
   end if
   a = q * log(one-x)
   betacdf63 = one
   if ( a >= explower ) betacdf63 = one - exp(a)
   goto 20
end if
if ( abs(q-one) <= eps ) then
   a = p * log(x)
   betacdf63 = zero
   if ( a >= explower ) betacdf63 = exp(a)
   goto 20
end if

!  p or q > pql : normal approximation (Peizer & Pratt, 1968)

if ( q > pql .or. p > pql ) then

   iap = 1
   erb = adjerr      ! Error used in final check is adjusted

   if ( min(p,q) >= pqmin ) then
      betacdf63 = bapn( x, p, q )
      goto 20
   end if

   !  Use recurrence to reduce to the case min(p,q) >= pqmin
   if ( p <= q ) then
      xx   = x
      cx   = one - x
      a    = p
      b    = q
      term = b*log(cx) - dlgama(b)
      rx   = log(xx)
      temp = zero
      do
         bt = term + dlgama(a+b) - dlgama(a+one) + a*rx
         if ( bt >= explower ) temp = temp + exp(bt)
         a  = a + one
         if ( a >= pqmin ) exit
      end do

      if ( x <= half ) then
         !  p < q and x =< 0.5
         betacdf63 = bapn( xx, a, b ) + temp
      else
         !  p < q and x > 0.5
```

```
      betacdf63 = one - bapn( cx, b, a ) + temp
    end if

  else
    xx   = one - x
    cx   = x
    a    = q
    b    = p
    term = b*log(cx) - dlgama(b)
    rx   = log(xx)
    temp = zero
    do
      bt = term + dlgama(a+b) - dlgama(a+one) + a*rx
      if ( bt >= explower ) temp = temp + exp(bt)
      a = a + one
      if ( a >= pqmin ) exit
    end do

    if ( x < half ) then
      !  q < p and x < 0.5
      betacdf63 = one - bapn( xx, a, b ) - temp
    else
      !  q < p and x >= 0.5
      betacdf63 = bapn( cx, b, a ) - temp
    end if

  end if
  goto 20

end if

!  General case (p and q < pql)

if ( x < (p+one)/(p+one+q+one) ) then
  lindx = .false.
  a  = p
  b  = q
  xx = x
else
  lindx = .true.
  a  = q
  b  = p
  xx = one - x
end if
qab = a + b
qap = a + one
qam = a - one
!  First step of Lentz's method
c = one
d = one - qab*xx/qap
if ( abs(d) < fpmin ) d = fpmin
d = one/d
```

```
h = d

do m = 1, maxit
   m2 = m+m
   aa = m*(b-m)*xx/((qam+m2)*(a+m2))
   !  One step of the recurrence (the even one)
   d = one + aa*d
   if ( abs(d) < fpmin ) d = fpmin
   c = one + aa/c
   if ( abs(c) < fpmin ) c = fpmin
   d = one/d
   h = h*d*c
   aa = -(a+m)*(qab+m)*xx/((a+m2)*(qap+m2))
   !  Next step of the recurrence (the odd one)
   d = one + aa*d
   if ( abs(d) < fpmin ) d = fpmin
   c = one + aa/c
   if ( abs(c) < fpmin ) c = fpmin
   d = one/d
   del = d*c
   h = h*del
   if ( abs(del-one) < eps ) goto 10
end do

!  Maximum iterations reached

ier = 2

10 continue
if ( h > zero .and. a > zero ) then
   bt = dlgama(p+q) - dlgama(p) - dlgama(q) + p*log(x) + q*log(one-x)
   bt = bt + log(h) -log(a)
   if ( bt >= explower ) then
      betacdf63 = exp(bt)
   else
      betacdf63 = zero
   end if
else
   betacdf63 = zero
end if
if ( lindx ) betacdf63 = one - betacdf63

20 continue
!  Final check, possibly adjust to 0 or 1
if ( betacdf63 < zero ) then
   if ( betacdf63 >= -erb ) then
      betacdf63 = zero
   else
      ier = 3
   end if
else if ( betacdf63 > one ) then
   if ( betacdf63 <= one+erb ) then
```

```
        betacdf63 = one
      else
        ier = 3
      end if
    end if

  end function betacdf63

  function bapn( x, p, q )

  !-----------------------------------------------------------------------
  !     - BAPN is called by BETACDF -
  !     Normal approximation of a Beta cumulative distribution function
  !     when  0 < x < 1  and  min(p,q) > 0.5
  !     see Peizer & Pratt; J.A.S.A., 1968, 63, 1416-1456
  !     Calls external function NCDF
  !-----------------------------------------------------------------------

    implicit none

    ! Function
    ! --------

    real(kind=8) :: bapn

    ! Arguments
    ! ---------

    real(kind=8), intent(in) :: x, p, q

    ! Local declarations
    ! ------------------

    real(kind=8), external :: ncdf

    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
    real(kind=8), parameter :: two=one+one, three=two+one
    real(kind=8), parameter :: third=one/three, sixth=half*third
    real(kind=8), parameter :: tiny1=1.0e-2_8, tiny2=tiny1+tiny1
    real(kind=8), parameter :: eps=1.0e-15_8

    real(kind=8) :: arg, cx, d2, gp, gq, psq, uma, xn, zx

    !-----------------------------------------------------------------------

    cx  = one - x
    psq = p + q
    xn  = psq - one
    d2  = (q - third + tiny2/q + (tiny1/psq))*x - (p - third + tiny2/p + (tiny1/psq))*cx
    arg = (p-half)/(xn*x)
    uma = one - arg
    if ( abs(uma) > eps ) then
      gp = ( one - arg*arg + two*arg*log(arg) ) / (uma*uma)
```

```
    else
      gp = zero
    end if
    arg = (q-half)/(xn*cx)
    uma = one - arg
    if ( abs(uma) > eps ) then
      gq = ( one - arg*arg + two*arg*log(arg) ) / (uma*uma)
    else
      gq = zero
    end if
    zx  = d2*sqrt( (one+ x*gq + cx*gp)/((xn+sixth)*cx*x) )

    !  Compute Proba(U<=zx) where U follows a normal(0,1) distribution
    bapn = ncdf( zx )

  end function bapn
```

Chapter 3
Noncentral Beta distribution

Function betanccdf(x, p, q, a2, delta, maxitr, ier)
File : Betanccdf.f90

```
!     function betanccdf( x, p, q, a2, delta, maxitr, ier )

!-----------------------------------------------------------------------
!
!     Calculates the probability that a random variable distributed
!     according to the non-central beta distribution with P and Q
!     degrees of freedom and A2 eccentricity parameter, is less than
!     or equal to X
!
!     X     - Input . Value of the variable          (X >= 0) - Real
!     P     - Input . First degrees of freedom       (P >  0) - Real
!     Q     - Input . Second   "    "     "          (Q >  0) - Real
!     A2    - Input . Eccentricity parameter         (A2>= 0) - Real
!     DELTA - Input . Maximum absolute error required on      - Real
!                     betanccdf (stopping criteria)
!                     (eps < delta < 1 where eps is machine
!                     epsilon; see parameter statement below)
!     MAXITR- Input . Maximum number of iterations            - Integer
!     IER   - Output. Return code :                           - Integer
!                     0 = normal
!                     1 = invalid input argument
!                        (then betanccdf = zero)
!                     2 = maximum number of iterations reached
!                        (then betanccdf = value at last iteration,
!                                 or zero)
!                     3 = required accuracy cannot be reached
!                        (then betanccdf = value at last iteration)
!                     4 = error in auxiliary function
!                        (betacdf or phi2cdf)
!                     6 = P exceeds limit (betanccdf = zero)
!
!     External functions called:
!       BETACDF PHI2CDF
!       DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!     Fortran functions called:
!       ABS  EXP  LOG  MIN  INT  (and DLGAMA if available)
!
!-----------------------------------------------------------------------

      implicit none

      ! Function
      ! --------

      real(kind=8) :: betanccdf

      ! Arguments
      ! ---------

      real(kind=8), intent(in) :: x, p, q, a2, delta
      integer, intent(in) :: maxitr
```

```
    integer, intent(out) :: ier

    ! Local declarations
    ! -----------------

  !! real(kind=8), external :: dlgama
    real(kind=8), external :: betacdf, phi2cdf

    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
    real(kind=8), parameter :: bel=1.0e6_8
    real(kind=8), parameter :: eps=0.223e-15_8, tinyr=1.0e-307_8, explower=-706.893_8
    real(kind=8), parameter :: relerr=1.0e-14_8 ! Relative error assumed
                                          ! in recurrence calculations
    real(kind=8), parameter :: xp=tinyr/relerr

    !     These constants are machine dependent:
    !     eps    = machine epsilon
    !              (the smallest real such that 1.0 + eps > 1.0)
    !     tinyr = the smallest positive real
    !     explower = minimum valid argument for the exponential function
    !              (i.e. log(tinyr))

    real(kind=8) :: a22, a22l, dj, dj1, erp, err, gl,    &
                    kgj, kgl, kx,                        &
                    qxql, rl,                            &
                    sum, sumg,                           &
                    xl
    integer :: j, jjj

    !-----------------------------------------------------------------------

    betanccdf = zero
    ier    = 0

    ! Test for valid input arguments

    if ( x < zero .or. x > one                       &
       .or. p <= zero .or. q <= zero .or. a2 < zero  &
       .or. delta >= one .or. delta <= eps ) then
       ier = 1
       return
    end if

    ! x close to 0 or 1

    if ( x <= eps ) return
    if ( x+eps >= one ) then
       betanccdf = one
       return
    end if

    ! Case a2 = 0
```

```
if ( a2 < eps ) then
  betanccdf = betacdf( x, p, q, ier )
  if ( ier /= 0 ) ier = 4
  return
end if

!  If q is large enough use limiting distribution

if ( q > bel ) then
  dj = p * p
  dj = q*x / ((dj+dj)*(one-x))
  betanccdf = phi2cdf( dj, p+p, a2, delta, maxitr, ier )
  if ( ier /= 0 ) ier = 4
  return
end if

!  If p is large that's all

if ( p > bel ) then
  ier = 6
  return
end if

!  General case (iterations)
!  kx's are decreasing for all j's
!  gl's are decreasing for j >= jjj = a2/2 - 1
!  kgj, sumg are only used for stopping rule
!  Logs are used to avoid underflows

err  = delta * half
erp  = err / relerr
a22  = a2 * half
a22l = log( a22 )
qxql = q * log(one-x) - dlgama(q)
xl   = log( x )
jjj  = int( a22 )

!  Initialize

gl = -a22
kx = betacdf( x, p, q, ier )
if ( ier /= 0 ) then
  ier = 4
  return
end if
kgj  = zero
sum  = zero
sumg = zero
dj   = zero

!  Iteration loop

do j = 0, maxitr
```

```
      if ( kx > zero ) then
        kgl = log(kx) + gl
        if ( kgl >= explower ) then
          kgl = exp( kgl )
          sum = sum + kgl
          kgj = kgj + kgl*dj
          !  Check loss of accuracy
          if ( kgj >= erp ) then
            ier = 3
            betanccdf = sum
            return
          end if
        end if
      end if

      betanccdf = sum

      !  Check accuracy (stopping rule)
      !  xp is used to prevent possible underflow

      if ( gl >= explower ) sumg = sumg + exp(gl)
      if ( kgj >= xp ) then
        if ( relerr*kgj+kx*(one-sumg) < err ) return
      else
        if (             kx*(one-sumg) < err ) return
      end if

      !  Prepare next iteration

      dj1 = j + 1
      gl  = gl + a22l - log(dj1)
      rl  = qxql + (p+dj)*xl + dlgama(p+dj+q) - dlgama(p+dj1)
      if ( rl >= explower ) kx = kx - exp( rl )
      dj = dj1

    end do

    !  Maximum number of iterations is reached

    ier = 2

  end function betanccdf
```

Chapter 4
Continued binomial distribution

Function binomccdf(x, n, p, ier)
File : binomccdf.f90

```
function binomccdf( x, n, p, ier )

!------------------------------------------------------------------------
!
!    Computes the probability that a random variable distributed
!    according to the continued binomial distribution with sample size
!    N and parameter P, is less than or equal to X. The function is
!    defined on the interval (-1; N).
!
!    X     - Input . Value of the variable (-1 <= X <= N)    - Real
!    N     - Input . Sample size (N > 0)                     - Integer
!    P     - Input . parameter of the binomial (0 <= P <= 1) - Real
!    IER   - Output. Return code :                           - Integer
!                  0 = normal
!                  1 = invalid input argument
!                  2 = maximum number of iterations reached
!                      (then binomccdf = value at last iteration,
!                                or zero)
!                  3 = result out of limits
!                      (binomccdf < 0 or binomccdf > 1)
!
!    External functions called:
!      BINOMCDF  BINOMPMF
!
!    Fortran functions called:
!      INT
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: binomccdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, p
   integer, intent(in) :: n
   integer, intent(out) :: ier

   ! Local declarations
   ! ------------------

   real(kind=8), external :: binomcdf, binompmf

   real(kind=8), parameter :: zero=0.0_8, one=1.0_8
   real(kind=8) :: t
   integer :: m

   !---------------------------------------------------------------------
```

```
  binomccdf = zero
  ier = 0

  ! Test for valid input arguments

  if ( n <= 0 .or. x < -one .or. x > n .or. p < zero .or. p > one ) then
    ier = 1
    binomccdf = -one
    return
  end if

  ! Special case

  if ( x >= n ) then
    binomccdf = one
    return
  end if
  if ( p == one ) return

  ! General case

  if ( x < zero ) then
    m = -1
    t = zero
  else
    m = int(x)
    t = binomcdf( m, n, p, ier )
    if ( ier /= 0 ) return
  end if
  binomccdf = t + (x-m)*binompmf( m+1, n, p, ier )

end function binomccdf
```

Chapter 5
Fiducial continued binomial distribution

Function binomcfcdf(x, a, b, u, ier)
File : binomcfcdf.f90

```
function binomcfcdf( x, a, b, u, ier )

!------------------------------------------------------------------------
!
!     Computes the probability that a random variable distributed
!     according to the fiducial continued binomial distribution with
!     parameters A, B, U, is less than or equal to X.
!     The function is defined on the interval (-1; N).
!
!     X      - Input . Value of the variable (0 <= X <= 1)      - Real
!     A      - Input . parameter: number of "positive" events  - Real
!                    (A > 0)
!     B      - Input . parameter: number of "negative" events  - Real
!                    (B > 0)
!     U      - Input . parameter                                - Real
!                    (0 <= U <= 1)
!     IER    - Output. Return code :                            - Integer
!                    0 = normal
!                    1 = invalid input argument
!                    2 = error in auxiliary function
!
!     External functions called:
!       BINOMCDF
!
!     Fortran functions called:
!       NINT
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: binomcfcdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, a, b, u
   integer, intent(out) :: ier

   ! Local declarations
   ! ------------------

   real(kind=8), external :: binomccdf

   real(kind=8), parameter :: zero=0.0_8, one=1.0_8
   real(kind=8), parameter :: prec=1.0e-14_8
   real(kind=8) :: t
   integer :: m

   !---------------------------------------------------------------------
```

```
  ! Test for valid input arguments

  if ( a < zero .or. b < zero .or. u < zero .or.   &
       x < zero .or. x > one ) then
    ier = 1
    binomcfcdf = -one
    return
  end if

  binomcfcdf = zero
  ier = 0

  ! Special cases

  if ( x < prec .and. a == zero ) then
    return
  else if ( x > one-prec .and. b == zero ) then
    binomcfcdf = one
    return
  end if

  ! General case

  binomcfcdf = one - binomccdf( a-u, nint(a+b), x, ier )
  if ( ier /= 0 ) ier = 2

end function binomcfcdf
```

Chapter 6
Confluent hypergeometric distribution

Function chgm(x, p, q, ier)
File : chgm.f90

```
function chgm( x, p, q, ier )

!----------------------------------------------------------------------
!
!     Computes the probability that a random variable distributed
!     according to the confluent hypergeometric distribution with
!     parameters P and Q, is less than or equal to X.
!
!     X     - Input . Value of the variable                - Real
!     P     - Input . 1st (numerator) parameter            - Real
!     Q     - Input . 2nd (denominator) parameter          - Real
!                     (Q /= 0,-1,-2,...)
!     IER   - Output. Return code :                        - Integer
!                     0 = normal
!                     1 = invalid input argument
!                        (then CHGM = 0)
!
!     External functions called:
!       DGAMMA if not in FORTRAN library
!
!     Fortran functions called:
!       ABS  COS  EXP  INT  (and DGAMMA if available)
!
!     From Zhang S. & Jin J. (1996). Computation of Special Functions.
!          New York: Wiley.
!
!     Warning.
!     In the case of a << -1 when x > 0 and a >> 1 when x < 0, the
!     evaluation involves the summation of a partially alternating
!     series which results in a loss of significant digits.
!
!----------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: chgm

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, p, q
   integer, intent(out) :: ier

   ! Local declarations
   ! ------------------

!! real(kind=8), external :: dgamma

   real(kind=8), parameter :: zero=0.0_8, one=1.0_8, two=2.0_8
   real(kind=8), parameter :: pi=3.141592653589793_8
```

```
real(kind=8), parameter :: eps=0.223e-15_8
                         ! eps = machine epsilon
                         ! (smallest real such that 1.0+eps > 1.0)
real(kind=8) :: a, a0, a1, b, hg, hg1, hg2, r, rg, r1, r2, sum1, sum2, ta, tb, tba, y0, y1, z
integer :: i, j, k, l, l0, m, nl

!-----------------------------------------------------------------------

ier = 0
hg = zero
z = x
a = p
b = q
a0 = a
a1 = a

if ( b == zero .or. b == -abs(int(b)) ) then   ! Invalid parameter
   chgm = zero
   ier = 1
   return
else if ( a == zero .or. z == zero ) then
   hg = one
else if ( a == -one ) then
   hg = one-z/b
else if ( a == b ) then
   hg = exp(z)
else if ( a-b == one ) then
   hg = (one+z/b)*exp(z)
else if ( a == one .and. b == two ) then
   hg = (exp(z)-one)/z
else if ( a < zero .and. a == int(a) ) then
   m = int(-a)
   r = one
   hg = one
   do k = 1, m
      r = r*(k-1+a)/k/(k-1+b)*z
      hg = hg + r
   end do
end if
chgm = hg
if ( hg /= zero ) return

if ( z < zero ) then
   a = b-a
   a0 = a
   z = abs(z)
end if
if ( a < two ) then
   nl = 0
else
   nl = 1
   l = int(a)
```

```
    a = a-(l+1)
  end if
  do l0 = 0, nl
    if ( a0 >= two ) a = a+one
    if ( a < zero .or. z <= 30.0_8+abs(b) ) then
      hg = one
      rg = one
      do j = 1, 500
        rg = rg*(j-1+a)/(j*(j-1+b))*z
        hg = hg + rg
        if ( abs(rg/hg) < eps ) goto 10
      end do
    else
      ta = dgamma(a)
      tb = dgamma(b)
      tba = dgamma(b-a)
      sum1 = one
      sum2 = one
      r1 = one
      r2 = one
      do i = 1, 8
        r1 = -r1*(i-1+a)*(a-b+i)/(z*i)
        r2 = -r2*(i-1+b-a)*(a-i)/(z*i)
        sum1 = sum1 + r1
        sum2 = sum2 + r2
      end do
      hg1 = tb/tba*z**(-a)*cos(pi*a)*sum1
      hg2 = tb/ta*exp(z)*z**(a-b)*sum2
      hg = hg1 + hg2
    end if
    10 continue
    if ( l0 == 0 ) y0 = hg
    if ( l0 == 1 ) y1 = hg
  end do
  if ( a0 >= two ) then
    do i = 1, l-1
      hg = ((a+a-b+z)*y1+(b-a)*y0)/a
      y0 = y1
      y1 = hg
      a = a + one
    end do
  end if
  if ( x < zero ) hg = hg*exp(x)
  chgm = hg

end function chgm
```

Chapter 7
Confluent hypergeometric distribution (other)

Function confhypg(x, p, q, maxitr, ier)
File : confhypg.f90

```
function confhypg( x, p, q, maxitr, ier )

!------------------------------------------------------------------------
!
!     Computes the probability that a random variable distributed
!     according to the confluent hypergeometric distribution with
!     parameters P and Q, is less than or equal to X.
!
!     X      - Input . Value of the variable                  - Real
!     P      - Input . 1st (numerator) parameter               - Real
!     Q      - Input . 2nd (denominator) parameter             - Real
!                    (Q /= 0,-1,-2,...)
!     MAXITR- Input . Maximum number of iterations            - Integer
!     IER    - Output. Return code :                          - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then confhypg = -1.)
!                    2 = maximum number of iterations reached
!                        (then confhypg = value at last iteration)
!
!     External functions called:
!       NONE
!
!     Fortran functions called:
!       ABS  INT
!
!------------------------------------------------------------------------

    implicit none

    ! Function
    ! --------

    real(kind=8) :: confhypg

    ! Arguments
    ! ---------

    real(kind=8), intent(in) :: x, p, q
    integer, intent(in) :: maxitr
    integer, intent(out) :: ier

    ! Local declarations
    ! ------------------

    real(kind=8), parameter :: zero=0.0_8, one=1.0_8
    real(kind=8) :: dj, s, sum
    integer :: j

    !------------------------------------------------------------------------

    ier = 0
```

```
    ! Test for valid input arguments

    if ( q == zero .or. q == -abs(int(q)) ) then
      confhypg = -one
      ier = 1
      return
    end if

    confhypg = one

    ! Case x = 0

    if ( x == zero ) return

    ! Iteration loop

    dj = zero
    s = confhypg
    sum = confhypg
    do j = 1, maxitr
      s = s*((p+dj)/(q+dj))
      dj = j
      s = s*(x/dj)
      confhypg = confhypg + s
      if ( confhypg == sum ) return    ! end of iterations
      sum = confhypg
    end do

    ! Maximum number of iterations is reached

    ier = 2

  end function confhypg
```

Chapter 8
Chi-square distribution

Function chi2cdf(x, p, plimit, ier)
File : Chi2cdf.f90

```
function chi2cdf( x, p, plimit, ier )

!------------------------------------------------------------------------
!
!     Computes the probability that a random variable distributed
!     according to the chi-square distribution with P degrees of
!     freedom, is less than or equal to X.
!
!     X     - Input . Value of the variable          (X >= 0)  - Real
!     P     - Input . Degrees of freedom             (P >  0)  - Real
!     PLIMIT- Input . Use Wilson and Hilferty's      (P >  0)  - Real
!                     approximation when P > PLIMIT
!                     (should be at least 1000)
!     IER   - Output. Return code :                            - Integer
!                     0 = normal
!                     1 = invalid input argument
!                        (then CHI2CDF = -1.)
!                     2 = maximum number of iterations reached
!                        (then CHI2CDF = value at last iteration)
!
!     External functions called:
!       NCDF standard normal distribution
!       DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!     Fortran functions called:
!       ABS  EXP  LOG  SQRT  (and DLGAMA if available)
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: chi2cdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, p, plimit
   integer, intent(out) :: ier

   ! Local declarations
   ! ------------------

!! real(kind=8), external :: dlgama
   real(kind=8), external :: ncdf

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
   real(kind=8), parameter :: two=one+one, three=two+one, nine=three*three
   real(kind=8), parameter :: xlimit=1.0e-40_8
   real(kind=8), parameter :: relerr=1.0e-14_8
   real(kind=8), parameter :: tinyr=1.0e-307_8, explower=-706.893_8
```

```
real(kind=8), parameter :: fpmin=1.0e-300_8
integer, parameter :: itmax=2000

!     relerr = relative error
!        These constants are machine dependent:
!     fpmin = a real nearby the smallest positive real
!     tinyr = the smallest positive real
!     explower = minimum valid argument for the exponential function
!              (i.e. log(tinyr))

real(kind=8) :: a, an, ap, b, c, d, del, h, y
integer :: i, n
logical :: flag

!-----------------------------------------------------------------------

! Test for valid input arguments

if ( x < zero .or. p <= zero .or. plimit <= zero ) then
   ier = 1
   chi2cdf = -one
   return
end if

ier = 0
if ( x < xlimit ) then
   chi2cdf = zero
   return
end if

if ( p <= plimit ) then
   y = x * half
   a = p * half

   if ( y < a+one ) then
      flag = .false.
      ap   = a
      h    = one/a
      del  = h
      do n = 1, itmax
         ap  = ap + one
         del = del*y/ap
         h   = h + del
         if ( abs(del) < abs(h)*relerr ) goto 10
      end do
   else
      flag = .true.
      b    = y + one - a
      c    = one/fpmin
      d    = one/b
      h    = d
      do i = 1, itmax
         an = -i*(i-a)
```

```
            b  = b + two
            d  = an*d + b
            if ( abs(d) < fpmin ) d = fpmin
            c = b + an/c
            if ( abs(c) < fpmin ) c = fpmin
            d   = one/d
            del = d*c
            h   = h*del
            if ( abs(del-one) < relerr ) goto 10
          end do
        end if
        ! Maximum number of iterations reached
        ier = 2

        10 continue
        a = - y + a*log(y) - dlgama(a)
        if ( h >= tinyr ) then
          a = log(h) + a
          if ( a >= explower ) then
            chi2cdf = exp(a)
          else
            chi2cdf = zero
          end if
        else
          chi2cdf = zero
        end if
        if ( flag ) chi2cdf = one - chi2cdf

      else
        ! Use Wilson & Hilferty's approximation when great df's
        ! Compute proba(U<=H) where U follows a normal(0,1) distr.

        b = two/(nine*p)
        a = log(x/p)/three
        if ( a >= explower ) then
          h = (exp(a)-one+b) / sqrt(b)
        else
          h = (b-one) / sqrt(b)
        end if
        chi2cdf = ncdf(h)
      end if

    end function chi2cdfend function chgm
```

Chapter 9
Noncentral chi-square distribution

Function chi2nccdf(x, p, a2, delta, maxitr, ier)
File : Chi2nccdf.f90

```
function chi2nccdf( x, p, a2, delta, maxitr, ier )

!------------------------------------------------------------------------
!
!    Calculates the probability that a random variable distributed
!    according to the noncentral chi-square distribution with P degrees of
!    freedom and A2 eccentricity parameter, is less than or equal to X
!
!    X     - Input . Value of the variable          (X >= 0) - Real
!    P     - Input . Degrees of freedom              (P >  0) - Real
!    A2    - Input . Eccentricity parameter          (A2>= 0) - Real
!    DELTA - Input . Maximum absolute error required on      - Real
!                    chi2nccdf (stopping criterion)
!                    (eps < DELTA < 1 where eps is machine
!                    epsilon; see parameter statement below)
!    MAXITR- Input . Maximum number of iterations            - Integer
!    IER   - Output. Return code :                           - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then chi2nccdf = zero)
!                    2 = maximum number of iterations reached
!                        (then chi2nccdf = value at last iteration,
!                                or zero)
!                    3 = required accuracy cannot be reached
!                        (then chi2nccdf = value at last iteration)
!                    4 = error in auxiliary function (CHI2CDF)
!
!    External functions called:
!      CHI2CDF  NORMINV
!      DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!    Fortran functions called:
!      ABS  EXP  LOG  INT  (and DLGAMA if available)
!
!    Starting index for iterations:
!    see Frick, H. (1990), Applied Statistics, 39, 311-312.
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: chi2nccdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, p, a2, delta
   integer, intent(in) :: maxitr
   integer, intent(out) :: ier
```

```
    ! local declarations
    ! -----------------

   !! real(kind=8), external :: dlgama
    real(kind=8), external :: chi2cdf, norminv

    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
    real(kind=8), parameter :: dflimit=1.0e6_8, errf0=1.0e-14_8, errfa=1.0e-6_8
    real(kind=8), parameter :: eps=0.223e-15_8, tinyr=1.0e-307_8, explower=-706.893_8
    real(kind=8), parameter :: relerr=1.0e-14_8 ! Relative error assumed
                                          ! in recurrence calculations
    real(kind=8), parameter :: xp=tinyr/relerr

    !     errf0 = relative error in chi2cdf function
    !     errfa = absolute error in chi2cdf function when approximation is used
    !     dflimit = limit for approximation in chi2cdf function
    !     These constants are machine dependent:
    !     eps   = machine epsilon
    !           (the smallest real such that 1.0 + eps > 1.0)
    !     tinyr = the smallest positive real
    !     explower = minimum valid argument for the exponential function
    !           (i.e. log(tinyr))

    real(kind=8) :: a22, a22l, dj, erp, err, err1, err2, gl,  &
                    kgj, kgl, kx,                             &
                    px2, px2l,                                &
                    p2, rl,                                  &
                    sum, sumg, xxx
    integer :: j, jjj, m

    !---------------------------------------------------------------

    chi2nccdf = zero
    ier  = 0

    ! Test for valid input arguments

    if ( x < zero .or. a2 < zero .or. p <= zero     &
       .or. delta >= one .or. delta <= eps ) then
       ier = 1
       return
    end if

    ! Case x = 0

    if ( x < eps ) return

    ! Case a2 = 0

    if ( a2 < eps ) then
       chi2nccdf = chi2cdf( x, p, dflimit, ier )
       if ( ier /= 0 ) then
          ier = 4
```

```
    else
      if (  p <= dflimit ) then
        if ( chi2nccdf*errf0 > delta ) ier = 3
      else
        if ( errfa > delta ) ier = 3
      end if
    end if
    return
  end if

  ! General case (iterations)
  ! kx's are decreasing for all j's
  ! gl's are decreasing for j > jjj = a2/2
  ! kgj, sumg are only used for stopping rule
  ! Logs are used to avoid underflows

  err1 = delta * half
  err2 = delta - err1
  p2   = p * half
  px2  = half * x
  px2l = log( px2 )
  a22  = a2 * half
  a22l = log( a22 )
  jjj  = int( a22 )
  xxx  = norminv( err1 )
  m    = int( max(a22+xxx*sqrt(a22), zero) )

  ! Initialize

  gl = m*log(a22)-(a22+dlgama(m+one))
  kx = chi2cdf( x, p+m+m, dflimit, ier )
  if ( ier /= 0 ) then
    ier = 4
    return
  end if
  ! Take account of initial error on function
  if ( p+m+m <= dflimit ) then
    err = err2 - errf0*kx
  else
    err = err2 - errfa
  end if
  if ( err <= zero ) then
    if ( gl >= explower ) chi2nccdf = exp(gl)*kx
    ier = 3
    return
  end if
  erp  = err / relerr
  rl   = (p2+m)*px2l - ( px2 + dlgama(p2+m+one) )
  kgj  = zero
  sum  = zero
  sumg = zero
  dj   = m
```

```
  ! Iteration loop

  do j = m+0, m+maxitr

    if ( kx > zero ) then
      kgl = log(kx) + gl
      if ( kgl >= explower ) then
        kgl = exp( kgl )
        sum = sum + kgl
        kgj = kgj + kgl*dj
        ! check loss of accuracy
        if ( kgj >= erp ) then
          ier = 3
          chi2nccdf = sum
          return
        end if
      end if
    end if

    chi2nccdf = sum

    ! Check accuracy (stopping rule)
    ! xp is used to prevent possible underflow

    if ( gl >= explower ) sumg = sumg + exp(gl)
    if ( kgj >= xp ) then
      if ( relerr*kgj+kx*(one-sumg) < err ) return
    else
      if (            kx*(one-sumg) < err ) return
    end if

    ! Prepare next iteration

    dj = j + 1
    gl = gl + a22l - log(dj)
    if ( rl >= explower ) kx = kx - exp( rl )
    rl = rl - log(p2+dj) + px2l

  end do

  ! Maximum number of iterations is reached

  ier = 2

end function chi2nccdf
```

Chapter 10
Noncentral chi-square distribution (other)

Function ding(x, p, a2, delta, maxitr, ier)
File : Ding.f90

```
function ding( x, p, a2, delta, maxitr, ier )

!------------------------------------------------------------------------
!
!     Calculates the probability that a random variable distributed
!     according to the noncentral chi-square distribution with P degrees of
!     freedom and A2 eccentricity parameter, is less than or equal to X
!
!     X      - Input . Value of the variable          (X >= 0) - Real
!     P      - Input . Degrees of freedom              (P >  0) - Real
!     A2     - Input . Eccentricity parameter          (A2>= 0) - Real
!     DELTA - Input . Maximum absolute error required on      - Real
!                     CHI2NC (stopping criterion)
!                     (eps < DELTA < 1 where eps is machine
!                     epsilon; see parameter statement below)
!     MAXITR- Input . Maximum number of iterations            - Integer
!     IER    - Output. Return code :                          - Integer
!                     0 = normal
!                     1 = invalid input argument
!                        (then CHI2NC = zero)
!                     2 = maximum number of iterations reached
!                        (then CHI2NC = value at last iteration,
!                                or zero)
!                     4 = error in auxiliary function (CHI2CDF)
!
!     Subroutines called:
!       NDTRI1
!
!     External functions called:
!       CHI2CDF
!       DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!     Fortran functions called:
!       ABS  EXP  LOG  INT  (and DLGAMA if available)
!
!     Basic algorithm:
!     Ding, C.G. (1992), Applied Statistics, 41, 478-482.
!
!------------------------------------------------------------------------

    implicit none

    ! Function
    ! --------

    real(kind=8) :: ding

    ! Arguments
    ! ---------

    real(kind=8), intent(in) :: x, p, a2 , delta
    integer, intent(out) :: maxitr, ier
```

```
    ! local declarations
    ! -----------------

   !! real(kind=8), external :: dlgama
    real(kind=8), external :: chi2cdf

    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
    real(kind=8), parameter :: twol=0.6931471805599453_8      ! =log(2)
    real(kind=8), parameter :: eps=0.223e-15_8, explower=-706.893_8

    !     These constants are machine dependent:
    !     eps    = machine epsilon
    !              (the smallest real such that 1.0 + eps > 1.0)
    !     explower = minimum valid argument for the exponential function

    real(kind=8) :: a22, d, eps1, eps2, g, p2, sum, sumg, w
    real :: xxx
    integer :: j, m

    !------------------------------------------------------------

    ding = zero
    ier  = 0

    ! Test for valid input arguments

    if ( x < zero .or. a2 < zero .or. p <= zero     &
       .or. delta >= one .or. delta <= eps ) then
       ier = 1
       return
    end if

    ! Case x = 0

    if ( x < eps ) return

    ! Case a2 = 0

    if ( a2 < eps ) then
       ding = chi2cdf( x, p, 1.e4_8, ier )
       if ( ier /= 0 ) ier = 4
       return
    end if

    ! General case (iterations)
    ! m is starting index for iterations (see Frick, 1990)
    ! eps1 is the error bound for the lower truncation error
    ! eps2 is the error bound for the upper truncation error

    eps1 = delta * half
    eps2 = delta - eps1
    call ndtri1( real(eps1), xxx, ier )
    a22 = a2*half
```

```
m = int( max(a22+xxx*sqrt(a22), zero) )
p2 = p * half

!  Initialize

w = chi2cdf( x, p+m+m, 1.e4_8, ier )
if ( ier /= 0 ) then
  ier = 4
  return
end if
if ( w <= zero ) return                ! since w's are decreasing
g = m*log(a22)-(a22+dlgama(m+one))
sum = g + log(w)
if ( g >= explower ) then
  g = exp(g)
else
  if ( m >= int(a22) ) return          ! since g's are decreasing
  g = zero
end if
sumg = g
if ( sum >= explower ) then
  sum = exp(sum)
else
  sum = zero
end if
d = (p2+m-one)*log(x) - ( x*half + (p2+m)*twol + dlgama(p2+m) )
if ( d >= explower ) then
  d = exp(d)
else
  d = zero
end if

!  Iteration loop

do j = m+1, m+maxitr

  g = g*(a22/j)
  d = d*(x/(j+j-2+p))
  w = w-(d+d)
  !  Check accuracy (stopping rule)
  if ( (one-sumg)*w < eps2 ) then
    ding = sum
    return
  end if
  sum = sum + g*w
  sumg = sumg + g

end do

!  Maximum number of iterations is reached

ier = 2
ding = sum
```

```
    end function ding
```

Chapter 11
Fisher-Snedecor F distribution

Function fcdf(x, p, q, ier)
File : Fcdf.f90

```
!------------------------------------------------------------------------
!
!    Computes the probability that a random variable distributed
!    according to the Fisher-Snedecor distribution with P and Q degrees
!    of freedom, is less than or equal to X.
!
!    X     - Input . Value of the variable          (X >= 0) - Real
!    P     - Input . First degrees of freedom        (P >  0) - Real
!    Q     - Input . Second   "    "     "          (Q >  0) - Real
!    IER   - Output. Return code :                          - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then fcdf = -1.)
!                    2 = maximum number of iterations reached
!                        (then fcdf = value at last iteration,
!                                     or zero)
!                    3 = result out of limits
!                        (fcdf < -eps or fcdf > 1+eps)
!
!    External functions called:
!      BETACDF
!
!    NOTE : the error on FCDF is supposed to be minimal.
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: fcdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, p, q
   integer, intent(out) :: ier

   ! local declarations
   ! ------------------

   real(kind=8), external :: betacdf

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8

   !-----------------------------------------------------------------

   ! Test for valid input arguments

   if ( x < zero .or. p <= zero .or. q <= zero ) then
      ier  = 1
      fcdf = -one
```

```
        return
    end if

    fcdf = betacdf( p*x/(p*x+q), p*half, q*half, ier )

end function fcdf
```

Chapter 12
Doubly noncentral F distribution

Function fdnccdf(x, df1, df2, alamb1, alamb2, eps, iflag)
File : fdnccdf.f90

```
function fdnccdf( x, df1, df2, alamb1, alamb2, eps, iflag )

!------------------------------------------------------------------
!
!    Computes the probability that a random variable distributed
!    according to the doubly noncentral F distribution with DF1 and
!    DF2 degrees of freedom, ALAMB1 and ALAMB2 noncentralitry
!    parameters, is less than or equal to X.
!
!    X     - Input . Value of the variable          (X >= 0) - Real
!    DF1   - Input . First degrees of freedom       (DF1> 0) - Real
!    DF2   - Input . Second   "    "     "          (DF2> 0) - Real
!    ALAMB1- Input . Noncentrality parameter for (ALAMB1>=0) - Real
!                    the numerator
!    ALAMB2- Input . Noncentrality parameter for (ALAMB2>=0) - Real
!                    the denominator
!    EPS   - Input . Desired absolute accuracy  (EPS>=1E-10) - Real
!    IFLAG - Output. Return code :                           - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then fdnccdf = -1.)
!                    2 = error in auxiliary function betacdf
!                        (then fdnccdf = value at last iteration,
!                                or zero)
!                    3 = vector dimensions too small
!                        (increase nx)
!
!    This is a simple Fortran90 adaptation of subroutine CDFDNF by
!    C.P. Reeve:
!    Reeve, C.P. (1986). An algorithm for computing the doubly
!    noncentral F c.d.f. to a specified accuracy.
!    Statistical Engineering Division, note 86-4, november.
!
!------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: fdnccdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, df1, df2, alamb1, alamb2, eps
   integer, intent(out) :: iflag

   ! Local declarations
   ! ------------------

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
   integer, parameter :: nx=10000
```

```
real(kind=8) :: cdfx, fa, fb, fc, ga, gb, gc, xx, yy
real(kind=8) :: bfi(1:nx), bfj(1:nx), poi(1:nx), poj(1:nx)
integer :: i, imin, j, jmin, ni, nj

!-----------------------------------------------------------------

! Test for valid input arguments

if ( x < zero .or. df1 <= zero .or. df2 <= zero .or.   &
     alamb1 < zero .or. alamb2 < zero .or.             &
     eps < 1.e-10_8 .or. eps >= one ) then
   iflag = 1
   fdnccdf = -one
   return
end if

fdnccdf = zero
iflag = 0

fa = half*alamb1
ga = half*alamb2
fb = half*df1
gb = half*df2
yy = df2/(df2+df1*x)
if ( yy >= one ) return
xx = one - yy
if ( xx >= one ) then
   fdnccdf = one
   return
end if
cdfx = zero

! Compute Poisson probabilities in vectors poi and poj

call poissf( fa, eps, imin, ni, poi(:nx), nx, iflag )
if ( iflag /= 0 ) return
fc = fb + imin
call poissf( ga, eps, jmin, nj, poj(:nx), nx, iflag )
if ( iflag /= 0 ) return
gc = gb + jmin

! Compute Beta cdf by recurrence when i=imin and j=jmin to jmax

call edgef( nj, gc, fc, yy, xx, bfj(1:nj), cdfx, poj(1:nj), poi(1:1), iflag, 1 )
if ( ni <= 1 .or. iflag /= 0 ) then
   fdnccdf = cdfx
   return
end if

! Compute Beta cdf by recurrence when j=jmin and i=imin to imax

bfi(1) = bfj(1)
call edgef( ni, fc, gc, xx, yy, bfi(1:ni), cdfx, poi(1:ni), poj(1:1), iflag, 2 )
```

```
    if ( nj <= 1 .or. iflag /= 0 ) then
      fdnccdf = cdfx
      return
    end if

    !  Compute Beta cdf by recurrence when i>imin and j>jmin

    do i = 2, ni
      bfj(1) = bfi(i)
      do j = 2, nj
        bfj(j) = xx*bfj(j) + yy*bfj(j-1)
        cdfx = cdfx + poi(i)*poj(j)*bfj(j)
      end do
    end do

    fdnccdf = cdfx

  end function fdnccdf
  subroutine poissf( alamb, eps, l, nspan, v, nv, iflag )
    ! Compute the Poisson(alamb) probabilities over the range (l,k)
    ! where the total tail probability is less than eps/2, sum the
    ! probabilities and shift them to the beginning of vector v.

    implicit none

    !  Arguments
    real(kind=8) :: alamb, eps
    integer :: l, nspan, nv, iflag
    real(kind=8) :: v(1:nv)

    !  Local declarations
    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
    real(kind=8) :: dal, dk, dlimit, dsum, pk, pl
    integer :: i, inc, k, nk, nl

    dlimit = one-half*eps
    k = int(alamb)
    l = k+1
    if ( alamb == zero ) then
      pl = one
    else
      dal = alamb
      dk = k
      pl = exp(dk*log(dal)-dal-dlgama(k+one))
    end if
    pk = alamb*pl/l
    nk = nv/2
    nl = nk+1
    dsum = zero
    do
      if ( pl < pk ) then
        nk = nk+1
        if ( nk > nv ) then
```

```
            iflag = 3
            return
          end if
          v(nk) = pk
          dsum =dsum + pk
          k = k+1
          if ( dsum >= dlimit ) exit
          pk = alamb*pk/(k+1)
        else
          nl = nl-1
          v(nl) = pl
          dsum = dsum + pl
          l = l-1
          if ( dsum >= dlimit ) exit
          pl = l*pl/alamb
        endif
      end do
      inc = nl-1
      do i = nl, nk
        v(i-inc) = v(i)
      end do
      nspan = nk-inc

    end subroutine poissf

    subroutine edgef( nk, fc, gc, xx, yy, bfk, cdfx, poi, poj, iflag, l )
      ! Compute the Beta cdf's by a recurrence relation along the edges
      ! i=imin and j=jmin of a grid. The corresponding components of
      ! the F'' cdf are included in the summation. Terms which might
      ! cause underflow are set to zero.

      implicit none

      !  Arguments
      real(kind=8) :: fc, gc, xx, yy, cdfx
      integer :: nk, iflag, l
      real(kind=8) :: bfk(1:nk), poi(1:nk), poj(1:1)

      !  Local declarations
      real(kind=8), external :: betacdf
      real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
      real(kind=8), parameter :: deuflo=-706.893_8
      real(kind=8) :: darg, di, dk, fd, fk
      integer :: i, j, k, kflag

      fd = fc-one
      k = max(l,min(nk,int((gc-one)*xx/yy-fd)))
      fk = fd+k
      bfk(k) = betacdf( xx,fk,gc,iflag )
      if ( iflag /= 0 ) then
        iflag = 2
        return
```

```
    end if
    if ( l == 1 ) bfk(k) = one-bfk(k)
    if ( nk /= 1 ) then
      darg = fk*log(xx)+gc*log(yy)-log(fk)+dlgama(fk+gc)-dlgama(fk)-dlgama(gc)
      if ( darg < deuflo ) then
        dk = zero
      else
        dk = exp(darg)*(-one)**l
      end if
      if ( k < nk ) then
        bfk(k+1) = bfk(k)-dk
        di = dk
        kflag = 1
        do i = k+1, nk-1
          if ( kflag == 1 ) then
            di = di*(fd+gc+(i-1))*xx/(fd+i)
            if ( dk+di == dk ) then
              kflag = 0
              di = zero
            end if
          end if
          bfk(i+1) = bfk(i)-di
        end do
      end if
      di = dk
      kflag = 1
      do i = k-1, l, -1
        if ( kflag == 1 ) then
          di = di*(fc+i)/((fd+gc+i)*xx)
          if ( dk+di == dk ) then
            kflag = 0
            di = zero
          end if
        end if
        bfk(i) = bfk(i+1)+di
      end do
    end if
    do i = l, nk
      cdfx = cdfx + poi(i)*poj(1)*bfk(i)
    end do

  end subroutine edgef
```

Chapter 13
Gamma distribution

Function gammai(x, p, plimit, ier)
File : GammaIcdf.F90

```
function gammai( x, p, plimit, ier )

!------------------------------------------------------------------
!
!    Computes the probability that a random variable distributed
!    according to the gamma distribution with P degrees of
!    freedom, is less than or equal to X.
!
!    X     - Input . Value of the variable          (X >= 0)  - Real
!    P     - Input . Degrees of freedom             (P >  0)  - Real
!    PLIMIT- Input . Use Wilson and Hilferty's      (P >  0)  - Real
!                    approximation when P > PLIMIT
!                    (should be at least 500)
!    IER   - Output. Return code :                            - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then gammai = -1.)
!                    2 = maximum number of iterations reached
!                        (then gammai = value at last iteration)
!
!    External functions called:
!      NCDF standard normal distribution
!      DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!    Fortran functions called:
!      ABS  EXP  LOG  SQRT  (and DLGAMA if available)
!
!------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: gammai

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, p, plimit
   integer, intent(out) :: ier

   ! Local declarations
   ! ------------------

!! real(kind=8), external :: dlgama
   real(kind=8), external :: ncdf

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
   real(kind=8), parameter :: two=2.0_8, three=3.0_8, nine=9.0_8
   real(kind=8), parameter :: xlimit=0.5e-40_8
   real(kind=8), parameter :: epsr=1.0e-14_8
   real(kind=8), parameter :: tinyr=1.0e-307_8, explower=-706.893_8
```

```
real(kind=8), parameter :: fpmin=1.0e-300_8
integer, parameter :: itmax=2000

!     epsr  = relative accuracy
!        These constants are machine dependent:
!     fpmin = a real nearby the smallest positive real
!     tinyr = the smallest positive real
!     explower = minimum valid argument for the exponential function
!              (i.e. log(tinyr))

real(kind=8) :: a, an, ap, b, c, d, del, h, y
integer :: i, n
logical :: flag

!-----------------------------------------------------------------------

! Test for valid input arguments

if ( x < zero .or. p <= zero .or. plimit <= zero ) then
  ier = 1
  gammai = -one
  return
end if

ier = 0
if ( x < xlimit ) then
  gammai = zero
  return
end if

if ( p <= plimit ) then
  y = x
  a = p

  if ( y < a+one ) then
    flag = .false.
    ap   = a
    h    = one/a
    del  = h
    do n = 1, itmax
      ap  = ap + one
      del = del*y/ap
      h   = h + del
      if ( abs(del) < abs(h)*epsr ) goto 10
    end do
  else
    flag = .true.
    b    = y + one - a
    c    = one/fpmin
    d    = one/b
    h    = d
    do i = 1, itmax
      an = -i*(i-a)
```

```
            b  = b + two
            d  = an*d + b
            if ( abs(d) < fpmin ) d = fpmin
            c = b + an/c
            if ( abs(c) < fpmin ) c = fpmin
            d  = one/d
            del = d*c
            h  = h*del
            if ( abs(del-one) < epsr ) goto 10
          end do
        end if
        ! Maximum number of iterations reached
        ier = 2

        10 continue
        a = - y + a*log(y) - dlgama(a)
        if ( h >= tinyr ) then
          a = log(h) + a
          if ( a >= explower ) then
            gammai = exp(a)
          else
            gammai = zero
          end if
        else
          gammai = zero
        end if
        if ( flag ) gammai = one - gammai

      else
        ! Use Wilson & Hilferty's approximation when great df's
        ! Compute proba(U<=H) where U follows a normal(0,1) distr.

        b = one/(nine*p)
        a = log(x/p)/three
        if ( a >= explower ) then
          h = (exp(a)-one+b) / sqrt(b)
        else
          h = (b-one) / sqrt(b)
        end if
        gammai = ncdf(h)
      end if

    end function gammai
```

Chapter 14
K-prime distribution

Function kprimecdf(x, q, r, a1, delta, maxitr, ier)
File : Kprimecdf.F90

```
function kprimecdf( x, q, r, a1, delta, maxitr, ier )

!------------------------------------------------------------------------
!
!     Calculates the probability that a random variable distributed
!     according to the K' distribution with Q and R degrees of
!     freedom, A1 centrality parameter, is less than or equal to X
!
!     X     - Input . Value of the variable                  - Real
!     Q     - Input . First degrees of freedom       (Q >  0) - Real
!     R     - Input . Second   "    "     "          (R >  0) - Real
!     A1    - Input . Eccentricity parameter                 - Real
!     DELTA - Input . Maximum absolute error required on     - Real
!                     kprimecdf (stopping criteria)
!                     (eps < DELTA < 1 where eps is machine
!                     epsilon; see parameter statement below)
!     MAXITR- Input . Maximum number of iterations           - Integer
!     IER   - Output. Return code :                          - Integer
!                     0 = normal
!                    -1 = no more evolution of the sum but
!                         required accuracy not reached yet
!                         (then kprimecdf = value at last iteration)
!                     1 = invalid input argument
!                         (then kprimecdf = zero)
!                     2 = maximum number of iterations reached
!                         (then kprimecdf = value at last iteration)
!                     3 = cannot be computed
!                         (then kprimecdf = zero)
!                     4 = error in auxiliary function
!                         (betacdf, lprimecdf or tcdf)
!                     5 = result out of limits (i.e. <0 or >1)
!                     7 = 2 + 5 both codes apply
!
!     External functions called:
!       BETACDF  LPRIMECDF  TCDF
!       DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!     Fortran functions called:
!       ABS  EXP  LOG  MAX  MOD  INT  (and DLGAMA if available)
!
!     Uses "method 2", see Benton & Krishnamoorthy (2003),
!     Computational Statistics & Data Analysis, 43, 249-267.
!     N.B. The mode is taken as the starting point for iterations
!     (forward and backward).
!     Starting index is modified if worthwile (see parameter betaratio)
!     To deal with the case where k=0 and r<=1, xgamf is updated at the
!     end of the iteration loop rather than at the beginning.
!
!------------------------------------------------------------------------

   implicit none
```

```
    ! Function
    ! --------

    real(kind=8) :: kprimecdf

    ! Arguments
    ! ---------

    real(kind=8), intent(in) :: x, q, r, a1, delta
    integer, intent(in) :: maxitr
    integer, intent(out) :: ier

    ! local declarations
    ! ------------------

  !! real(kind=8), external :: dlgama
    real(kind=8), external :: betacdf, lprimecdf, tcdf

    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8, two=2.0_8
    real(kind=8), parameter :: dlg15=-0.120782237635245204_8  ! =log(gamma(1.5))
    real(kind=8), parameter :: bel=1.0e6_8, qlimit=2.0e6_8, rlimit=2.0e6_8
    real(kind=8), parameter :: eps=0.223e-15_8, explower=-706.893_8
    real(kind=8), parameter :: betaratio=0.01_8
    integer, parameter :: kmin=10    ! When starting point of iterations is below
                                     ! this limit "method 2" is not judged worthwile

    !    These constants are machine dependent:
    !    eps = machine epsilon
    !          (the smallest real such that 1.0 + eps > 1.0)
    !    explower = minimum valid argument for the exponential function

    real(kind=8) :: a2, aqa, aqal, beta0, betak, dj2, q2, q2l, qqal
    real(kind=8) :: r2, r2l, r2l1mx, sumg, xarg, xl, xx
    integer :: iok, j, jm, k, kit
    logical :: xneg
    real(kind=8) :: betab(0:1), betaf(0:1), gcoefb(0:1), gcoeff(0:1)
    real(kind=8) :: cdf(0:1), xgamb(0:1), xgamf(0:1)
    real(kind=8) :: prv(0:1)

    !---------------------------------------------------------------

    kprimecdf = zero
    ier = 0

    ! Test for valid input arguments

    if ( q <= zero .or. r <= zero .or. delta >= one .or. delta <= eps ) then
      ier = 1
      return
    end if

    ! Case x = 0
```

```
if ( abs(x) < eps ) then
   cdf(0) = zero
   cdf(1) = zero
   xneg = .false.
   goto 10
end if

!  If a1 is close to zero use approximation (exact if a1 = 0)

if ( abs(a1) < eps ) then
   kprimecdf = tcdf( x, r, bel, ier )
   if ( ier /= 0 ) ier = 4
   return
end if

!  If q or r is large enough use limiting distribution

if ( q > qlimit ) then
   kprimecdf = one - lprimecdf( a1, r, x, delta, maxitr, ier ) ! noncentral t
   if ( ier /= 0 ) ier = 4
   return
else if ( r > rlimit ) then
   kprimecdf = lprimecdf( x, q, a1, delta, maxitr, ier )
   if ( ier /= 0 ) ier = 4
   return
end if

!  Define usefull parameters

xx = x
xarg = x*x
xarg = xarg / (xarg+r)
q2 = q*half
r2 = r*half
a2 = a1*a1

!  Case xarg close to one (x tends to +/- infinity)

if ( abs(xarg-one) < eps+eps ) then
   if ( x > zero ) kprimecdf = one
   return
end if

!  Case a1 < 0 : change sign of x :
!  Pr( K'(a1,1) < x ) = 1 - Pr( K'(-a1,1) < -x )

if ( a1 < zero ) xx = -xx
xneg = xx < zero          ! When a1 and x are not of same sign, the series is alternate
                          ! and this is flagged by xneg

!  General case (iterations)

!  When a1 > 0:
```

```
!     1) if xx > 0:
!        Pr( K'<xx ) =  Pr( K'<0 ) + Pr( 0<K'<xx )
!     2) if xx < 0:
!        Pr( K'<xx ) =  Pr( K'<0 ) - Pr( xx<K'<0 )

! First, calculate  Pr( Min(0,xx)<K'<Max(0,xx) )

aqa = a2/(q+a2)
aqal = log(aqa)
qqal = q2*log(one-aqa)
q2l = dlgama(q2)
r2l = dlgama(r2)
r2l1mx = r2*log(one-xarg)
xl = log(xarg)
k = max( 0, int( a2-(a2+a2)/q ) )   ! Mode of neg. bin. distribution
if ( k < kmin ) k = 0               ! k is not large enough to be worthwile
k = (k/2)*2                         ! Make k an even number

! To deal with the case where k=0 and r<=1, xgamf is updated at
! end of iteration loop, while xgamb is updated at beginning of loop

dj2 = k*half
betaf(0) = betacdf( xarg, dj2+half, r2, ier )
if ( ier /= 0 ) then
  ier = 4
  return
end if
if ( k > 0 ) then   ! Remember k is even
  beta0 = betacdf( xarg, half, r2, ier )
  if ( ier /= 0 ) then
    ier = 4
    return
  end if
  if ( betaf(0) < beta0*betaratio ) then ! Beta at k is small, so k is moved to
    k = xarg*k                          ! somewhere between modes of g_j's and H_j's
    k = (k/2)*2                         ! Make k an even number
    dj2 = k*half
    betaf(0) = betacdf( xarg, dj2+half, r2, ier )
    if ( ier /= 0 ) then
      ier = 4
      return
    end if
  end if
  if ( k > 0 ) then
    gcoeff(0) = qqal - q2l + dlgama(dj2+q2) - dlgama(dj2+one) + dj2*aqal
    if ( gcoeff(0) >= explower ) then
      gcoeff(0) = exp(gcoeff(0))*half
    else
      gcoeff(0) = zero
    end if
    dj2 = (k+1)*half
    betaf(1) = betacdf( xarg, dj2+half, r2, ier )
```

```
      if ( ier /= 0 ) then
        ier = 4
        return
      end if
      gcoeff(1) = qqal - q2l + dlgama(dj2+q2) - dlgama(dj2+one) + dj2*aqal
      if ( gcoeff(1) >= explower ) then
        gcoeff(1) = exp(gcoeff(1))*half
      else
        gcoeff(1) = zero
      end if
      xgamf(0) = (k+1)*half*xl+r2l1mx+dlgama(r2+(k+1)*half)-r2l-dlgama((k+3)*half)
      xgamf(1) = (k+2)*half*xl+r2l1mx+dlgama(r2+k*half+one)-r2l-dlgama((k+4)*half)
      xgamb(0) = xgamf(0)
      xgamb(1) = k*half*xl+r2l1mx+dlgama(r2+k*half)-r2l-dlgama((k+2)*half)
      if ( xgamf(0) >= explower .and. xgamf(1) >= explower .and.   &
           xgamb(1) >= explower ) then
        xgamf(0) = exp(xgamf(0))
        xgamf(1) = exp(xgamf(1))
        xgamb(0) = xgamf(0)
        xgamb(1) = exp(xgamb(1))
      else         ! xgamf/b at k too small for recurrence to hold
        k = 0      ! thus start at k=0
      end if
    end if
  else
    beta0 = betaf(0)
  end if
  if ( k > 0 ) then
    gcoefb(0) = gcoeff(0)
    betab(0) = betaf(0)
    dj2 = (k-1)*half
    gcoefb(1) = qqal - q2l + dlgama(dj2+q2) - dlgama(dj2+one) + dj2*aqal
    if ( gcoefb(1) >= explower ) then
      gcoefb(1) = exp(gcoefb(1))*half
    else
      gcoefb(1) = zero
    end if
    betab(1) = betaf(1) + xgamb(1)
  else
    gcoeff(0) = ((one-aqa)**q2)*half
    gcoeff(1) = qqal - q2l - dlg15 + half*aqal + dlgama(q2+half)
    if ( gcoeff(1) >= explower ) then
      gcoeff(1) = exp(gcoeff(1))*half
    else
      gcoeff(1) = zero
    end if
    betaf(0) = beta0
    betaf(1) = betacdf( xarg, one, r2, ier )
    if ( ier /= 0 ) then
      ier = 4
      return
    end if
```

```
    xgamf(0) = r2l1mx+dlgama(r2+half)-r2l-dlg15+half*xl
    xgamf(1) = (one-xarg)**r2*xarg*r2
    xgamb(0) = xgamf(0)
    xgamb(1) = (one-xarg)**r2
    if ( xgamf(0) >= explower ) then
      xgamf(0) = exp(xgamf(0))
      xgamb(0) = xgamf(0)
    else         ! xgamf/b at 0 too small for recurrence to hold
      ier = 3   ! thus quit
      return
    end if
    gcoefb(0) = zero
    gcoefb(1) = zero
    betab(0) = zero
    betab(1) = zero
  end if
  betak = betaf(0)
  cdf(0) = gcoeff(0)*betaf(0)      ! sum of even terms (positive)
  cdf(1) = gcoeff(1)*betaf(1) + & ! sum of odd terms (possibly negative)
           gcoefb(1)*betab(1)     ! (negative when xx < 0)
  sumg = gcoeff(0) + gcoeff(1) + gcoefb(1)
  prv = -one

  !  Iteration loops

  !  Do forward and backward computations k times or until convergence
  kit = min( k, maxitr )
  do j = 2, kit
    jm = mod(k+j,2)
    betaf(jm) = max( betaf(jm)-xgamf(jm), zero )
    gcoeff(jm) = gcoeff(jm)*(k+j-two+q)*aqa/(k+j)
    xgamb(jm) = xgamb(jm)*(k-j+3)/((k-j+1+r)*xarg)
    betab(jm) = betab(jm) + xgamb(jm)
    gcoefb(jm) = gcoefb(jm)*(k-j+two)/(aqa*(k-j+q))

    cdf(jm) = cdf(jm) + gcoeff(jm)*betaf(jm) + gcoefb(jm)*betab(jm)

    sumg = sumg + gcoeff(jm) + gcoefb(jm)
    if ( (one-sumg)*beta0 <= delta ) goto 10
    !  Looking for no more evolution of the sum
    if ( cdf(0) == prv(0) .and. cdf(1) == prv(1) ) then
      ier = -1
      goto 10
    end if
    prv(jm) = cdf(jm)
    xgamf(jm) = xgamf(jm)*xarg*(k+j-1+r)/(k+j+1)
  end do

  !  Do forward computations until convergence
  do j = max(kit,1)+1, maxitr
    jm = mod(k+j,2)
    betaf(jm) = max( betaf(jm)-xgamf(jm), zero )
```

```
   gcoeff(jm) = gcoeff(jm)*(k+j-two+q)*aqa/(k+j)

   cdf(jm) = cdf(jm) + gcoeff(jm)*betaf(jm)

   sumg = sumg + gcoeff(jm)
   if ( (one-sumg)*betaf(jm) <= delta ) goto 10
   !  Looking for no more evolution of the sum
   if ( cdf(0) == prv(0) .and. cdf(1) == prv(1) ) then
      ier = -1
      goto 10
   end if
   prv(jm) = cdf(jm)
   xgamf(jm) = xgamf(jm)*xarg*(k+j-1+r)/(k+j+1)
end do

!  Maximum number of iterations is reached

ier = 2

10 continue
!  The end: add Pr( K'<0 )
!  Pr( K'(q,r,abs(a1)) < 0 ) = Pr( t(q) < -abs(a1) )

if ( xneg ) then
   kprimecdf = tcdf( -abs(a1), q, bel, iok ) + cdf(1) - cdf(0)
else
   kprimecdf = tcdf( -abs(a1), q, bel, iok ) + cdf(1) + cdf(0)
end if
if ( iok /= 0 ) then
   ier = 4
   return
end if
if ( a1 < zero ) kprimecdf = one - kprimecdf

if ( ier == -1 ) then
   ! See if the "reverse" problem is OK
   call kprimebis( a1, r, q, x, delta, maxitr, iok, xx )
   if ( iok == 0 ) then
      kprimecdf = one - xx
      ier = 0
   end if
end if

   ! Check out of limits
if ( kprimecdf < zero ) then
   if ( kprimecdf >= -delta ) then
      kprimecdf = zero
   else
      ier = 5 + ier
   end if
else if ( kprimecdf > one ) then
   if ( kprimecdf <= one+delta ) then
      kprimecdf = one
```

```
      else
        ier = 5 + ier
      end if
    end if

  end function kprimecdf
  subroutine kprimebis( x, q, r, a1, delta, maxitr, ier, result )

  !    Returns in result the K' cdf
  !    This is the subroutine version of the Kprimecdf function used
  !    for the reverse problem when ier=-1 in the Kprimecdf function
  !    (Same arguments as in kprimecdf function)

    implicit none

    ! Arguments
    ! ---------

    real(kind=8), intent(in) :: x, q, r, a1, delta
    real(kind=8), intent(out) :: result
    integer, intent(in) :: maxitr
    integer, intent(out) :: ier

    ! local declarations
    ! ------------------

  !! real(kind=8), external :: dlgama
    real(kind=8), external :: betacdf, lprimecdf, tcdf

    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8, two=2.0_8
    real(kind=8), parameter :: dlg15=-0.120782237635245204_8  ! =log(gamma(1.5))
    real(kind=8), parameter :: bel=1.0e6_8, qlimit=2.0e6_8, rlimit=2.0e6_8
    real(kind=8), parameter :: eps=0.223e-15_8, explower=-706.893_8
    real(kind=8), parameter :: betaratio=0.01_8
    integer, parameter :: kmin=10    ! When starting point of iterations is below
                                     ! this limit "method 2" is not judged worthwile

    !    These constants are machine dependent:
    !    eps = machine epsilon
    !          (the smallest real such that 1.0 + eps > 1.0)
    !    explower = minimum valid argument for the exponential function

    real(kind=8) :: a2, aqa, aqal, beta0, betak, dj2, q2, q2l, qqal
    real(kind=8) :: r2, r2l, r2l1mx, sumg, xarg, xl, xx
    integer :: iok, j, jm, k, kit
    logical :: xneg
    real(kind=8) :: betab(0:1), betaf(0:1), gcoefb(0:1), gcoeff(0:1)
    real(kind=8) :: cdf(0:1), xgamb(0:1), xgamf(0:1)
    real(kind=8) :: prv(0:1)

    !---------------------------------------------------------------

    xx = zero
```

```
ier = 0

! Test for valid input arguments

if ( q <= zero .or. r <= zero .or. delta >= one .or. delta <= eps ) then
  ier = 1
  return
end if

! Case x = 0

if ( abs(x) < eps ) then
  cdf(0) = zero
  cdf(1) = zero
  xneg = .false.
  goto 10
end if

! If a1 is close to zero use approximation (exact if a1 = 0)

if ( abs(a1) < eps ) then
  result = tcdf( x, r, bel, ier )
  if ( ier /= 0 ) ier = 4
  return
end if

! If q or r is large enough use limiting distribution

if ( q > qlimit ) then
  result = one - lprimecdf( a1, r, x, delta, maxitr, ier ) ! noncentral t
  if ( ier /= 0 ) ier = 4
  return
else if ( r > rlimit ) then
  result = lprimecdf( x, q, a1, delta, maxitr, ier )
  if ( ier /= 0 ) ier = 4
  return
end if

! Define usefull parameters

xx = x
xarg = x*x
xarg = xarg / (xarg+r)
q2 = q*half
r2 = r*half
a2 = a1*a1

! Case xarg close to one (x tends to +/- infinity)

if ( abs(xarg-one) < eps+eps ) then
  if ( x > zero ) result = one
  return
end if
```

```
! Case a1 < 0 : change sign of x :
! Pr( K'(a1,1) < x ) = 1 - Pr( K'(-a1,1) < -x )

if ( a1 < zero ) xx = -xx
xneg = xx < zero          ! When a1 and x are not of same sign, the series is alternate
                          ! and this is flagged by xneg

! General case (iterations)

! When a1 > 0:
!    1) if xx > 0:
!       Pr( K'<xx ) =  Pr( K'<0 ) + Pr( 0<K'<xx )
!    2) if xx < 0:
!       Pr( K'<xx ) =  Pr( K'<0 ) - Pr( xx<K'<0 )

! First, calculate  Pr( Min(0,xx)<K'<Max(0,xx) )

aqa = a2/(q+a2)
aqal = log(aqa)
qqal = q2*log(one-aqa)
q2l = dlgama(q2)
r2l = dlgama(r2)
r2l1mx = r2*log(one-xarg)
xl = log(xarg)
k = max( 0, int( a2-(a2+a2)/q ) )   ! Mode of neg. bin. distribution
if ( k < kmin ) k = 0               ! k is not large enough to be worthwile
k = (k/2)*2                         ! Make k an even number

! To deal with the case where k=0 and r<=1, xgamf is updated at
! end of iteration loop, while xgamb is updated at beginning of loop

dj2 = k*half
betaf(0) = betacdf( xarg, dj2+half, r2, ier )
if ( ier /= 0 ) then
  ier = 4
  return
end if
if ( k > 0 ) then   ! Remember k is even
  beta0 = betacdf( xarg, half, r2, ier )
  if ( ier /= 0 ) then
    ier = 4
    return
  end if
  if ( betaf(0) < beta0*betaratio ) then ! Beta at k is small, so k is moved to
    k = xarg*k                          ! somewhere between modes of g_j's and H_j's
    k = (k/2)*2                         ! Make k an even number
    dj2 = k*half
    betaf(0) = betacdf( xarg, dj2+half, r2, ier )
    if ( ier /= 0 ) then
      ier = 4
      return
    end if
```

```
      end if
      if ( k > 0 ) then
        gcoeff(0) = qqal - q2l + dlgama(dj2+q2) - dlgama(dj2+one) + dj2*aqal
        if ( gcoeff(0) >= explower ) then
          gcoeff(0) = exp(gcoeff(0))*half
        else
          gcoeff(0) = zero
        end if
        dj2 = (k+1)*half
        betaf(1) = betacdf( xarg, dj2+half, r2, ier )
        if ( ier /= 0 ) then
          ier = 4
          return
        end if
        gcoeff(1) = qqal - q2l + dlgama(dj2+q2) - dlgama(dj2+one) + dj2*aqal
        if ( gcoeff(1) >= explower ) then
          gcoeff(1) = exp(gcoeff(1))*half
        else
          gcoeff(1) = zero
        end if
        xgamf(0) = (k+1)*half*xl+r2l1mx+dlgama(r2+(k+1)*half)-r2l-dlgama((k+3)*half)
        xgamf(1) = (k+2)*half*xl+r2l1mx+dlgama(r2+k*half+one)-r2l-dlgama((k+4)*half)
        xgamb(0) = xgamf(0)
        xgamb(1) = k*half*xl+r2l1mx+dlgama(r2+k*half)-r2l-dlgama((k+2)*half)
        if ( xgamf(0) >= explower .and. xgamf(1) >= explower .and.   &
             xgamb(1) >= explower ) then
          xgamf(0) = exp(xgamf(0))
          xgamf(1) = exp(xgamf(1))
          xgamb(0) = xgamf(0)
          xgamb(1) = exp(xgamb(1))
        else         ! xgamf/b at k too small for recurrence to hold
          k = 0      ! thus start at k=0
        end if
      end if
    else
      beta0 = betaf(0)
    end if
    if ( k > 0 ) then
      gcoefb(0) = gcoeff(0)
      betab(0) = betaf(0)
      dj2 = (k-1)*half
      gcoefb(1) = qqal - q2l + dlgama(dj2+q2) - dlgama(dj2+one) + dj2*aqal
      if ( gcoefb(1) >= explower ) then
        gcoefb(1) = exp(gcoefb(1))*half
      else
        gcoefb(1) = zero
      end if
      betab(1) = betaf(1) + xgamb(1)
    else
      gcoeff(0) = ((one-aqa)**q2)*half
      gcoeff(1) = qqal - q2l - dlg15 + half*aqal + dlgama(q2+half)
      if ( gcoeff(1) >= explower ) then
```

```
      gcoeff(1) = exp(gcoeff(1))*half
    else
      gcoeff(1) = zero
    end if
    betaf(0) = beta0
    betaf(1) = betacdf( xarg, one, r2, ier )
    if ( ier /= 0 ) then
      ier = 4
      return
    end if
    xgamf(0) = r2l1mx+dlgama(r2+half)-r2l-dlg15+half*xl
    xgamf(1) = (one-xarg)**r2*xarg*r2
    xgamb(0) = xgamf(0)
    xgamb(1) = (one-xarg)**r2
    if ( xgamf(0) >= explower ) then
      xgamf(0) = exp(xgamf(0))
      xgamb(0) = xgamf(0)
    else         ! xgamf/b at 0 too small for recurrence to hold
      ier = 3   ! thus quit
      return
    end if
    gcoefb(0) = zero
    gcoefb(1) = zero
    betab(0) = zero
    betab(1) = zero
  end if
  betak = betaf(0)
  cdf(0) = gcoeff(0)*betaf(0)     ! sum of even terms (positive)
  cdf(1) = gcoeff(1)*betaf(1) + & ! sum of odd terms (possibly negative)
           gcoefb(1)*betab(1)     ! (negative when xx < 0)
  sumg = gcoeff(0) + gcoeff(1) + gcoefb(1)
  prv = -one

  !  Iteration loops

  !  Do forward and backward computations k times or until convergence
  kit = min( k, maxitr )
  do j = 2, kit
    jm = mod(k+j,2)
    betaf(jm) = max( betaf(jm)-xgamf(jm), zero )
    gcoeff(jm) = gcoeff(jm)*(k+j-two+q)*aqa/(k+j)
    xgamb(jm) = xgamb(jm)*(k-j+3)/((k-j+1+r)*xarg)
    betab(jm) = betab(jm) + xgamb(jm)
    gcoefb(jm) = gcoefb(jm)*(k-j+two)/(aqa*(k-j+q))

    cdf(jm) = cdf(jm) + gcoeff(jm)*betaf(jm) + gcoefb(jm)*betab(jm)

    sumg = sumg + gcoeff(jm) + gcoefb(jm)
    if ( (one-sumg)*beta0 <= delta ) goto 10
    !  Looking for no more evolution of the sum
    if ( cdf(0) == prv(0) .and. cdf(1) == prv(1) ) then
      ier = -1
```

```
        goto 10
      end if
      prv(jm) = cdf(jm)
      xgamf(jm) = xgamf(jm)*xarg*(k+j-1+r)/(k+j+1)
    end do

    ! Do forward computations until convergence
    do j = max(kit,1)+1, maxitr
      jm = mod(k+j,2)
      betaf(jm) = max( betaf(jm)-xgamf(jm), zero )
      gcoeff(jm) = gcoeff(jm)*(k+j-two+q)*aqa/(k+j)

      cdf(jm) = cdf(jm) + gcoeff(jm)*betaf(jm)

      sumg = sumg + gcoeff(jm)
      if ( (one-sumg)*betaf(jm) <= delta ) goto 10
      ! Looking for no more evolution of the sum
      if ( cdf(0) == prv(0) .and. cdf(1) == prv(1) ) then
        ier = -1
        goto 10
      end if
      prv(jm) = cdf(jm)
      xgamf(jm) = xgamf(jm)*xarg*(k+j-1+r)/(k+j+1)
    end do

    ! Maximum number of iterations is reached

    ier = 2

    10 continue
    ! The end: add Pr( K'<0 )
    ! Pr( K'(q,r,abs(a1)) < 0 ) = Pr( t(q) < -abs(a1) )

    if ( xneg ) then
      result = tcdf( -abs(a1), q, bel, iok ) + cdf(1) - cdf(0)
    else
      result = tcdf( -abs(a1), q, bel, iok ) + cdf(1) + cdf(0)
    end if
    if ( iok /= 0 ) then
      ier = 4
      return
    end if
    if ( a1 < zero ) result = one - result

  end subroutine kprimebis
```

Chapter 15
K-square distribution

Function k2cdf(x, p, q, r, a2, delta, maxitr, ier)
File : K2cdf.f90

```
      function k2cdf( x, p, q, r, a2, delta, maxitr, ier )

      !------------------------------------------------------------------------
      !
      !     Calculates the probability that a random variable distributed
      !     according to the K-square distribution with P, Q and R
      !     degrees of freedom and A2 eccentricity parameter, is less than
      !     or equal to X
      !
      !     X     - Input . Value of the variable          (X >= 0) - Real
      !     P     - Input . First degrees of freedom       (P >  0) - Real
      !     Q     - Input . Second   "    "     "         (Q >  0) - Real
      !     R     - Input . Third    "    "     "         (R >  0) - Real
      !     A2    - Input . Eccentricity parameter         (A2>= 0) - Real
      !     DELTA - Input . Maximum absolute error required on      - Real
      !                     k2cdf (stopping criteria)
      !                     (eps < delta < 1 where eps is machine
      !                     epsilon; see parameter statement below)
      !     MAXITR- Input . Maximum number of iterations            - Integer
      !     IER   - Output. Return code :                           - Integer
      !                     0 = normal
      !                    -1 = no more evolution of the sum but
      !                         required accuracy not reached yet
      !                         (then k2cdf = value at last iteration)
      !                     1 = invalid input argument
      !                         (then k2cdf = zero)
      !                     2 = maximum number of iterations reached
      !                         (then k2cdf = value at last iteration,
      !                         or zero)
      !                     3 = cannot be computed
      !                         (then k2cdf = zero)
      !                     4 = error in auxiliary function
      !                         (betacdf or lambda2cdf)
      !
      !     External functions called:
      !       BETACDF LAMBDA2CDF
      !       DLGAMA (Log(Gamma(.)) if not in FORTRAN library
      !
      !     Fortran functions called:
      !       ABS  EXP  INT  LOG  MAX  MIN  (and DLGAMA if available)
      !
      !     Uses "method 2", see Benton & Krishnamoorthy (2003),
      !     Computational Statistics & Data Analysis, 43, 249-267.
      !     N.B. The mode is taken as the starting point for iterations
      !     (forward and backward).
      !     Starting index is modified if worthwile (see parameter betaratio)
      !     To deal with the case where p+r<=2, xgamf is updated at the
      !     end of the iteration loop rather than at the beginning.
      !
      !------------------------------------------------------------------------

        implicit none
```

```
    ! Function
    ! --------

    real(kind=8) :: k2cdf

    ! Arguments
    ! ---------

    real(kind=8), intent(in) :: x, p, q, r, a2, delta
    integer, intent(in) :: maxitr
    integer, intent(out) :: ier

    ! Local declarations
    ! ------------------

  !! real(kind=8), external :: dlgama
    real(kind=8), external :: betacdf, lambda2cdf

    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
    real(kind=8), parameter :: rlimit=1.0e6_8
    real(kind=8), parameter :: eps=0.223e-15_8, explower=-706.893_8
    real(kind=8), parameter :: betaratio=0.01_8
    integer, parameter :: kmin=10    ! When starting point of iterations is below
                                     ! this limit "method 2" is not judged worthwile

    !    These constants are machine dependent:
    !    eps = machine epsilon
    !          (the smallest real such that 1.0 + eps > 1.0)
    !    explower = minimum valid argument for the exponential function

    real(kind=8) :: a, aqa, b, beta0, betab, betaf, betak, cdf
    real(kind=8) :: gcoefb, gcoeff, prv, p2, q2, sumg, xarg, xgamb, xgamf
    integer :: j, k, kit

    !-----------------------------------------------------------------

    k2cdf = zero
    ier = 0

    ! Test for valid input arguments

    if ( x < zero .or. a2 < zero                        &
      .or. p <= zero .or. q <= zero .or. r <= zero     &
      .or. delta >= one .or. delta <= eps ) then
      ier = 1
      return
    end if

    ! Case x = 0

    if ( x < eps ) return

    xarg = p*x / (p*x+r)
```

```
! Case p*x/(p*x+r) = 1

if ( abs(xarg-one) < eps ) then
  k2cdf = one
  return
end if

p2 = p * half
q2 = q * half
b = r * half

! If r is large enough use limiting distribution

if ( r > rlimit ) then
  k2cdf = lambda2cdf( x, p, q, a2, delta, maxitr, ier )
  if ( ier /= 0 ) ier = 4
  return
end if

beta0 = betacdf( xarg, p2, b, ier ) ! Beta at step 0
if ( ier /= 0 ) then
  ier = 4
  return
end if

! Case a2 = 0

if ( a2 < eps ) then
  k2cdf = beta0
  if ( ier /= 0 ) ier = 4
  return
end if

! General case (iterations)

aqa = a2/(q+a2)
k = int( (q2-one)*a2/q )   ! Mode of neg. bin. distribution
if ( k < kmin ) k = 0      ! k is not large enough to be worthwile
a = p2 + k
betak = betacdf( xarg, a, b, ier )
if ( ier /= 0 ) then
  ier = 4
  return
end if
if ( k > 0 .and. betak < beta0*betaratio ) then ! Beta at k is small, so k is moved to
  k = xarg*k                                    ! somewhere between modes of g_j's and H_j's
  a = p2 + k
  if ( k > 0 ) then
    betak = betacdf( xarg, a, b, ier )
    if ( ier /= 0 ) then
      ier = 4
      return
    end if
```

```
    else
      betak = beta0
    end if
  end if
  xgamf = a*log(xarg) + b*log(one-xarg) + dlgama(a+b) - dlgama(a+one) - dlgama(b)
  if ( xgamf >= explower ) then
    xgamf = exp(xgamf)
  else if ( k > 0 ) then   ! xgamf at k too small for recurrence to hold
    k = 0                  ! thus start at k=0
    a = p2
    xgamf = a*log(xarg) + b*log(one-xarg) + dlgama(a+b) - dlgama(a+one) - dlgama(b)
    if ( xgamf >= explower ) then
      xgamf = exp(xgamf)
    else                           ! Cannot be computed
      ier = 3
      return
    end if
    betak = beta0
  else                             ! Cannot be computed
    ier = 3
    return
  end if
  betaf = betak
  betab = betak
  xgamb = xgamf
  gcoeff = dlgama(q2+k) - dlgama(k+one) - dlgama(q2) + k*log(aqa) + q2*log(one-aqa)
  if ( gcoeff >= explower ) then
    gcoeff = exp(gcoeff)
  else
    gcoeff = zero
  end if
  gcoefb = gcoeff
  cdf = gcoeff*betaf
  sumg = gcoeff
  prv = -one

  !  Iteration loops

  !  Do forward and backward computations k times or until convergence
  kit = min( k, maxitr )
  do j = 1, kit
      !  Forward computations
    betaf = max( betaf-xgamf, zero )
    gcoeff = gcoeff*(k+j-1+q2)*aqa/(k+j)
      !  Backward computations
    xgamb = xgamb*(1-j+a)/(xarg*(a+b-j))
    betab = betab + xgamb
    gcoefb = gcoefb*(k-j+1)/(aqa*(k-j+q2))

    cdf = cdf + gcoeff*betaf + gcoefb*betab

    sumg = sumg + gcoeff + gcoefb
```

```
      if ( (one-sumg)*beta0 <= delta ) then
        k2cdf = cdf
        return
      end if
      ! Looking for no more evolution of the sum
      if ( cdf == prv ) then
        ier = -1
        k2cdf = cdf
        return
      end if
      prv = cdf
      xgamf = xgamf*(j-1+a+b)*xarg/(j+a)
    end do

    ! Do forward computations until convergence
    do j = kit+1, maxitr
      betaf = max( betaf-xgamf, zero )
      gcoeff = gcoeff*(k+j-1+q2)*aqa/(k+j)

      cdf = cdf + gcoeff*betaf

      sumg = sumg + gcoeff
      if ( (one-sumg)*betaf <= delta ) then
        k2cdf = cdf
        return
      end if
      ! Looking for no more evolution of the sum
      if ( cdf == prv ) then
        ier = -1
        k2cdf = cdf
        return
      end if
      prv = cdf
      xgamf = xgamf*(j-1+a+b)*xarg/(j+a)
    end do

    ! Maximum number of iterations is reached

    ier = 2
    k2cdf = cdf

  end function k2cdf
```

Chapter 16
Ksi-square distribution

Function ksi2cdf(x, p, q, a2, delta, maxitr, iap, ier)
File : Ksi2cdf.f90

```
function ksi2cdf( x, p, q, a2, delta, maxitr, iap, ier )

!------------------------------------------------------------------------
!
!    Computes the probability that a random variable distributed
!    according to the ksi-square distribution with P and Q degrees
!    of freedom and A2 eccentricity parameter, is less than or
!    equal to X.
!
!    X     - Input . Value of the variable          (X >= 0) - Real
!    P     - Input . First degrees of freedom       (P >  0) - Real
!    Q     - Input . Second   "    "     "          (Q >  0) - Real
!    A2    - Input . Eccentricity parameter         (A2>= 0) - Real
!    DELTA - Input . Maximum absolute error required on      - Real
!                    ksi2cdf (stopping criterion)
!                    (eps < DELTA < 1 where eps is machine
!                    epsilon; see parameter statement below)
!    MAXITR- Input . Maximum number of iterations            - Integer
!                    (2000 is sufficient for most cases)
!    IAP   - Output. 0 if no approximation used              - Integer
!                    1 otherwise
!    IER   - Output. Return code :                           - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then KSi2cdf = zero)
!                    2 = maximum number of iterations reached
!                        (then KSi2cdf = value at last iteration,
!                                or zero)
!                    3 = required accuracy cannot be reached
!                        (then KSi2cdf = value at last iteration)
!                    4 = error in auxiliary function
!                        (betacdf or phi2cdf)
!
!    External functions called:
!      BETACDF  PHI2CDF
!      DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!    Fortran functions called:
!      ABS  EXP  INT  LOG  MAX  NINT  SIGN  SQRT
!      (and DLGAMA if available)
!
!    Algorithm by J-L. Guigues (1981), improved by B. Lecoutre (1986)
!
!    J. Poitevineau (CNRS-URA1201)
!    January 01, 1991 (First version 26/04/85)
!    Modif.  23/09/93 (jjj)  25/11/93 (jj, zg)  10/08/94 (y1, phi2cdf)
!
!------------------------------------------------------------------------

   implicit none

   ! Function
```

```
    ! --------

    real(kind=8) :: ksi2cdf

    ! Arguments
    ! ---------

    real(kind=8), intent(in) :: x, p, q, a2, delta
    integer, intent(in) :: maxitr
    integer, intent(out) :: iap, ier

    ! local declarations
    ! ------------------

   !! real(kind=8), external :: dlgama
    real(kind=8), external :: betacdf, phi2cdf

    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
    real(kind=8), parameter :: two=one+one
    real(kind=8), parameter :: qlimit=1.0e9_8
    real(kind=8), parameter :: eps=0.223e-15_8, tinyr=1.0e-307_8, explower=-706.893_8
    real(kind=8), parameter :: relerr=1.0e-14_8 ! Relative error assumed
                                          ! in recurrence calculations
    real(kind=8), parameter :: xp=tinyr/relerr
    real(kind=8), parameter :: xmxint=2147483647.0_8

    !     These constants are machine dependent:
    !     eps   = machine epsilon
    !           (the smallest real such that 1.0 + eps > 1.0)
    !     tinyr = the smallest positive real
    !     explower = minimum valid argument for the exponential function
    !           (i.e. log(tinyr))
    !     xmxint= maximum positive integer

    real(kind=8) :: aqal, bl, cl, dax, dj, dj1,  &
                    erp, err, g, gcj, gcl,       &
                    pq2, p2, qqal, q2, r, rl,    &
                    sum, sumc,                   &
                    v, w, y, yyl,                &
                    y1, z, zg, zm, zn
    integer :: iok, j, jj, jok
    logical :: lalf

    !-----------------------------------------------------------------

    ksi2cdf = zero
    iap = 0
    ier = 0

    ! Test for valid input arguments

    if ( x < zero .or. a2 < zero .or. p <= zero .or. q <= zero  &
       .or. delta >= one .or. delta <= eps ) then
```

```
   ier  = 1
   return
end if

y  = x      / (a2+q+x)
y1 = (a2+q) / (a2+q+x)

!  y near 0 or 1

if ( y  <= tinyr ) return
if ( y1 <= tinyr ) then
   ksi2cdf = one
   return
end if

!  Define usefull parameters

p2  = p * half
q2  = q * half
pq2 = p2 + q2

!  Case a2 = 0

if ( a2 < eps ) then
   ksi2cdf = betacdf( x/(x+q), p2, q2, iok )
   if ( iok /= 0 ) ier = 4
   return
end if

!  Case p = 1

if ( abs(p-one) < eps ) then
   dax  = two * sqrt( a2*x )
   ksi2cdf = half * ( sign( one, x-a2 ) *                    &
            betacdf( (a2+x-dax)/(a2+x-dax+q),half,q2,iok ) + &
            betacdf( (a2+x+dax)/(a2+x+dax+q),half,q2,jok ) )
   if ( iok /= 0 .or. jok /= 0 ) ier = 4
   return
end if

! If q is large enough use limiting distribution

if ( q > qlimit ) then
   ksi2cdf = phi2cdf( x/p, p, a2, delta, maxitr, ier )
   if ( ier /= 0 ) ier = 4
   iap = 1
   return
end if

!  Calculate 1-f(x) or f(x) (lalf is the indicator)

lalf = .false.
z    = y
```

```
zm   = p2
zn   = q2
if ( abs(y-half) < eps ) then
  ! Look for special case y = 0.5 and p = q
  if ( abs(p-q) < eps ) then
    ksi2cdf = half
    return
  end if
  if ( p > q ) then
    lalf = .true.
    zm   = q2
    zn   = p2
  end if
else if ( y > half ) then
  lalf = .true.
  z    = y1
  zm   = q2
  zn   = p2
end if

! General case (iterations)
! g's  are decreasing for j >= jj = -v/w + 1
! cl's are decreasing for j >= jjj = a2/2 - a2/q - 1
! gcj, sumc, zg are only used for stopping rule
! Logs are used to avoid underflows

! Define constants

err  = delta * half
erp  = err / relerr
yyl  = log( y*y1 )
qqal = q2 * log( q/(q+a2) )
aqal = log( a2/(q+a2) )
v    = pq2*z - zn
w    = z + z - one

! Initialize

bl = zero
cl = qqal
g  = betacdf( z, zm, zn, iok )
if ( iok /= 0 ) then
  ier  = 4
  return
end if
zg = g
rl = p2*log(y) + q2*log(y1) + dlgama(pq2) - dlgama(p2) - dlgama(q2) - log(p2) - log(q2)
if ( rl >= explower ) then
  r = exp( rl )
else
  r = zero
end if
```

```
sum  = zero
gcj  = zero
sumc = zero

!  Define jj (and associated upper bound zg)
!  (note that w=0 if and only if y=0.5, and in such a case zm<zn)

if ( zm > zn ) then
  zg = -v/w
  if ( abs(zg) < xmxint ) then
    jj = int( zg ) + 1
    if ( jj > 0 ) then
      zg = betacdf( z, zm+jj, zn+jj, iok )
      if ( iok /= 0 ) then
        ier  = 4
        return
      end if
    end if
  else
    zg = one
    jj = maxitr + 1
  end if
else
  jj = 0
end if

dj = zero

!  Iteration loop

do j = 0, maxitr

  if ( g > zero ) then
    gcl = log(g) + cl
    if ( gcl >= explower ) then
      gcl = exp( gcl )
      sum = sum + gcl
      gcj = gcj + gcl*dj
      !  Check loss of accuracy
      if ( gcj >= erp ) then
        ier = 3
        goto 10
      end if
    end if
  end if

  !  Check accuracy (stopping rule)
  !  xp is used to prevent possible underflow

  if ( cl >= explower ) sumc = sumc + exp(cl)
  if ( j >= jj ) zg = g
  if ( gcj >= xp ) then
    if ( relerr*gcj+zg*(one-sumc) < err ) goto 10
```

```
      else
        if (            zg*(one-sumc) < err ) goto 10
      end if

      !  Prepare next iteration

      dj1 = j + 1
      bl  = bl + log( (q2+dj)/dj1 )
      cl  = bl + qqal + dj1*aqal
      g   = g  + (v+w*dj)*r
      rl  = rl + yyl + log( ((dj+dj+pq2)/(dj1+p2)) * ((dj+dj1+pq2)/(dj1+q2)) )
      if ( rl >= explower ) then
        r = exp( rl )
      else
        r = zero
      end if
      dj = dj1

    end do

    !  Maximum number of iterations is reached

    ier = 2

    !  The end

    10 continue
    if ( lalf ) then
      ksi2cdf = one - sum
    else
      ksi2cdf = sum
    end if

  end function ksi2cdf
```

Chapter 17
Lambda-prime distribution

Function lprimecdf(x, q, a, delta, maxitr, ier)
File : Lprimecdf.f90

function lprimecdf(x, q, a, delta, maxitr, ier)

```
!------------------------------------------------------------------------
!
!    Calculates the probability that a random variable distributed
!    according to the Lambda' distribution with Q degrees of freedom,
!    A centrality parameter, is less than or equal to X
!    Note: P( L'q (a) < x ) = P( t'q (x) > a )
!
!
!    X     - Input . Value of the variable                  - Real
!    Q     - Input . Degrees of freedom             (Q >  0) - Real
!    A     - Input . Eccentricity parameter                  - Real
!    DELTA - Input . Maximum absolute error required on     - Real
!                    lprimecdf (stopping criterion)
!                    (eps < delta < 1 where eps is machine
!                    epsilon; see parameter statement below)
!    MAXITR- Input . Maximum number of iterations           - Integer
!    IER   - Output. Return code :                          - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then lprimecdf = zero)
!                    2 = maximum number of iterations reached
!                        (then lprimecdf = value at last iteration,
!                                or zero)
!                    3 = required accuracy cannot be reached
!                        (then lprimecdf = value at last iteration)
!                    4 = error in auxiliary function
!                        (chi2cdf or tcdf)
!                    7 = result out of limits (i.e. <0 or >1)
!
!    External functions called:
!      CHI2CDF  NCDF  TCDF
!      DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!    Fortran functions called:
!      ABS  EXP  LOG  NINT  (and DLGAMA if available)
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: lprimecdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, q, a, delta
   integer, intent(in) :: maxitr
   integer, intent(out) :: ier

   ! local declarations
```

```
    ! ------------------

   !! real(kind=8), external :: dlgama
    real(kind=8), external :: chi2cdf, ncdf, tcdf

    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
    real(kind=8), parameter :: onep5=1.5_8, two=2.0_8, three=3.0_8, four=4.0_8
    real(kind=8), parameter :: dlg15=-0.120782237635245204_8  ! =log(gamma(1.5))
    real(kind=8), parameter :: twol=0.6931471805599453_8      ! =log(2)
    real(kind=8), parameter :: bel=1.0e6_8, qlimit=1.0e5_8, cinf=1.0e-20_8
    real(kind=8), parameter :: dflimit=1.0e6_8

    real(kind=8), parameter :: eps=0.223e-15_8, tinyr=1.0e-307_8, explower=-706.893_8
    real(kind=8), parameter :: relerr=1.0e-14_8 ! Relative error assumed
                                          ! in recurrence calculations
    real(kind=8), parameter :: xp=tinyr/relerr
    integer, parameter :: jmax=200

    !     dflimit = limit for approximation in chi2cdf function
    !     These constants are machine dependent:
    !     eps   = machine epsilon
    !           (the smallest real such that 1.0 + eps > 1.0)
    !     tinyr = the smallest positive real
    !     explower = minimum valid argument for the exponential function
    !           (i.e. log(tinyr))

    real(kind=8) :: aqal, a2, dj, dj2, erp, err, ga, g0,  &
                    kgj, kgl,                             &
                    qqal, q2, q2l,                         &
                    sum, sumg, sumgk, sumneg,                 &
                    xarg, xl, x2, xx
    integer :: iok, j, ja, jjj, jm, jz, j0
    logical :: xneg
    real(kind=8) :: gl(0:1), kx(0:1), rl(0:1)

    !-----------------------------------------------------------------

    lprimecdf = zero
    ier = 0

    ! Test for valid input arguments

    if ( q <= zero .or. delta >= one .or. delta <= eps ) then
      ier = 1
      return
    end if

    ! Case x = 0

    if ( abs(x) < eps ) then
      sum = zero
      sumneg = zero
      xneg = .false.
```

```
   goto 10
end if

! If a is close to zero or q is large enough use approximation
! (exact if a = 0)

if ( abs(a) < eps ) then
   lprimecdf = ncdf( x )
   return
else if ( q > qlimit ) then
   g0 = one - one/(four*q)
   xl = g0*a                    ! mean
   x2 = one + (one-g0*g0)*a*a   ! variance
   lprimecdf = ncdf( (x-xl)/sqrt(x2) )
   return
end if

! Define usefull parameters

xx   = x
xarg = x*x
x2   = half * xarg
xl   = log( xarg )
q2   = q * half
a2   = a*a

! Case a < 0 : change sign of x :
! Pr( L'(a,1) < x ) = 1 - Pr( L'(-a,1) < -x )

if ( a < zero ) xx = -xx
xneg = xx < zero

! General case (iterations)
! kx's are decreasing for all j's
! gl's are decreasing for j >= jjj = 2*(a2/2 - a2/q - 1)
! kgj, sumg, sumgk are only used for stopping rule
! Logs are used to avoid underflows

err  = delta * half
erp  = err / relerr
qqal = q2 * log( q/(q+a2) )
aqal = log( a2/(q+a2) )
q2l  = dlgama( q2 )
jjj  = 2 * nint( a2*half - a2/q - half )
jjj  = max( jjj, 0 )

! Examine rate of convergence:
! find starting index for iterations (j0)
! cinf is a limit under which sum of g coefficients is
! considered as negligible
! g0 is gl at j0

j0 = 0
```

```
g0 = qqal - twol
if ( jjj >= jmax ) then
  if ( g0 < log(cinf) ) then
    dj2 = half * jjj
    ga  = qqal - twol - q2l + dlgama(dj2+q2) - dlgama(dj2+one) + dj2*aqal
    if ( ga < explower )  goto 5   !  Cannot be computed
    if ( exp(ga) < cinf ) goto 5   !  Cannot be computed
    ja = 0
    jz = jjj
    ga = g0
    do
      j0  = (ja+jz) / 2
      dj2 = half * j0
      g0  = qqal - twol - q2l + dlgama(dj2+q2) - dlgama(dj2+one) + dj2*aqal
      if ( g0 >= explower ) then
        if ( (j0+1)*exp(g0) < cinf ) then
          ja = j0
          ga = g0
        else
          jz = j0
        end if
      else
        ja = j0
        ga = g0
      end if
      if ( ja+1 >= jz ) exit
    end do
    j0 = ja
    g0 = ga
  end if
end if
5 continue

!  Initialize

!  When j0 > 0, it is assumed sumg at j0 is negligible,
!  and the upper bound is taken for kgj:
!  kgj = sum( j*g*kx ) <= sum( j*g )  (as kx <= 1)
!                      <= exp(g0)*sum( j )    (as j <= j0 < jjj)
!                      <= exp(g0)*j0*(j0+1)/2

kgj = zero
if ( j0 > 0 ) then
  jm = mod(j0,2)
  dj = j0
  if ( g0 >= explower ) kgj = (dj+one) * dj * half * exp(g0)
  gl(jm) = g0
  kx(jm) = chi2cdf( xarg, dj+one, dflimit, ier )
  if ( ier /= 0 ) then
    ier = 4
    return
  end if
```

```
    rl(jm) = (dj+one)*half*log(x2) - x2 - dlgama((dj+three)*half)
    jm  = abs( jm-1 )
    dj  = j0 + 1
    dj2 = dj * half
    gl(jm) = qqal - twol - q2l + dlgama(dj2+q2) - dlgama(dj2+one) + dj2*aqal
    kx(jm) = chi2cdf( xarg, dj+one, dflimit, ier )
    if ( ier /= 0 ) then
      ier = 4
      return
    end if
    rl(jm) = (dj+one)*half*log(x2) - x2 - dlgama((dj+three)*half)
  else
    gl(0) = qqal - twol
    gl(1) = gl(0) - q2l - dlg15 + half*aqal + dlgama(q2+half)
    kx(0) = chi2cdf( xarg, one, dflimit, ier )
    if ( ier /= 0 ) then
      ier = 4
      return
    end if
    kx(1) = chi2cdf( xarg, two, dflimit, ier )
    if ( ier /= 0 ) then
      ier = 4
      return
    end if
    rl(0) = half*log(x2) - ( x2 + dlgama(onep5) )
    rl(1) = log(x2) - ( x2 + dlgama(two) )
  end if
  sum   = zero
  sumneg= zero
  sumg  = zero
  sumgk = zero

  !  Iteration loop

  do j = j0, j0+maxitr
    dj = j
    jm = mod(j,2)

    if ( kx(jm) > zero ) then
      kgl = log(kx(jm)) + gl(jm)
      if ( kgl >= explower ) then
        kgl   = exp(kgl)
        sumgk = sumgk + kgl
        !  When x<0 the signs of the alternating series are reversed
        !  so that sum will be added to the t in the final step
        !  rather than substracted
        if ( xneg .and. jm == 0 ) then
          sumneg = sumneg + kgl  ! sum of negative terms
        else
          sum = sum + kgl
        end if
        kgj = kgj + kgl*dj
```

```
            ! Check loss of accuracy
            if ( kgj >= erp ) then
              ier = 3
              goto 10
            end if
          end if
        end if

        ! Check accuracy (stopping rule)
        ! xp is used to prevent possible underflow

        if ( gl(jm) >= explower ) sumg = sumg + exp(gl(jm))
        if ( kgj >= xp ) then
          if ( relerr*kgj+kx(jm)*(one-sumg) < err ) goto 10
        else
          if (             kx(jm)*(one-sumg) < err ) goto 10
        end if

        ! Prepare next iteration

        dj2    = dj * half
        gl(jm) = gl(jm) + aqal + log(dj2+q2) - log(dj2+one)
        if ( rl(jm) >= explower ) kx(jm) = kx(jm) - exp( rl(jm) )
        rl(jm) = rl(jm) - log(dj+three) + xl

      end do

      ! Maximum number of iterations is reached

      ier = 2

      ! The end

      10 continue
      ! Pr( L'(q,abs(a)) < 0 ) = Pr( t(q) < -abs(a) )
      lprimecdf = sum - sumneg + tcdf( -abs(a), q, bel, iok )
      if ( iok /= 0 ) then
        ier = 4
        return
      end if
      if ( a < zero ) lprimecdf = one - lprimecdf
      if ( lprimecdf < zero ) then
        if ( lprimecdf >= -delta ) then
          lprimecdf = zero
        else
          ier = 7
        end if
      else if ( lprimecdf > one ) then
        if ( lprimecdf <= one+delta ) then
          lprimecdf = one
        else
          ier = 7
        end if
```

```
    end if

  end function lprimecdf
```

Chapter 18
Lambda-square distribution

Function lambda2cdf(x, p, q, a2, delta, maxitr, ier)
File : Lambda2cdf.f90

```
function lambda2cdf( x, p, q, a2, delta, maxitr, ier )

!------------------------------------------------------------------------
!
!    Calculates the probability that a random variable distributed
!    according to the lambda-square distribution with P and Q
!    degrees of freedom and A2 eccentricity parameter, is less than
!    or equal to X
!
!    X     - Input . Value of the variable          (X >= 0) - Real
!    P     - Input . First degrees of freedom        (P >  0) - Real
!    Q     - Input . Second   "    "     "          (Q >  0) - Real
!    A2    - Input . Eccentricity parameter         (A2>= 0) - Real
!    DELTA - Input . Maximum absolute error required on      - Real
!                    lambda2cdf (stopping criterion)
!                    (eps < delta < 1 where eps is machine
!                    epsilon; see parameter statement below)
!    MAXITR- Input . Maximum number of iterations            - Integer
!    IER   - Output. Return code :                           - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then lambda2cdf = zero)
!                    2 = maximum number of iterations reached
!                        (then lambda2cdf = value at last iteration,
!                                or zero)
!                    3 = required accuracy cannot be reached
!                        (then lambda2cdf = value at last iteration)
!                    4 = error in auxiliary function
!                        (chi2cdf or phi2cdf)
!
!    External functions called:
!      CHI2CDF  PHI2CDF
!      DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!    Fortran functions called:
!      ABS  EXP  LOG  MAX  NINT  (and DLGAMA if available)
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: lambda2cdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, p, q, a2 , delta
   integer, intent(in) :: maxitr
   integer, intent(out) :: ier
```

```
    !  Local declarations
    !  -----------------

   !! real(kind=8), external :: dlgama
    real(kind=8), external :: chi2cdf, phi2cdf

    real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
    real(kind=8), parameter :: qlimit=1.0e5_8, cinf=1.0e-20_8
    real(kind=8), parameter :: dflimit=1.0e6_8
    real(kind=8), parameter :: eps=0.223e-15_8, tinyr=1.0e-307_8, explower=-706.893_8
    real(kind=8), parameter :: relerr=1.0e-14_8 ! Relative error assumed
                                          ! in recurrence calculations
    real(kind=8), parameter :: xp=tinyr/relerr
    integer, parameter :: jmax=200

    !     dflimit = limit for approximation in chi2cdf function
    !     These constants are machine dependent:
    !     eps   = machine epsilon
    !           (the smallest real such that 1.0 + eps > 1.0)
    !     tinyr = the smallest positive real
    !     explower = minimum valid argument for the exponential function
    !           (i.e. log(tinyr))

    real(kind=8) :: aqal, bl, dj, dj1, erp, err, ga, gl, g0,  &
                    kgj, kgl, kx,                             &
                    px2, px2l,                                &
                    p2, qqal, q2, q2l, rl,                     &
                    sum, sumg
    integer :: j, ja, jjj, jz, j0

    !---------------------------------------------------------------

    lambda2cdf = zero
    ier = 0

    !  Test for valid input arguments

    if ( x < zero .or. a2 < zero .or. p <= zero .or. q <= zero  &
      .or. delta >= one .or. delta <= eps ) then
      ier = 1
      return
    end if

    !  Case x = 0

    if ( x < eps ) return

    !  Case a2 = 0

    if ( a2 < eps ) then
      lambda2cdf = chi2cdf( p*x, p, dflimit, ier )
      if ( ier /= 0 ) ier = 4
      return
```

```
end if

! If q is large enough use limiting distribution

if ( q > qlimit ) then
  lambda2cdf = phi2cdf( x, p, a2, delta, maxitr, ier )
  if ( ier /= 0 ) ier = 4
  return
end if

! General case (iterations)
! kx's are decreasing for all j's
! gl's are decreasing for j >= jjj = a2/2 - a2/q - 1
! kgj, sumg are only used for stopping rule
! Logs are used to avoid underflows

err  = delta * half
erp  = err / relerr
p2   = p * half
px2  = p2 * x
px2l = log( px2 )
q2   = q * half
qqal = q2 * log( q/(q+a2) )
aqal = log( a2/(q+a2) )
q2l  = dlgama( q2 )
jjj  = nint( a2*half - a2/q - half )
jjj  = max( jjj, 0 )

! Examine rate of convergence:
! find starting index for iterations (j0)
! cinf is a limit under which sum of g coefficients is
! considered as negligible
! g0 is gl at j0

j0 = 0
g0 = qqal
if ( jjj >= jmax ) then
  if ( g0 < log(cinf) ) then
    ga = qqal - q2l + dlgama(jjj+q2) - dlgama(jjj+one) + jjj*aqal
    if ( ga < explower )  goto 5   ! Cannot be computed
    if ( exp(ga) < cinf ) goto 5   ! Cannot be computed
    ja = 0
    jz = jjj
    ga = g0
    do
      j0 = (ja+jz) / 2
      g0 = qqal - q2l + dlgama(j0+q2) - dlgama(j0+one) + j0*aqal
      if ( g0 >= explower ) then
        if ( (j0+1)*exp(g0) < cinf ) then
          ja = j0
          ga = g0
        else
```

```
            jz = j0
          end if
        else
          ja = j0
          ga = g0
        end if
        if ( ja+1 >= jz ) exit
      end do
      j0 = ja
      g0 = ga
    end if
  end if
  5 continue

  ! Initialize

  ! when j0 > 0, it is assumed sumg at j0 is negligible,
  ! and the upper bound is taken for kgj:
  ! kgj = sum( j*g*kx ) <= sum( j*g )  (as kx <= 1)
  !                     <= exp(g0)*sum( j )    (as j <= j0 < jjj)
  !                     <= exp(G0)*J0*(J0+1)/2

  dj = j0
  gl = g0
  kx = chi2cdf( p*x, p+dj+dj, dflimit, ier )
  if ( ier /= 0 ) then
    ier = 4
    return
  end if
  rl  = (p2+dj) * px2l - ( px2 + dlgama(p2+dj+one) )
  kgj = zero
  if ( j0 > 0 ) then
    bl = dlgama(q2+dj) - q2l - dlgama(dj+one)
    if ( g0 >= explower ) kgj = (dj+one) * dj * half * exp(g0)
  else
    bl = zero
  end if
  sum  = zero
  sumg = zero

  ! Iteration loop

  do j = j0, j0+maxitr

    if ( kx > zero ) then
      kgl = log(kx) + gl
      if ( kgl >= explower ) then
        kgl = exp( kgl )
        sum = sum + kgl
        kgj = kgj + kgl*dj
        ! check loss of accuracy
        if ( kgj >= erp ) then
```

```
            ier = 3
            lambda2cdf = sum
            return
          end if
        end if
      end if

      lambda2cdf = sum

      ! Check accuracy (stopping rule)
      ! xp is used to prevent possible underflow

      if ( gl >= explower ) sumg = sumg + exp(gl)
      if ( kgj >= xp ) then
        if ( relerr*kgj+kx*(one-sumg) < err ) return
      else
        if (            kx*(one-sumg) < err ) return
      end if

      ! Prepare next iteration

      dj1   = j + 1
      bl    = bl + log( (q2+dj)/dj1 )
      gl    = bl + qqal + dj1*aqal
      if ( rl >= explower ) kx = kx - exp( rl )
      rl = rl - log(p2+dj1) + px2l
      dj = dj1
      !!* autre solution: sans bl: recurrence "directe" sur gl
      !!* inconvenient: si aqal tres faible devant les autres termes
      !!* (tel qu'ayant contribution exactement 0), ca ne peux pas etre
      !!* compense par dj1*aqal
      !**      dj1   = j + 1
      !**      gl    = gl + log( (q2+dj)/dj1 ) + aqal
      !**      if ( rl >= explower ) kx = kx - exp( rl )
      !**      rl = rl - log(p2+dj1) + px2l
      !**      dj = dj1

    end do

    ! Maximum number of iterations is reached

    ier = 2

  end function lambda2cdf
```

Chapter 19
Normal distribution

Function ncdf(x)
File : Ncdf.f90

```
function ncdf( x )

!------------------------------------------------------------------
!
!    Returns the probability that a random variable distributed
!    according to the Normal distribution with zero mean and unit
!    variance, is less than or equal to x.
!
!    X    - Input . Argument                          - Real
!
!    Fortran  functions called:
!       ABS
!
!    Calculation is based upon the error function, using Chebyshev
!    approximation over the interval (0, 6.09).
!
!------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: ncdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x

   ! Local declarations
   ! ------------------

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
   real(kind=8), parameter :: b=6.09_8
   real(kind=8), parameter :: epsl=1.0e-15_8
   real(kind=8), parameter :: sq2=0.70710678118654748_8  ! = 1/sqrt(2)
   integer, parameter :: m=43

   real(kind=8) :: ax, d, dd, sv, y, y2, z
   integer :: j
   ! Chebyshev coefficients for the error function:
   real(kind=8) :: c(m)=(/                                    &
               0.1635454272828649e+01_8, 0.3340345052068802e+00_8, &
              -0.2550158252490571e+00_8, 0.1576322031081386e+00_8, &
              -0.7292374105725899e-01_8, 0.1844981586051652e-01_8, &
               0.5451830906274930e-02_8,-0.9378172345987997e-02_8, &
               0.5461774234132427e-02_8,-0.1369434835987556e-02_8, &
              -0.4791673249620534e-03_8, 0.6448576838517482e-03_8, &
              -0.2787148092871001e-03_8, 0.1375372712760661e-04_8, &
               0.5421909937317043e-04_8,-0.3178226588449582e-04_8, &
               0.4982720716690025e-05_8, 0.3910358552338792e-05_8, &
              -0.2794516331451236e-05_8, 0.5298695171317318e-06_8, &
```

```
                0.2745633118680364e-06_8,-0.2071635016913643e-06_8, &
                0.3687897944251272e-07_8, 0.1967578278235985e-07_8, &
               -0.1324655993569593e-07_8, 0.1759666490397310e-08_8, &
                0.1371495927659758e-08_8,-0.7222588259009134e-09_8, &
                0.4293513190125905e-10_8, 0.8604672660411214e-10_8, &
               -0.3240778980162313e-10_8,-0.1531203125775596e-11_8, &
                0.4585991808162509e-11_8,-0.1112803317375455e-11_8, &
               -0.2501311672927168e-12_8, 0.1996065316425628e-12_8, &
               -0.2333998072735281e-13_8,-0.1670821608490672e-13_8, &
                0.6679068635371678e-14_8, 0.2100528246172614e-15_8, &
               -0.7805650505799361e-15_8,-0.2386236005000803e-15_8, &
               -0.1792153097738274e-15_8 /)

  !------------------------------------------------------------------

  if ( abs(x) < epsl ) then
    ncdf = half
    return
  end if

  z  = x * sq2
  ax = abs( z )
  if ( ax < b ) then
    d  = zero
    dd = zero
    y  = (ax+ax-b)/b
    y2 = y + y
    do j = m, 2, -1
      sv = d
      d  = y2*d - dd + c(j)
      dd = sv
    end do
    ncdf = half + half*(y*d-dd+half*c(1))
  else
    ncdf = one
  end if
  if ( x < zero ) ncdf = one - ncdf

end function ncdf
```

Chapter 20
Normal distribution (other)

Subroutine ndtri1(p, x, ier)
File : Ndtri1.f90

```
subroutine ndtri1( p, x, ier )

!------------------------------------------------------------------
!
!    Computes X = p**(-1)(P), the argument X such that P = p(X) =
!    the probability that the random variable u, distributed
!    normally(0,1), is less than or equal to X.
!    From IBM-SSP subroutine NDTRI
!
!    P     - Input . Probability              (0 < P < 1) - Real
!    X     - Output. Value of the variable                - Real
!    IER   - Output. Return code :                        - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then X = 0.)
!
!------------------------------------------------------------------

  ! Arguments
  ! ---------

  real, intent(in) :: p
  real, intent(out) :: x
  integer, intent(out) :: ier

  ! local declarations
  ! ------------------

  real(kind=8), parameter :: zero=0.0, half=0.5, one=1.0
  real :: c, t, t2

  !-----------------------------------------------------------------

  ! Test for valid input arguments

  if ( p <= zero .or. p >= one ) then
    ier = 1
    x = zero
    return
  end if

  ier = 0

  c  = p
  if ( c > half ) c = one - c
  t2 = log( one/(c*c) )
  t  = sqrt(t2)
  x  = t - (2.515517e0+0.802853e0*t+0.010328e0*t2)/              &
           (1.0e0+1.432788e0*t+0.189269e0*t2+0.001308e0*t*t2)
  if ( p < half ) x = -x

end subroutine ndtri1
```

Chapter 21
Normal - Error function)

Function erfun(x)
File : erfun.f90

```
function erfun( x )

!------------------------------------------------------------------
!    Returns the error function
!
!    X     - Input . Argument                          - Real
!
!------------------------------------------------------------------
!
!  The main computation evaluates near-minimax approximations
!  from "Rational Chebyshev approximations for the error function"
!  by W. J. Cody, Math. Comp., 1969, PP. 631-638.  This
!  transportable program uses rational functions that theoretically
!  approximate  erf(x)  and  erfc(x)  to at least 18 significant
!  decimal digits.  The accuracy achieved depends on the arithmetic
!  system, the compiler, the intrinsic functions, and proper
!  selection of the machine-dependent constants.
!
!*******************************************************************
!
! Explanation of machine-dependent constants
!
!  XMIN   = the smallest positive floating-point number.
!  XINF   = the largest positive finite floating-point number.
!  XNEG   = the largest negative argument acceptable to ERFCX;
!           the negative of the solution to the equation
!           2*exp(x*x) = XINF.
!  XSMALL = argument below which erf(x) may be represented by
!           2*x/sqrt(pi)  and above which  x*x  will not underflow.
!           A conservative value is the largest machine number X
!           such that   1.0 + X = 1.0   to machine precision.
!  XBIG   = largest argument acceptable to ERFC;  solution to
!           the equation:  W(x) * (1-0.5/x**2) = XMIN,  where
!           W(x) = exp(-x*x)/[x*sqrt(pi)].
!  XHUGE  = argument above which  1.0 - 1/(2*x*x) = 1.0  to
!           machine precision.  A conservative value is
!           1/[2*sqrt(XSMALL)]
!  XMAX   = largest acceptable argument to ERFCX; the minimum
!           of XINF and 1/[sqrt(pi)*XMIN].
!
!  Approximate values for some important machines are:
!
!                        XMIN       XINF        XNEG     XSMALL
!
! CDC 7600      (S.P.)  3.13E-294   1.26E+322   -27.220  7.11E-15
! CRAY-1        (S.P.)  4.58E-2467  5.45E+2465  -75.345  7.11E-15
! IEEE (IBM/XT,
!   SUN, etc.)  (S.P.)  1.18E-38    3.40E+38     -9.382  5.96E-8
! IEEE (IBM/XT,
!   SUN, etc.)  (D.P.)  2.23D-308   1.79D+308   -26.628  1.11D-16
! IBM 195       (D.P.)  5.40D-79    7.23E+75    -13.190  1.39D-17
! UNIVAC 1108   (D.P.)  2.78D-309   8.98D+307   -26.615  1.73D-18
```

```
! VAX D-Format  (D.P.)  2.94D-39    1.70D+38     -9.345  1.39D-17
! VAX G-Format  (D.P.)  5.56D-309   8.98D+307   -26.615  1.11D-16
!
!
!                       XBIG       XHUGE       XMAX
!
! CDC 7600      (S.P.)  25.922      8.39E+6     1.80X+293
! CRAY-1        (S.P.)  75.326      8.39E+6     5.45E+2465
! IEEE (IBM/XT,
!   SUN, etc.)  (S.P.)   9.194      2.90E+3     4.79E+37
! IEEE (IBM/XT,
!   SUN, etc.)  (D.P.)  26.543      6.71D+7     2.53D+307
! IBM 195       (D.P.)  13.306      1.90D+8     7.23E+75
! UNIVAC 1108   (D.P.)  26.582      5.37D+8     8.98D+307
! VAX D-Format  (D.P.)   9.269      1.90D+8     1.70D+38
! VAX G-Format  (D.P.)  26.569      6.71D+7     8.98D+307
!
!*****************************************************************
!
! Intrinsic functions required are:
!     ABS, AINT, EXP
!
!  Author: W. J. Cody
!          Mathematics and Computer Science Division
!          Argonne National Laboratory
!          Argonne, IL 60439
!
!  Latest modification: March 19, 1990
!
!-----------------------------------------------------------------------

    implicit none

    !  Function
    !  --------

    real(kind=8) :: erfun

    !  Arguments
    !  ---------

    real(kind=8), intent(in) :: x

    !  Local declarations
    !  ------------------

    integer :: i
    real(kind=8) :: del, result, xden, xnum, y, ysq

      !  Mathematical constants:
     real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8, two=2.0_8, four=4.0_8, six-
ten=16.0_8
    real(kind=8), parameter :: sqrpi=5.6418958354775628695e-1_8, thresh=0.46875_8
```

```
  ! Machine-dependent constants:
real(kind=8), parameter :: xinf=1.79e308_8, xneg=-26.628_8, xsmall=1.11e-16_8
real(kind=8), parameter :: xbig=26.543_8, xhuge=6.71e7_8, xmax=2.53e307_8

  ! Coefficients for approximation to  erf  in first interval:
real(kind=8), dimension(5) :: a=(/                              &
        3.16112374387056560_8,   1.13864154151050156e02_8, &
        3.77485237685302021e02_8,3.20937758913846947e03_8, &
        1.85777706184603153e-1_8/)
real(kind=8), dimension(4) :: b=(/                              &
        2.36012909523441209e01_8,2.44024637934444173e02_8, &
        1.28261652607737228e03_8,2.84423683343917062e03_8/)

  ! Coefficients for approximation to  erfc  in second interval:
real(kind=8), dimension(9) :: c=(/                              &
        5.64188496988670089e-1_8,8.88314979438837594_8,    &
        6.61191906371416295e01_8,2.98635138197400131e02_8, &
        8.81952221241769090e02_8,1.71204761263407058e03_8, &
        2.05107837782607147e03_8,1.23033935479799725e03_8, &
        2.15311535474403846e-8_8/)
real(kind=8), dimension(8) :: d=(/                              &
        1.57449261107098347e01_8,1.17693950891312499e02_8, &
        5.37181101862009858e02_8,1.62138957456669019e03_8, &
        3.29079923573345963e03_8,4.36261909014324716e03_8, &
        3.43936767414372164e03_8,1.23033935480374942e03_8/)

  ! Coefficients for approximation to  erfc  in third interval:
real(kind=8), dimension(6) :: p=(/                              &
        3.05326634961232344e-1_8,3.60344899949804439e-1_8, &
        1.25781726111229246e-1_8,1.60837851487422766e-2_8, &
        6.58749161529837803e-4_8,1.63153871373020978e-2_8/)
real(kind=8), dimension(5) :: q=(/                              &
        2.56852019228982242_8,   1.87295284992346047_8,    &
        5.27905102951428412e-1_8,6.05183413124413191e-2_8, &
        2.33520497626869185e-3_8/)

!------------------------------------------------------------------

y = abs(x)

if ( y <= thresh ) then
     !------------------------------------!
     ! Evaluate  erf  for  |X| <= 0.46875  !
     !------------------------------------!
   ysq = zero
   if ( y > xsmall ) ysq = y * y
   xnum = a(5)*ysq
   xden = ysq
   do i = 1, 3
      xnum = (xnum + a(i)) * ysq
      xden = (xden + b(i)) * ysq
   end do
```

```
      erfun = x * (xnum + a(4)) / (xden + b(4))
      return

   else if ( y <= four ) then
         !------------------------------------------!
         ! Evaluate  erfc  for 0.46875 <= |X| <= 4.0  !
         !------------------------------------------!
      xnum = c(9)*y
      xden = y
      do i = 1, 7
         xnum = (xnum + c(i)) * y
         xden = (xden + d(i)) * y
      end do
      result = (xnum + c(8)) / (xden + d(8))
      ysq = aint(y*sixten)/sixten
      del = (y-ysq)*(y+ysq)
      result = exp(-ysq*ysq) * exp(-del) * result

   else
         !---------------------------------!
         ! Evaluate  erfc  for |X| > 4.0  !
         !---------------------------------!
      result = zero
      if ( y >= xbig ) then
         erfun = (half - result) + half
         if ( x < zero ) erfun = -erfun
         return
      end if
      ysq = one / (y * y)
      xnum = p(6)*ysq
      xden = ysq
      do i = 1, 4
         xnum = (xnum + p(i)) * ysq
         xden = (xden + q(i)) * ysq
      end do
      result = ysq *(xnum + p(5)) / (xden + q(5))
      result = (sqrpi -  result) / y
      ysq = aint(y*sixten)/sixten
      del = (y-ysq)*(y+ysq)
      result = exp(-ysq*ysq) * exp(-del) * result
   end if

      !-------------------------------!
      ! Fix up for negative argument  !
      !-------------------------------!
   erfun = (half - result) + half
   if ( x < zero ) erfun = -erfun

end function erfun
```

Chapter 22
Normal - Complement to one of the error function

Function function erfunc(x)
File : erfunc.f90

```
function erfunc( x )

!------------------------------------------------------------------
!    Returns the complement to one of the error function
!
!    X     - Input . Argument                          - Real
!
!------------------------------------------------------------------
!
!  The main computation evaluates near-minimax approximations
!  from "Rational Chebyshev approximations for the error function"
!  by W. J. Cody, Math. Comp., 1969, PP. 631-638.  This
!  transportable program uses rational functions that theoretically
!  approximate  erf(x)  and  erfc(x)  to at least 18 significant
!  decimal digits.  The accuracy achieved depends on the arithmetic
!  system, the compiler, the intrinsic functions, and proper
!  selection of the machine-dependent constants.
!
!*******************************************************************
!
! Explanation of machine-dependent constants
!
!  XMIN   = the smallest positive floating-point number.
!  XINF   = the largest positive finite floating-point number.
!  XNEG   = the largest negative argument acceptable to ERFCX;
!           the negative of the solution to the equation
!           2*exp(x*x) = XINF.
!  XSMALL = argument below which erf(x) may be represented by
!           2*x/sqrt(pi)  and above which  x*x  will not underflow.
!           A conservative value is the largest machine number X
!           such that   1.0 + X = 1.0   to machine precision.
!  XBIG   = largest argument acceptable to ERFC;  solution to
!           the equation:  W(x) * (1-0.5/x**2) = XMIN,  where
!           W(x) = exp(-x*x)/[x*sqrt(pi)].
!  XHUGE  = argument above which  1.0 - 1/(2*x*x) = 1.0  to
!           machine precision.  A conservative value is
!           1/[2*sqrt(XSMALL)]
!  XMAX   = largest acceptable argument to ERFCX; the minimum
!           of XINF and 1/[sqrt(pi)*XMIN].
!
!  Approximate values for some important machines are:
!
!                        XMIN       XINF        XNEG     XSMALL
!
! CDC 7600      (S.P.)  3.13E-294   1.26E+322   -27.220  7.11E-15
! CRAY-1        (S.P.)  4.58E-2467  5.45E+2465  -75.345  7.11E-15
! IEEE (IBM/XT,
!   SUN, etc.)  (S.P.)  1.18E-38    3.40E+38     -9.382  5.96E-8
! IEEE (IBM/XT,
!   SUN, etc.)  (D.P.)  2.23D-308   1.79D+308   -26.628  1.11D-16
! IBM 195       (D.P.)  5.40D-79    7.23E+75    -13.190  1.39D-17
! UNIVAC 1108   (D.P.)  2.78D-309   8.98D+307   -26.615  1.73D-18
```

```
! VAX D-Format  (D.P.)  2.94D-39    1.70D+38     -9.345  1.39D-17
! VAX G-Format  (D.P.)  5.56D-309   8.98D+307   -26.615  1.11D-16
!
!
!                       XBIG       XHUGE       XMAX
!
! CDC 7600      (S.P.)  25.922      8.39E+6     1.80X+293
! CRAY-1        (S.P.)  75.326      8.39E+6     5.45E+2465
! IEEE (IBM/XT,
!   SUN, etc.)  (S.P.)   9.194      2.90E+3     4.79E+37
! IEEE (IBM/XT,
!   SUN, etc.)  (D.P.)  26.543      6.71D+7     2.53D+307
! IBM 195       (D.P.)  13.306      1.90D+8     7.23E+75
! UNIVAC 1108   (D.P.)  26.582      5.37D+8     8.98D+307
! VAX D-Format  (D.P.)   9.269      1.90D+8     1.70D+38
! VAX G-Format  (D.P.)  26.569      6.71D+7     8.98D+307
!
!*****************************************************************
!
! Error returns
!  The program returns  ERFC = 0      for  ARG .GE. XBIG;
!
! Intrinsic functions required are:
!     ABS, AINT, EXP
!
!  Author: W. J. Cody
!          Mathematics and Computer Science Division
!          Argonne National Laboratory
!          Argonne, IL 60439
!
!  Latest modification: March 19, 1990
!
!------------------------------------------------------------------

    implicit none

    ! Function
    ! --------

    real(kind=8) :: erfunc

    ! Arguments
    ! ---------

    real(kind=8), intent(in) :: x

    ! Local declarations
    ! ------------------

    integer :: i
    real(kind=8) :: del, result, xden, xnum, y, ysq

      ! Mathematical constants:
```

```
 real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8, two=2.0_8, four=4.0_8, six-
ten=16.0_8
    real(kind=8), parameter :: sqrpi=5.6418958354775628695e-1_8, thresh=0.46875_8

      ! Machine-dependent constants:
    real(kind=8), parameter :: xinf=1.79e308_8, xneg=-26.628_8, xsmall=1.11e-16_8
    real(kind=8), parameter :: xbig=26.543_8, xhuge=6.71e7_8, xmax=2.53e307_8

      ! Coefficients for approximation to  erf  in first interval:
    real(kind=8), dimension(5) :: a=(/                              &
            3.16112374387056560_8,   1.1386415415105 0156e02_8, &
            3.77485237685302021e02_8,3.20937758913846947e03_8, &
            1.85777706184603153e-1_8/)
    real(kind=8), dimension(4) :: b=(/                              &
            2.36012909523441209e01_8,2.44024637934444173e02_8, &
            1.28261652607737228e03_8,2.84423683343917062e03_8/)

      ! Coefficients for approximation to  erfc  in second interval:
    real(kind=8), dimension(9) :: c=(/                              &
            5.64188496988670089e-1_8,8.88314979438837594_8,     &
            6.61191906371416295e01_8,2.98635138197400131e02_8, &
            8.81952221241769090e02_8,1.71204761263407058e03_8, &
            2.05107837782607147e03_8,1.23033935479799725e03_8, &
            2.15311535474403846e-8_8/)
    real(kind=8), dimension(8) :: d=(/                              &
            1.57449261107098347e01_8,1.17693950891312499e02_8, &
            5.37181101862009858e02_8,1.62138957456669019e03_8, &
            3.29079923573345963e03_8,4.36261909014324716e03_8, &
            3.43936767414372164e03_8,1.23033935480374942e03_8/)

      ! Coefficients for approximation to  erfc  in third interval:
    real(kind=8), dimension(6) :: p=(/                              &
            3.05326634961232344e-1_8,3.60344899949804439e-1_8, &
            1.25781726111229246e-1_8,1.60837851487422766e-2_8, &
            6.58749161529837803e-4_8,1.63153871373020978e-2_8/)
    real(kind=8), dimension(5) :: q=(/                              &
            2.56852019228982242_8,   1.87295284992346047_8,     &
            5.27905102951428412e-1_8,6.05183413124413191e-2_8, &
            2.33520497626869185e-3_8/)

    !------------------------------------------------------------------

    y = abs(x)

    if ( y <= thresh ) then
       !------------------------------------!
       ! Evaluate  erf  for  |X| <= 0.46875  !
       !------------------------------------!
      ysq = zero
      if ( y > xsmall ) ysq = y * y
      xnum = a(5)*ysq
      xden = ysq
```

```
    do i = 1, 3
      xnum = (xnum + a(i)) * ysq
      xden = (xden + b(i)) * ysq
    end do
    result = x * (xnum + a(4)) / (xden + b(4))
    erfunc = one - result
    return

  else if ( y <= four ) then
      !---------------------------------------------!
      ! Evaluate  erfc  for 0.46875 <= |X| <= 4.0  !
      !---------------------------------------------!
    xnum = c(9)*y
    xden = y
    do i = 1, 7
      xnum = (xnum + c(i)) * y
      xden = (xden + d(i)) * y
    end do
    result = (xnum + c(8)) / (xden + d(8))
    ysq = aint(y*sixten)/sixten
    del = (y-ysq)*(y+ysq)
    result = exp(-ysq*ysq) * exp(-del) * result

  else
      !---------------------------------!
      ! Evaluate  erfc  for |X| > 4.0  !
      !---------------------------------!
    result = zero
    if ( y >= xbig ) then
      erfunc = result
      if ( x < zero ) erfunc = two - erfunc
      return
    end if
    ysq = one / (y * y)
    xnum = p(6)*ysq
    xden = ysq
    do i = 1, 4
      xnum = (xnum + p(i)) * ysq
      xden = (xden + q(i)) * ysq
    end do
    result = ysq *(xnum + p(5)) / (xden + q(5))
    result = (sqrpi -  result) / y
    ysq = aint(y*sixten)/sixten
    del = (y-ysq)*(y+ysq)
    result = exp(-ysq*ysq) * exp(-del) * result
  end if

    !--------------------------------!
    ! Fix up for negative argument  !
    !--------------------------------!
  erfunc = result
  if ( x < zero ) erfunc = two - erfunc
```

```
end function erfunc
```

Chapter 23
Normal - Error function (subroutine)

Subroutine calerf(x, result, jint)
File : erf.f90

```
!!!
```

Chapter 24
Phi-square distribution)

Function phi2cdf(x, p, a2, delta, maxitr, ier)
File : Phi2cdf.f90

```
function phi2cdf( x, p, a2, delta, maxitr, ier )

!------------------------------------------------------------------------
!
!    Calculates the probability that a random variable distributed
!    according to the phi-square distribution with P degrees of
!    freedom and A2 eccentricity parameter, is less than or equal to X
!
!    X     - Input . Value of the variable          (X >= 0) - Real
!    P     - Input . Degrees of freedom              (P >  0) - Real
!    A2    - Input . Eccentricity parameter          (A2>= 0) - Real
!    DELTA - Input . Maximum absolute error required on      - Real
!                    phi2cdf (stopping criterion)
!                    (eps < DELTA < 1 where eps is machine
!                    epsilon; see parameter statement below)
!    MAXITR- Input . Maximum number of iterations            - Integer
!    IER   - Output. Return code :                           - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then phi2cdf = zero)
!                    2 = maximum number of iterations reached
!                        (then phi2cdf = value at last iteration,
!                                or zero)
!                    3 = required accuracy cannot be reached
!                        (then phi2cdf = value at last iteration)
!                    4 = error in auxiliary function (chi2cdf)
!
!    External functions called:
!      CHI2CDF
!      DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!    Fortran functions called:
!      ABS  EXP  LOG  INT  (and DLGAMA if available)
!
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: phi2cdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, p, a2 , delta
   integer, intent(in) :: maxitr
   integer, intent(out) :: ier

   ! local declarations
```

```
    !  ------------------

   !! real(kind=8), external :: dlgama
      real(kind=8), external :: chi2cdf

      real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
      real(kind=8), parameter :: erraux=1.0e-14_8, dflimit=1.0e6_8
      real(kind=8), parameter :: eps=0.223e-15_8, tinyr=1.0e-307_8, explower=-706.893_8
      real(kind=8), parameter :: relerr=1.0e-14_8 ! Relative error assumed
                                                  ! in recurrence calculations
      real(kind=8), parameter :: xp=tinyr/relerr

      !     erraux = relative error in chi2cdf function
      !     dflimit = limit for approximation in chi2cdf function
      !     These constants are machine dependent:
      !     eps    = machine epsilon
      !            (the smallest real such that 1.0 + eps > 1.0)
      !     tinyr = the smallest positive real
      !     explower = minimum valid argument for the exponential function
      !            (i.e. log(tinyr))

      real(kind=8) :: a22, a22l, dj, erp, err, gl,  &
                      kgj, kgl, kx,                 &
                      px2, px2l,                    &
                      p2, rl,                      &
                      sum, sumg
      integer :: j

      !---------------------------------------------------------------

      phi2cdf = zero
      ier = 0

      !  Test for valid input arguments

      if ( x < zero .or. a2 < zero .or. p <= zero     &
        .or. delta >= one .or. delta <= eps ) then
        ier = 1
        return
      end if

      !  Case x = 0

      if ( x < eps ) return

      !  Case a2 = 0

      if ( a2 < eps ) then
        phi2cdf = chi2cdf( p*x, p, dflimit, ier )
        if ( ier /= 0 ) then
          ier = 4
        else
          if ( phi2cdf*erraux > delta ) ier = 3
```

```
   end if
   return
end if

! General case (iterations)
! kx's are decreasing for all j's
! gl's are decreasing for j > a2/2
! kgj, sumg are only used for stopping rule
! Logs are used to avoid underflows

err  = delta * half
erp  = err / relerr
p2   = p * half
px2  = p2 * x
px2l = log( px2 )
a22  = a2 * half
a22l = log( a22 )

! Initialize

gl = -a22
kx = chi2cdf( p*x, p, dflimit, ier )
if ( ier /= 0 ) then
   ier = 4
   return
else if ( kx*erraux > delta ) then
   ier = 3
   return
end if
rl   = p2*px2l - ( px2 + dlgama(p2+one) )
kgj  = zero
sum  = zero
sumg = zero
dj   = zero

! Iteration loop

do j = 0, maxitr

   if ( kx > zero ) then
      kgl = log(kx) + gl
      if ( kgl >= explower ) then
         kgl = exp( kgl )
         sum = sum + kgl
         kgj = kgj + kgl*dj
         ! check loss of accuracy
         if ( kgj >= erp ) then
            ier = 3
            phi2cdf = sum
            return
         end if
      end if
```

```
      end if

      phi2cdf = sum

      ! Check accuracy (stopping rule)
      ! xp is used to prevent possible underflow

      if ( gl >= explower ) sumg = sumg + exp(gl)
      if ( kgj >= xp ) then
        if ( relerr*kgj+kx*(one-sumg) < err ) return
      else
        if (             kx*(one-sumg) < err ) return
      end if

      ! Prepare next iteration

      dj = j + 1
      gl = gl + a22l - log(dj)
      if ( rl >= explower ) kx = kx - exp( rl )
      rl = rl - log(p2+dj) + px2l

    end do

    ! Maximum number of iterations is reached

    ier = 2

  end function phi2cdf
```

Chapter 25
Continuous Poisson distribution)

Function poissonccdf(x, p, ier)
File : poissonccdf.f90

```
function poissonccdf( x, p, ier )

!-----------------------------------------------------------------------
!
!     Computes the probability that a random variable distributed
!     according to the continuous poisson distribution with parameter P,
!     is less than or equal to X.
!
!     X     - Input . Value of the variable          (X >=-1)  - Real
!     P     - Input . Parameter                     (P >= 0)  - Real
!     IER   - Output. Return code :                            - Integer
!                   0 = normal
!                   1 = invalid input argument
!                       (then POISSONCCDF = 0.)
!
!     External functions called:
!       POISSONCDF  POISSONPMF
!
!     Fortran functions called:
!       INT
!
!-----------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: poissonccdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: p
   integer, intent(in) :: x
   integer, intent(out) :: ier

   ! Local declarations
   ! ------------------

   real(kind=8), external :: poissoncdf, poissonpmf

   real(kind=8), parameter :: zero=0.0_8, one=1.0_8

   real(kind=8) :: a, b
   integer :: y

   !-----------------------------------------------------------------------

   poissonccdf = zero
   if ( x < -one .or. p < zero ) then  ! Test for valid input arguments
      ier = 1
      return
```

```
    end if

    ier = 0
    y = int(x)
    a = poissoncdf( y, p, ier )
    if ( ier /= 0 ) return
    b = poissonpmf( y+1, p, ier )
    if ( ier /= 0 ) return
    poissonccdf = a + (x-y)*b

  end function poissonccdf
```

Chapter 26
Fiducial continued Poisson distribution)

Function poissoncfcdf(x, a, u, ier)
File : poissoncfcdf.f90

```
function poissoncfcdf( x, a, u, ier )

!------------------------------------------------------------------------
!
!     Computes the probability that a random variable distributed
!     according to the fiducial continued poisson distribution with
!     parameters A, U is less than or equal to X.
!
!     X     - Input . Value of the variable          (X >= 0)  - Real
!     A     - Input . parameter: number of "positive" events  - Real
!                   (A > 0)
!     U     - Input . parameter                                - Real
!                   (0 <= U <= 1)
!     IER   - Output. Return code :                             - Integer
!                   0 = normal
!                   1 = invalid input argument
!                       (then POISSONCFCDF = 0.)
!

!     External functions called:
!       POISSONCCDF
!
!     Fortran functions called:
!       INT
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: poissoncfcdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, a, u
   integer, intent(out) :: ier

   ! Local declarations
   ! ------------------

   real(kind=8), external :: poissonccdf

   real(kind=8), parameter :: zero=0.0_8, one=1.0_8

   !------------------------------------------------------------------------

   poissoncfcdf = zero
   ! Test for valid input arguments
   if ( a < zero .or. u < zero .or. x < zero ) then
      ier = 1
```

```
      return
    end if

    ier = 0
    poissoncfcdf = one - poissonccdf( a-u, x, ier )

  end function poissoncfcdf
```

Chapter 27
Psi-square distribution)

Function psi2cdf(x, p, q, a2, delta, maxitr, iap, ier)
File : Psi2cdf.f90

```
function psi2cdf( x, p, q, a2, delta, maxitr, iap, ier )

!------------------------------------------------------------------------
!
!    Computes the probability that a random variable distributed
!    according to the psi-square distribution with P and Q degrees
!    of freedom and A2 eccentricity parameter, is less than or
!    equal to X.
!
!    X     - Input . Value of the variable          (X >= 0) - Real
!    P     - Input . First degrees of freedom       (P >  0) - Real
!    Q     - Input . Second   "    "     "          (Q >  0) - Real
!    A2    - Input . Eccentricity parameter         (A2>= 0) - Real
!    DELTA - Input . Maximum absolute error required on      - Real
!                    psi2cdf (stopping criterion)
!                    (eps < DELTA < 1 where eps is machine
!                    epsilon; see parameter statement below)
!    MAXITR- Input . Maximum number of iterations            - Integer
!                    (2000 is sufficient for most cases)
!    IAP   - Output. 0 if no approximation used              - Integer
!                    1 otherwise
!    IER   - Output. Return code :                           - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then psi2cdf = zero)
!                    2 = maximum number of iterations reached
!                        (then psi2cdf = value at last iteration,
!                                or zero)
!                    3 = required accuracy cannot be reached
!                        (then psi2cdf = value at last iteration)
!                    4 = error in auxiliary function
!                        (betacdf or phi2cdf)
!
!    External functions called:
!      BETACDF  PHI2CDF
!      DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!    Fortran functions called:
!      ABS  EXP  INT  LOG  MAX  NINT  SIGN  SQRT
!      (and DLGAMA if available)
!
!    Algorithm by J-L. Guigues (1981), improved by B. Lecoutre (1986)
!
!    J. Poitevineau (CNRS-URA1201)
!    January 01, 1991 (First version 26/04/85)
!    Modif.  23/09/93 (jjj)  25/11/93 (jj, zg)  10/08/94 (y1, phi2cdf)
!
!------------------------------------------------------------------------

  implicit none

  ! Function
```

```
! --------

real(kind=8) :: psi2cdf

! Arguments
! ---------

real(kind=8), intent(in) :: x, p, q, a2, delta
integer, intent(in) :: maxitr
integer, intent(out) :: iap, ier

! local declarations
! ------------------

!! real(kind=8), external :: dlgama
real(kind=8), external :: betacdf, phi2cdf

real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
real(kind=8), parameter :: two=one+one
real(kind=8), parameter :: qlimit=1.0e9_8
real(kind=8), parameter :: eps=0.223e-15_8, tinyr=1.0e-307_8, explower=-706.893_8
real(kind=8), parameter :: relerr=1.0e-14_8 ! Relative error assumed
                                            ! in recurrence calculations
real(kind=8), parameter :: xp=tinyr/relerr
real(kind=8), parameter :: xmxint=2147483647.0_8

!     These constants are machine dependent:
!     eps   = machine epsilon
!             (the smallest real such that 1.0 + eps > 1.0)
!     tinyr = the smallest positive real
!     explower = minimum valid argument for the exponential function
!             (i.e. log(tinyr))
!     xmxint= maximum positive integer

real(kind=8) :: aqal, bl, cl, dax, dj, dj1,  &
                erp, err, g, gcj, gcl,       &
                pq2, p2, qqal, q2, r, rl,    &
                sum, sumc,                   &
                v, w, y, yyl,                &
                y1, z, zg, zm, zn
integer :: iok, j, jj, jok
logical :: lalf

!-----------------------------------------------------------------

psi2cdf = zero
iap = 0
ier = 0

! Test for valid input arguments

if ( x < zero .or. a2 < zero .or. p <= zero .or. q <= zero  &
   .or. delta >= one .or. delta <= eps ) then
```

```
   ier  = 1
   return
end if

y  = p*x    / (a2+q+p*x)
y1 = (a2+q) / (a2+q+p*x)

!  y near 0 or 1

if ( y  <= tinyr ) return
if ( y1 <= tinyr ) then
   psi2cdf = one
   return
end if

!  Define usefull parameters

p2  = p * half
q2  = q * half
pq2 = p2 + q2

!  Case a2 = 0

if ( a2 < eps ) then
   psi2cdf = betacdf( p*x/(p*x+q), p2, q2, iok )
   if ( iok /= 0 ) ier = 4
   return
end if

!  Case p = 1

if ( abs(p-one) < eps ) then
   dax = two * sqrt( a2*x )
   psi2cdf = half * ( sign( one, x-a2 ) *                    &
            betacdf( (a2+x-dax)/(a2+x-dax+q),half,q2,iok ) + &
            betacdf( (a2+x+dax)/(a2+x+dax+q),half,q2,jok ) )
   if ( iok /= 0 .or. jok /= 0 ) ier = 4
   return
end if

! If q is large enough use limiting distribution

if ( q > qlimit ) then
   psi2cdf = phi2cdf( x, p, a2, delta, maxitr, ier )
   if ( ier /= 0 ) ier = 4
   iap = 1
   return
end if

!  Calculate 1-f(x) or f(x) (lalf is the indicator)

lalf = .false.
z    = y
```

```
zm   = p2
zn   = q2
if ( abs(y-half) < eps ) then
  ! Look for special case y = 0.5 and p = q
  if ( abs(p-q) < eps ) then
    psi2cdf = half
    return
  end if
  if ( p > q ) then
    lalf = .true.
    zm   = q2
    zn   = p2
  end if
else if ( y > half ) then
  lalf = .true.
  z    = y1
  zm   = q2
  zn   = p2
end if

! General case (iterations)
! g's  are decreasing for j >= jj = -v/w + 1
! cl's are decreasing for j >= jjj = a2/2 - a2/q - 1
! gcj, sumc, zg are only used for stopping rule
! Logs are used to avoid underflows

! Define constants

err  = delta * half
erp  = err / relerr
yyl  = log( y*y1 )
qqal = q2 * log( q/(q+a2) )
aqal = log( a2/(q+a2) )
v    = pq2*z - zn
w    = z + z - one

! Initialize

bl = zero
cl = qqal
g  = betacdf( z, zm, zn, iok )
if ( iok /= 0 ) then
  ier  = 4
  return
end if
zg = g
rl = p2*log(y) + q2*log(y1) + dlgama(pq2) - dlgama(p2) - dlgama(q2) - log(p2) - log(q2)
if ( rl >= explower ) then
  r = exp( rl )
else
  r = zero
end if
```

```
sum  = zero
gcj  = zero
sumc = zero

!  Define jj (and associated upper bound zg)
!  (note that w=0 if and only if y=0.5, and in such a case zm<zn)

if ( zm > zn ) then
  zg = -v/w
  if ( abs(zg) < xmxint ) then
    jj = int( zg ) + 1
    if ( jj > 0 ) then
      zg = betacdf( z, zm+jj, zn+jj, iok )
      if ( iok /= 0 ) then
        ier  = 4
        return
      end if
    end if
  else
    zg = one
    jj = maxitr + 1
  end if
else
  jj = 0
end if

dj = zero

!  Iteration loop

do j = 0, maxitr

  if ( g > zero ) then
    gcl = log(g) + cl
    if ( gcl >= explower ) then
      gcl = exp( gcl )
      sum = sum + gcl
      gcj = gcj + gcl*dj
      !  Check loss of accuracy
      if ( gcj >= erp ) then
        ier = 3
        goto 10
      end if
    end if
  end if

  !  Check accuracy (stopping rule)
  !  xp is used to prevent possible underflow

  if ( cl >= explower ) sumc = sumc + exp(cl)
  if ( j >= jj ) zg = g
  if ( gcj >= xp ) then
    if ( relerr*gcj+zg*(one-sumc) < err ) goto 10
```

```
      else
        if (            zg*(one-sumc) < err ) goto 10
      end if

      !  Prepare next iteration

      dj1 = j + 1
      bl  = bl + log( (q2+dj)/dj1 )
      cl  = bl + qqal + dj1*aqal
      g   = g  + (v+w*dj)*r
      rl  = rl + yyl + log( ((dj+dj+pq2)/(dj1+p2)) * ((dj+dj1+pq2)/(dj1+q2)) )
      if ( rl >= explower ) then
        r = exp( rl )
      else
        r = zero
      end if
      dj = dj1

    end do

    !  Maximum number of iterations is reached

    ier = 2

    !  The end

    10 continue
    if ( lalf ) then
      psi2cdf = one - sum
    else
      psi2cdf = sum
    end if

  end function psi2cdf
```

Chapter 28
Square of the multiple correlation coefficient distribution)

Function SQMCOR(X, IP, N, RHO2, IFAULT)
File : sqmcor.f90

```
      REAL FUNCTION SQMCOR(X, IP, N, RHO2, IFAULT)
      !
      !       ALGORITHM AS 260  APPL. STATIST. (1991) VOL. 40, NO. 1
      !
      !       Computes the C.D.F. for the distribution of the
      !       square of the multiple correlation coefficient
      !       with parameters IP, N, and RHO2
      !
      !       The following auxiliary algorithms are required:
      !       ALOGAM - Log-gamma function (CACM 291)
      !        (or  ALNGAM    AS 245)
      !       BETAIN - Incomplete beta function (AS 63)
      !
            REAL X, RHO2
            INTEGER IP, N, IFAULT
            REAL A, B, BETA, ERRBD, ERRMAX, GX, Q, SUMQ, TEMP, TERM, XJ,  &
                ZERO, HALF, ONE
            INTEGER ITRMAX
            REAL ALOGAM, BETAIN
            EXTERNAL ALOGAM, BETAIN
      !
            DATA ERRMAX, ITRMAX / 1.0E-6, 100 /
            DATA ZERO, HALF, ONE / 0.0, 0.5, 1.0 /
      !
            SQMCOR = X
            IFAULT = 2
            IF (RHO2 .LT. ZERO .OR. RHO2 .GT. ONE .OR. IP .LT. 2 .OR.    &
               N .LE. IP) RETURN
            IFAULT = 3
            IF (X .LT. ZERO .OR. X .GT. ONE) RETURN
            IFAULT = 0
            IF (X .EQ. ZERO .OR. X .EQ. ONE) RETURN
      !
            A = HALF * (IP - 1)
            B = HALF * (N - IP)
      !
      !       Initialize the series
      !
            BETA = EXP(ALOGAM(A, IFAULT) + ALOGAM(B, IFAULT) -   &
                  ALOGAM(A + B, IFAULT))
            TEMP = BETAIN(X, A, B, BETA, IFAULT)
      !
      !       There is no need to test IFAULT since all of the
      !       parameter values have already been checked
      !
            GX = EXP(A * LOG(X) + B * LOG(ONE - X) - LOG(A)) / BETA
            Q = (ONE - RHO2) ** (A + B)
            XJ = ZERO
            TERM = Q * TEMP
            SUMQ = ONE - Q
            SQMCOR = TERM
      !
```

```
!       Perform recurrence until convergence is achieved
!
  10 XJ = XJ + ONE
     TEMP = TEMP - GX
     GX = GX * (A + B + XJ - ONE) * X / (A + XJ)
     Q = Q * (A + B + XJ - ONE) * RHO2 / XJ
     SUMQ = SUMQ - Q
     TERM = TEMP * Q
     SQMCOR = SQMCOR + TERM
!
!       Check for convergence and act accordingly
!
     ERRBD = (TEMP - GX) * SUMQ
     IF ((INT(XJ) .LT. ITRMAX) .AND. (ERRBD .GT. ERRMAX)) GO TO 10
     IF (ERRBD .GT. ERRMAX) IFAULT = 1
     RETURN
     END
```

==========

The author of algorithms AS 260 & 261 has found an improved algorithm which has been published in the journal *Computational Statistics & Data Analysis*. Here it is.

```
!------------------------------------------------------------
!
! The following code for the distribution function of R^2 (cdfr2.for)
! and for its probability density (pdfr2.for) is by the author of AS 260
! and uses an improved algorithm.

      REAL FUNCTION CDFR2(Y, M, SIZE, RHO2, IFAULT)
!
!  Computes the distribution function of the square of the sample
!  multiple correlation coefficient for given abscissa Y, number
!  of random variables M, sample size SIZE, and square of the population
!  multiple correlation coefficient RHO2
!
!  Reference:
!  Ding, C.G. (1996) - On the computation of the distribution of
!  the square of the sample multiple correlation coefficient',
!  Computational Statistics & Data Analysis, 22, 345-350.
!
!  IFAULT is a fault indicator:
!  = 1 if any of the input values is illegal
!  = 0 otherwise
!
!  No auxiliary algorithm is required
!
      INTEGER SIZE
      REAL N
      DATA EPS / 1.0E-6 /
      DATA HALF, ZERO, ONE, TWO / 0.5, 0.0, 1.0, 2.0 /
      DATA RP / 1.772453850905516028 /

      CDFR2 = ZERO
```

```
      IFAULT = 1
      IF (M .LE. 1 .OR. SIZE .LE. M .OR. RHO2 .LT. ZERO .OR.  &
          RHO2 .GT. ONE) RETURN
      IFAULT = 0
      IF (Y .LE. ZERO) RETURN
      CDFR2 = ONE
      IF (Y .GE. ONE) RETURN
      A = (M - 1) / TWO
      B = (SIZE - M) / TWO
      AB = (SIZE - 1) / TWO
      IF (MOD(M + 1, 2) .EQ. 0) THEN
        NA = A + HALF
        GA = ONE
        DO 10 I = 1, NA
   10   GA = GA * I
      ELSE
        NA = A + ONE
        GA = RP
        DO 20 I = 1, NA
   20   GA = GA * (I - HALF)
      ENDIF
      IF (MOD(SIZE - M, 2) .EQ. 0) THEN
        NB = B - HALF
        GB = ONE
        IF (NB .EQ. 0) GO TO 50
        DO 30 I = 1, NB
   30   GB = GB * I
      ELSE
        NB = B
        GB = RP
        IF (NB .EQ. 0) GO TO 50
        DO 40 I = 1, NB
   40   GB = GB * (I - HALF)
      ENDIF
   50 IF (MOD(SIZE - 1, 2) .EQ. 0) THEN
        NAB = AB - HALF
        GAB = ONE
        IF (NAB .EQ. 0) GO TO 80
        DO 60 I = 1, NAB
   60   GAB = GAB * I
      ELSE
        NAB = AB
        GAB = RP
        DO 70 I = 1, NAB
   70   GAB = GAB * (I - HALF)
      ENDIF

!
!       Evaluate the first term
!
   80 N = ONE
      Q = (ONE - RHO2) ** AB
```

```
      V = Q
      T = Y ** A * (ONE - Y) ** B * GAB / GA / GB
      TERM = V * T
      CDFR2 = TERM
!
!       Check if a + n > (a + b + n)y
!
   90 IF (A + N .GT. (A + B + N) * Y) GO TO 100
!
!       Evaluate the next term of the expansion and then the
!       partial sum
!
      Q = Q * (A + B + N - ONE) * RHO2 / N
      V = V + Q
      T = T * Y * (A + B + N - ONE) / (A + N)
      TERM = V * T
      CDFR2 = CDFR2 + TERM
      N = N + ONE
      GO TO 90
!
!       Find the error bound and check for convergence
!
  100 BOUND = T * Y * (A + B + N - ONE) / ((A + N) - (A + B + N) * Y)
      IF (BOUND .LE. EPS) RETURN
!
!       Evaluate the next term of the expansion and then the
!       partial sum
!
      Q = Q * (A + B + N - ONE) * RHO2 / N
      V = V + Q
      T = T * Y * (A + B + N - ONE) / (A + N)
      TERM = V * T
      CDFR2 = CDFR2 + TERM
      N = N + ONE
      GO TO 100
      END
!
      PROGRAM MAIN
!
!       is a driver program that calls CDFR2 and produces output
!
      INTEGER SIZE
   10 WRITE (*,11)
   11 FORMAT (/1X,'ENTER Y, M (>1), N (>M), RHO2 (BETWEEN 0 AND 1)',  &
             /1X,'FOR CDFR2 ==>')
      READ (*,*) Y, M, SIZE, RHO2
      CDF = CDFR2(Y, M, SIZE, RHO2, IFAULT)
      IF (IFAULT .EQ. 0) THEN
        WRITE (*,21) Y, M, SIZE, RHO2, CDF
   21    FORMAT (/1X, 'CDFR2(', E10.4, ',', 2(I3, ','), E10.4, ') =', &
                E12.6)
      ELSE
```

```
      WRITE (*,31)
31    FORMAT (/1X,'THE INPUT VALUE IS ILLEGAL!')
  ENDIF
  WRITE(*,41)
41 FORMAT (/1X, 'ENTER 1 TO CONTINUE OR 0 TO QUIT ==> ')
  READ (*,*)K
  IF (K .EQ. 1) GOTO 10
  STOP
  END
```

Chapter 29
Student's t distribution)

Function tcdf(x, p, plim, ier)
File : Tcdf.f90

```
function tcdf( x, p, plim, ier )

!------------------------------------------------------------------------
!
!    Computes the probability that a random variable distributed
!    according to the Student's t distribution with P degrees of
!    freedom, is less than or equal to X.
!
!    X     - Input . Value of the variable                  - Real
!    P     - Input . Degrees of freedom              (P > 0) - Real
!    PLIM  - Input . Limit for P over which the distribution - Real
!                    is approximated by a normal distr.
!    IER   - Output. Return code :                          - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then tcdf = -1.)
!                    2 = maximum number of iterations reached
!                        (then tcdf = value at last iteration,
!                                     or zero)
!                    3 = Beta out of limits
!                        (betacdf < -EPS or betacdf > 1+EPS)
!
!    External functions called:
!      BETACDF  NCDF
!
!    NOTE : the error on tcdf is supposed to be minimal.
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: tcdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, p, plim
   integer, intent(out) :: ier

   ! local declarations
   ! ------------------

   real(kind=8), external :: betacdf, ncdf

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8

   real(kind=8) :: t, xx

   !----------------------------------------------------------------
```

```
    ! Test for valid input arguments

    if ( p <= zero ) then
      ier  = 1
      tcdf = -one
      return
    end if

    if ( p <= plim ) then
      xx = x * x
      t  = betacdf( xx/(xx+p), half, p*half, ier )
      if ( x >= zero ) then
        tcdf = half + t*half
      else
        tcdf = half - t*half
      end if
    else
      ier  = 0
      tcdf = ncdf( x )
    end if

  end function tcdf
```

Chapter 30
Noncentral t distribution)

Function tnccdf(x, q, a, delta, maxitr, ier)
File : tnccdf.f90

```
function tnccdf( x, q, a, delta, maxitr, ier )

!------------------------------------------------------------------
!
!    Computes the probability that a random variable distributed
!    according to the noncentral t distribution with Q degrees of
!    freedom, A centrality parameter, is less than or equal to X
!    Note: P( t'q (x) < a ) = P( L'q (a) > x )
!
!    X     - Input . Value of the variable                  - Real
!    Q     - Input . Degrees of freedom            (Q >  0) - Real
!    A     - Input . Eccentricity parameter                 - Real
!    DELTA - Input . Maximum absolute error required on     - Real
!                    tnccdf (stopping criterion)
!                    (eps < delta < 1 where eps is machine
!                    epsilon; see parameter statement below)
!    MAXITR- Input . Maximum number of iterations           - Integer
!    IER   - Output. Return code :                          - Integer
!                    0 = normal
!                    1 = invalid input argument
!                        (then tnccdf = one)
!                    2 = maximum number of iterations reached
!                        (then tnccdf = value at last iteration,
!                              or one)
!                    3 = required accuracy cannot be reached
!                        (then tnccdf = value at last iteration)
!                    4 = error in auxiliary function (chi2cdf or tcdf)
!                    7 = result out of limits (i.e. <0 or >1)
!
!    External functions called:
!      LPRIMECDF
!
!------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: tnccdf

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, q, a, delta
   integer, intent(in) :: maxitr
   integer, intent(out) :: ier

   ! local declarations
   ! ------------------

   real(kind=8), external :: lprimecdf
```

```
    real(kind=8), parameter :: one=1.0_8

    !-----------------------------------------------------------------

    tnccdf = one - lprimecdf( a, q, x, delta, maxitr, ier )

  end function tnccdf
```

Chapter 31
Doubly noncentral t distribution)

Function tdnccdf(x, df, delta, alamb, eps, iflag)
File : tdnccdf.f90

```
function tdnccdf( x, df, delta, alamb, eps, iflag )

!------------------------------------------------------------------
!
!    Computes the probability that a random variable distributed
!    according to the doubly noncentral t distribution with DF
!    degrees of freedom, DELTA and ALAMB noncentralitry
!    parameters, is less than or equal to X.
!
!    X     - Input . Value of the variable                  - Real
!    DF    - Input . Degrees of freedom in the      (DF > 0) - Real
!                denominator
!    DELTA - Input . Noncentrality parameter for            - Real
!                the numerator
!    ALAMB - Input . Noncentrality parameter for  (ALAMB>=0) - Real
!                the denominator
!    EPS   - Input . Desired absolute accuracy  (EPS>=1E-10) - Real
!    IFLAG - Output. Return code :                          - Integer
!                0 = normal
!                1 = invalid input argument
!                    (then tdnccdf = -1.)
!                2 = error in auxiliary function betacdf
!                    (then tdnccdf = value at last iteration,
!                                or zero)
!                3 = vector dimensions too small
!                    (increase nx)
!
!    This is a simple Fortran90 adaptation of subroutine CDFDNT by
!    C.P. Reeve:
!    Reeve, C.P. (1986). An algorithm for computing the doubly
!    noncentral t c.d.f. to a specified accuracy.
!    Statistical Engineering Division, note 86-5, december.
!
!------------------------------------------------------------------

  implicit none

  ! Function
  ! --------

  real(kind=8) :: tdnccdf

  ! Arguments
  ! ---------

  real(kind=8), intent(in) :: x, df, delta, alamb, eps
  integer, intent(out) :: iflag

  ! Local declarations
  ! ------------------

  real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
  integer, parameter :: nx=10000
```

```
real(kind=8) :: cdfx, darg, delsq, fa, fc, ga, gb, gc, sdelta, sum, xx, yy
real(kind=8) :: bfi(1:nx), bfj(1:nx), poi(1:nx), poj(1:nx)
integer :: i, imin, jmin, k, l, ni, nj
logical :: ll

!------------------------------------------------------------------

! Test for valid input arguments

if ( df <= zero .or. alamb < zero .or.      &
     eps < 1.e-10_8 .or. eps >= one ) then
   iflag = 1
   tdnccdf = -one
   return
end if

cdfx = zero
tdnccdf = zero
iflag = 0

delsq = delta**2
fa = half*delsq
ga = half*alamb
gb = half*df
yy = df/(df+x*x)
xx = one - yy

! If x<0 set ll=true, reverse sign of delta and use the
! identity described up front for computing the cdf

ll = x < zero
if ( xx >= one ) then
   tdnccdf = one
   if ( ll ) tdnccdf = zero
   return
end if
sdelta = delta
if ( ll ) sdelta = -sdelta

! Compute Poisson probabilities in vector poi

call poisst( fa, eps, imin, ni, poi(:nx), nx, iflag )
if ( iflag /= 0 ) return
if ( yy < one ) then
   fc = half + imin
   ! Compute Poisson probabilities in vector poj
   call poisst( ga, eps, jmin, nj, poj(:nx), nx, iflag )
   if ( iflag /= 0 ) return
   gc = gb + jmin

   ! Sum the terms corresponding to even values of index i

   call grid( ni, nj, fc, gc, bfi(1:ni), bfj(1:nj), poi(1:ni), poj(1:nj), xx, yy, cdfx, iflag )
```

```
    if ( iflag /= 0 ) then
      tdnccdf = cdfx
      return
    end if
  end if
  if ( delta == zero ) then
    ni = 0
    sum = zero
    if ( yy >= one ) then
      cdfx = half*(cdfx+one-sum)
      if ( ll ) cdfx = one - cdfx
      tdnccdf = cdfx
      return
    end if
  else

    ! Compute Poisson-like probabilities in vector poi

    k = int(fa)
    if ( imin > 0 ) then
      imin = imin-1
      ni = ni+1
    end if
    darg = (k+half)*log(fa)-fa-dlgama(k+one+half)
    l = k-imin+1
    poi(l) = sign(exp(darg),sdelta)
    sum = poi(l)
    do i = k-1, imin, -1
      l = l-1
      poi(l) = poi(l+1)*(i+one+half)/fa
      sum = sum+poi(l)
    end do
    l = k-imin+1
    do i = k+1, imin+ni-1
      l = l+1
      poi(l) = poi(l-1)*fa/(i+half)
      sum = sum+poi(l)
    end do
    if ( yy >= one ) then
      cdfx = half*(cdfx+one-sum)
      if ( ll ) cdfx = one - cdfx
      tdnccdf = cdfx
      return
    end if
    fc = one+imin

    ! Sum the terms corresponding to odd values of index i

    call grid( ni, nj, fc, gc, bfi(1:ni), bfj(1:nj), poi(1:ni), poj(1:nj), xx, yy, cdfx, iflag )
    if ( iflag /= 0 ) then
      tdnccdf = cdfx
      return
```

```
      end if
    end if

    cdfx = half*(cdfx+one-sum)
    if ( ll ) cdfx = one - cdfx
    tdnccdf = cdfx

  end function tdnccdf
  subroutine poisst( alamb, eps, l, nspan, v, nv, iflag )
    ! Compute the Poisson(alamb) probabilities over the range (l,k)
    ! where the total tail probability is less than eps/2, sum the
    ! probabilities and shift them to the beginning of vector v.

    implicit none

    !  Arguments
    real(kind=8) :: alamb, eps
    integer :: l, nspan, nv, iflag
    real(kind=8) :: v(1:nv)

    !  Local declarations
    real(kind=8), parameter :: zero=0.0_8, one=1.0_8, two=2.0_8, three=3.0_8
    real(kind=8) :: dal, dk, dlimit, dsum, pk, pl
    integer :: i, inc, k, nk, nl

    dlimit = one-two*eps/three
    k = int(alamb)
    l = k+1
    if ( alamb == zero ) then
      pl = one
    else
      dal = alamb
      dk = k
      pl = exp(dk*log(dal)-dal-dlgama(k+one))
    end if
    pk = alamb*pl/l
    nk = nv/2
    nl = nk+1
    dsum = zero
    do
      if ( pl < pk ) then
        nk = nk+1
        if ( nk > nv ) then
          iflag = 3
          return
        end if
        v(nk) = pk
        dsum =dsum + pk
        k = k+1
        if ( dsum >= dlimit ) exit
        pk = alamb*pk/(k+1)
      else
```

```
            nl = nl-1
            v(nl) = pl
            dsum = dsum + pl
            l = l-1
            if ( dsum >= dlimit ) exit
            pl = l*pl/alamb
          endif
        end do
        inc = nl-1
        do i = nl, nk
          v(i-inc) = v(i)
        end do
        nspan = nk-inc

      end subroutine poisst

      subroutine edget( nk, fc, gc, xx, yy, bfk, cdfx, poi, poj, iflag, l )
        ! Compute the Beta cdf's by a recurrence relation along the edges
        ! i=imin and j=jmin of a grid. The corresponding components of
        ! the F'' cdf are included in the summation. Terms which might
        ! cause underflow are set to zero.

        implicit none

        ! Arguments
        real(kind=8) :: fc, gc, xx, yy, cdfx
        integer :: nk, iflag, l
        real(kind=8) :: bfk(1:nk), poi(1:nk), poj(1:1)

        ! Local declarations
        real(kind=8), external :: betacdf
        real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
        real(kind=8), parameter :: deuflo=-706.893_8
        real(kind=8) :: darg, di, dk, fd, fk
        integer :: i, j, k, kflag

        fd = fc-one
        k = max(l,min(nk,int((gc-one)*xx/yy-fd)))
        fk = fd+k
        bfk(k) = betacdf( xx,fk,gc,iflag )
        if ( iflag /= 0 ) then
          iflag = 2
          return
        end if
        if ( l == 1 ) bfk(k) = one-bfk(k)
        if ( nk /= 1 ) then
          darg = fk*log(xx)+gc*log(yy)-log(fk)+dlgama(fk+gc)-dlgama(fk)-dlgama(gc)
          if ( darg < deuflo ) then
            dk = zero
          else
            dk = exp(darg)*(-one)**l
          end if
```

```
      if ( k < nk ) then
        bfk(k+1) = bfk(k)-dk
        di = dk
        kflag = 1
        do i = k+1, nk-1
          if ( kflag == 1 ) then
            di = di*(fd+gc+(i-1))*xx/(fd+i)
            if ( dk+di == dk ) then
              kflag = 0
              di = zero
            end if
          end if
          bfk(i+1) = bfk(i)-di
        end do
      end if
      di = dk
      kflag = 1
      do i = k-1, l, -1
        if ( kflag == 1 ) then
          di = di*(fc+i)/((fd+gc+i)*xx)
          if ( dk+di == dk ) then
            kflag = 0
            di = zero
          end if
        end if
        bfk(i) = bfk(i+1)+di
      end do
    end if
    do i = l, nk
      cdfx = cdfx + poi(i)*poj(1)*bfk(i)
    end do

  end subroutine edget

  subroutine grid( ni, nj, fc, gc, bfi, bfj, poi, poj, xx, yy, cdfx, iflag )
    ! Compute double summation of components of the t'' c.d.f. over the
    ! grid i=imin to imax and j=jmin to jmax

    implicit none

    !  Arguments
    real(kind=8) :: fc, gc, xx, yy, cdfx
    integer :: ni, nj, iflag
    real(kind=8) :: bfi(1:ni), bfj(1:nj), poi(1:ni), poj(1:nj)

    !  Local declarations
    integer :: i, j

    !  Compute Beta cdf by recurrence when i=imin and j=jmin to jmax

    call edget( nj, gc, fc, yy, xx, bfj(1:nj), cdfx, poj(1:nj), poi(1:1), iflag, 1 )
    if ( ni <= 1 .or. iflag /= 0 ) return
```

```
  ! Compute Beta cdf by recurrence when j=jmin, i=imin to imax

  bfi(1) = bfj(1)
  call edget( ni, fc, gc, xx, yy, bfi(1:ni), cdfx, poi(1:ni), poj(1:1), iflag, 2 )
  if ( nj <= 1 .or. iflag /= 0 ) return

  ! Compute Beta cdf by recurrence when i>imin and j>jmin

  do i = 2, ni
    bfj(1) = bfi(i)
    do j = 2, nj
      bfj(j) = xx*bfj(j) + yy*bfj(j-1)
      cdfx = cdfx + poi(i)*poj(j)*bfj(j)
    end do
  end do

end subroutine grid
```

Part II
INVERSE DISTRIBUTION FUNCTIONS

Chapter 32
Beta distribution

Function ibeta(alpha, p, q, ier)
File : iBeta.f90

Chapter 33
Continued binomial distribution

Function ibinomc(pbin, n, p, ier)
File : iBinomc.f90

Chapter 34
Fiducial continued binomial distribution

Function ibinomcf(f, a, b, u, ier)
File : iBinomcf.f90

Chapter 35
Ksi-square distribution

Function ksi2i(f, p, q, a2, delta, maxit, iap, ier)
File : Ksi2i.f90

Chapter 36
Normal distribution

Function norminv(p)
File : Norminv.f90

Chapter 37
Student's t distribution

Function tinv(prob, p, plim, ier)
File : tinv.f90

```
function tinv( prob, p, plim, ier )

!------------------------------------------------------------------------
!
!     Computes the value of a random variable distributed
!     according to the Student's t distribution with P degrees of
!     freedom, given the cdf at this value.
!
!     PROB  - Input . The probability (cdf)                    - Real
!     P     - Input . Degrees of freedom              (P > 0) - Real
!     PLIM  - Input . Limit for P over which the distribution - Real
!                     is approximated by a normal distr.
!     IER   - Output. Return code :                            - Integer
!                     0 = normal
!                     1 = invalid input argument
!                         (then tinv = 0.)
!                     2 = error in auxiliary function
!
!     External functions called:
!       IBETA  NORMINV
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: tinv

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: prob, p, plim
   integer, intent(out) :: ier

   ! local declarations
   ! ------------------

   real(kind=8), external :: ibeta, norminv

   real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8

   real(kind=8) :: b, pb

   !---------------------------------------------------------------

   ! Test for valid input arguments

   tinv = zero
   if ( p <= zero ) then
      ier  = 1
      return
```

```
    end if

    ier = 0
    if ( p <= plim ) then
       pb = abs(prob+prob-one)
       b  = ibeta( pb, half, p*half, ier )
       if ( ier /= 0 ) then
          ier = 2
          return
       end if
       tinv = sqrt(b*p/(one-b))
       if ( prob < half ) tinv = -tinv
    else
       tinv = norminv( prob )
    end if

  end function tinv
```

Part III
PROBABILITY MASS FUNCTIONS

Chapter 38
Ratio of two normal distributions

Subroutine normratio(w, xm, ym, xs, ys, r, f, ier)
File : normratio.f90

```
subroutine normratio( w, xm, ym, xs, ys, r, f, ier )

  ! dll pour densite du rapport de 2 normales
  ! BL 2010

  ! entree:
  !    w : rapport x/y (reel*8)
  !    xm: moyenne     (reel*8)
  !    ym: moyenne     (reel*8)
  !    xs: ec.type     (reel*8)
  !    ys: ec.type     (reel*8)
  !    r : correlation (reel*8)
  ! sortie:
  !    f   : resultat  (reel*8)
  !    ier :  0 = OK   (entier)
  !           /0 = erreur dans fonction

  implicit none

  DLL_EXPORT normratio

  real(kind=8), intent(in) :: w, xm, ym, xs, ys, r
  real(kind=8), intent(out) :: f
  integer, intent(out) :: ier

!! real(kind=8), external :: dlgama

  real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8, two=2.0_8
  real(kind=8), parameter :: pi=3.14159265358979323846264 34_8
  real(kind=8), parameter :: explower=-706.893_8
  !     explower = minimum valid argument for the exponential function
  integer, parameter :: maxit=5000   ! maximum number of iterations

  real(kind=8) :: a, hb, hf, fp, r2r, u, v, ww, xms, yms
  integer :: j, j0

  !------------------------------------------------------------------

  f = zero
  ier = 0
  !  Test for valid input arguments
  if ( r <= -one .or. r >= one .or. xs <= zero .or. ys <= zero ) then
    ier = 1
    return
  end if

  ww = w*ys/xs
  xms = xm/xs
  yms = ym/ys
  r2r = one - r*r
  u = yms + xms*ww - r*(xms+yms*ww)
  v = one + ww*ww - two*r*ww
  a = half*u*u/v/r2r
```

```
    j0 = max(0, int(a-half))
    if ( j0 < 50 ) j0 = 0
    hf = j0*log(a) - dlgama(j0+half) - half*(xms*xms+yms*yms-two*r*xms*yms)/r2r &
       + half*log(r2r/pi) - log(v)
    if ( hf >= explower ) then
      if ( hf <= -explower ) then
        hf = exp(hf)
      else
        ier = 2
        return
      end if
    else
      return
    end if

    hb = hf
    f = hf
    fp = f
    do j = 1, j0
      hf = hf/(j0+j-half)*a
      hb = hb/a*(j0-j+half)
      f = f + hf + hb
      if ( f == fp ) then
        f = f*ys/xs
        return
      end if
      fp = f
    end do
    do j = j0+1, maxit
      hf = hf/(j-half)*a
      f = f + hf
      if ( f == fp ) then
        f = f*ys/xs
        return
      end if
      fp = f
    end do
    if ( j >= maxit ) ier = 3
    f = f*ys/xs

  end subroutine normratio
```

Chapter 39
ksi-square distribution

Function ksi2d(x, p, q, a2, err, maxitr, ier)
File : ksi2d.f90

```
function ksi2d( x, p, q, a2, err, maxitr, ier )

!------------------------------------------------------------------------
!
!    Computes the probability density at point X of a random variable
!    distributed according to the ksi-square distribution with P and Q
!    degrees of freedom and A2 eccentricity parameter.
!
!    X     - Input . Value of the variable          (X >= 0) - Real
!    P     - Input . First degrees of freedom        (P >  0) - Real
!    Q     - Input . Second   "    "     "          (Q >  0) - Real
!                  N.B. Although the function is defined
!                  for any positive Q, it is wise to
!                  take Q >= P
!    A2    - Input . Eccentricity parameter         (A2>= 0) - Real
!    ERR   - Input . Maximum relative error required         - Real
!                  (stopping criterion)  (1e-12 < err < 1)
!    MAXITR- Input . Maximum number of iterations            - Integer
!    IER   - Output. Return code :                           - Integer
!                  0 = normal
!                  1 = invalid input argument
!                     (then KSI2D = -1.0)
!                  2 = maximum number of iterations reached
!                     (then KSI2D = value at last iteration)
!
!    External functions called:
!      DLGAMA (Log(Gamma(.)) if not in FORTRAN library
!
!    Fortran functions called:
!      ABS  EXP  LOG  SQRT  (and DLGAMA if available)
!
!    Algorithm by J-L. Guigues (1981)
!
!    J. Poitevineau (CNRS-URA1201)
!    December 03, 1986   (first version 12/14/84)
!
!------------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: ksi2d

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: x, p, q, a2, err
   integer, intent(in) :: maxitr
   integer, intent(out) :: ier

   ! local declarations
```

```
  ! ------------------

!! real(kind=8), external :: dlgama

  real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8
  real(kind=8), parameter :: two=one+one
  real(kind=8), parameter :: dlgam5=0.5723649429247000_8    ! =log(gamma(0.5))
  real(kind=8), parameter :: twol=0.6931471805599453_8      ! =log(2)
  real(kind=8), parameter :: eps=0.223e-15_8, explower=-706.893_8
  real(kind=8), parameter :: xinfinity=1.0e300_8
  real(kind=8), parameter :: relerr=1.0e-14_8 ! Relative error assumed
                                              ! in recurrence calculations

  !     These constants are machine dependent:
  !     eps    = machine epsilon
  !              (the smallest real such that 1.0 + eps > 1.0)
  !     explower = minimum valid argument for the exponential function
  !              (i.e. log(tinyr))
  !     xinfinity = stands for +infinity

  real(kind=8) :: axql, dj, pa, pb, phitl, pq2, p2, q2, uk, ukl, uphi, usum, xkl
  integer :: j

  !---------------------------------------------------------------------

  ! Test for valid input arguments

  if ( x < zero .or. a2 < zero .or. p <= zero .or. q <= zero  &
     .or. err >= one .or. err <= 1.0e-12_8 ) then
     ier   = 1
     ksi2d = -one
     return
  end if

  ier = 0

  ! Define constants

  p2  = p * half
  q2  = q * half
  pq2 = p2 + q2

  ! Case p = 1

  if ( abs(p-one) <= eps ) then
     if ( x > eps ) then
        ksi2d = zero
        pa = -twol - half*log(q*x) +                        &
             dlgama(half+q2) - dlgam5 - dlgama(q2) +        &
             (q2+half)*log(q/((q+a2+x)-(two*sqrt(a2*x))))
        pb = -twol - half*log(q*x) +                        &
             dlgama(half+q2) - dlgam5 - dlgama(q2) +        &
             (q2+half)*log(q/((q+a2+x)+(two*sqrt(a2*x))))
```

```
      if ( pa >= explower ) ksi2d = exp(pa)
      if ( pb >= explower ) ksi2d = ksi2d + exp(pb)
    else
      ksi2d = xinfinity
    end if
    return
  end if

  ksi2d = zero

  if ( abs(p2-one) > eps ) then

    !  Case p /= 2

    if ( x <= eps ) return
    xkl = dlgama(pq2)-dlgama(p2)-dlgama(q2)+q2*log(q)+(p2-one)*log(x)-pq2*log(a2+q+x)
    if ( xkl >= explower ) ksi2d = exp(xkl)
    !  Case a2 = 0
    if ( a2 <= eps ) return
  else

    !  Case p = 2

    xkl = dlgama(pq2) - dlgama(q2) + q2*log(q) - pq2*log(a2+q+x)
    if ( xkl >= explower ) ksi2d = exp(xkl)
    !  Case a2 = 0 or x = 0
    if ( a2 <= eps .or. x <= eps ) return
  end if

  !  General case (iterations)

  axql = log( a2*x /((a2+q+x)*(a2+q+x)) )
  ukl  = xkl
  uk   = ksi2d
  usum = zero

  !  Iteration loop

  do j = 0, maxitr
    dj    = j
    usum  = usum + dj*uk
    phitl = log((two*dj+pq2+one)*(two*dj+pq2)/((dj+one)*(dj+p2))) + axql
    ukl   = phitl + ukl
    uk    = zero
    if ( ukl >= explower ) uk = exp(ukl)
    !  Check accuracy (stopping rule)
    if ( phitl < zero ) then
      uphi = one
      if ( phitl >= explower ) uphi = uphi - exp(phitl)
      !  the end ?
      if ( (two*relerr*usum+uk/uphi)/ksi2d < err ) return
    end if
    ksi2d = ksi2d + uk
```

```
    end do

    ! Maximum number of iterations is reached

    ier = 2

  end function ksi2d
```

Chapter 40
Square of the multiple correlation coefficient distribution)

Function SQMCOR(X, IP, N, RHO2, IFAULT)
File : sqmcor.f90

```
      REAL FUNCTION PDFR2(Y, M, SIZE, RHO2, IFAULT)
!
! Computes the density (pdf) of the square of the sample multiple
! correlation coefficient for given abscissa Y, number of random
! variables M, sample size SIZE, and square of the population
! multiple correlation coefficient RHO2
!
! Reference:
! Ding, C.G. (1996) On the computation of the distribution of
! the square of the sample multiple correlation coefficient,
! Computational Statistics & Data Analysis, 22 , 345-350.
!
! IFAULT is a fault indicator:
! = 1 if any of the input values is illegal
! = 0 otherwise
!
! No auxiliary algorithm is required
!
      INTEGER SIZE
      REAL N
      DATA EPS / 1.0E-6 /
      DATA HALF, ZERO, ONE, TWO / 0.5, 0.0, 1.0, 2.0 /
      DATA RP / 1.772453850905516028 /

      PDFR2 = ZERO
      IFAULT = 1
      IF (M .LE. 1 .OR. SIZE .LE. M .OR. RHO2 .LT. ZERO .OR.  &
         RHO2 .GT. ONE) RETURN
      IFAULT = 0
      IF (Y .LE. ZERO .OR. Y .GE. ONE) RETURN
      A = (M - 1) / TWO
      B = (SIZE - M) / TWO
      AB = (SIZE - 1) / TWO
      IF (MOD(M - 1, 2) .EQ. 0) THEN
        NA = A - HALF
        GA = ONE
        IF (NA .EQ. 0) GO TO 25
        DO 10 I = 1, NA
   10    GA = GA * I
      ELSE
        NA = A
        GA = RP
        IF (NA .EQ. 0) GO TO 25
        DO 20 I = 1, NA
   20    GA = GA * (I - HALF)
      ENDIF
   25 IF (MOD(SIZE - M, 2) .EQ. 0) THEN
        NB = B - HALF
        GB = ONE
        IF (NB .EQ. 0) GO TO 50
        DO 30 I = 1, NB
   30    GB = GB * I
```

```
      ELSE
        NB = B
        GB = RP
        IF (NB .EQ. 0) GO TO 50
        DO 40 I = 1, NB
   40     GB = GB * (I - HALF)
      ENDIF
   50 IF (MOD(SIZE - 1, 2) .EQ. 0) THEN
        NAB = AB - HALF
        GAB = ONE
        IF (NAB .EQ. 0) GO TO 80
        DO 60 I = 1, NAB
   60     GAB = GAB * I
      ELSE
        NAB = AB
        GAB = RP
        DO 70 I = 1, NAB
   70     GAB = GAB * (I - HALF)
      ENDIF

!
!       Evaluate the first term
!
   80 N = ONE
      Q = (ONE - RHO2) ** AB
      V = Q
      S = Y ** (A - ONE) * (ONE - Y) ** (B - ONE) * GAB / GA / GB
      TERM = Q * S
      PDFR2 = TERM
!
!       Check if a + n > (a + b + n)y
!
   90 IF (A + N .GT. (A + B + N) * Y) GO TO 100
!
!       Evaluate the next term of the expansion and then the
!       partial sum
!
      Q = Q * (A + B + N - ONE) * RHO2 / N
      V = V + Q
      S = S * Y * (A + B + N - ONE) / (A + N - ONE)
      TERM = Q * S
      PDFR2 = PDFR2 + TERM
      N = N + ONE
      GO TO 90
!
!       Find the error bound and check for convergence
!
  100 BOUND = S * Y * (A + B + N - ONE) * (ONE - V) / (A + N - ONE)
      IF (BOUND .LE. EPS) RETURN
!
!       Evaluate the next term of the expansion and then the
!       partial sum
```

```
!
      Q = Q * (A + B + N - ONE) * RHO2 / N
      V = V + Q
      S = S * Y * (A + B + N - ONE) / (A + N - ONE)
      TERM = Q * S
      PDFR2 = PDFR2 + TERM
      N = N + ONE
      GO TO 100
      END
!
      PROGRAM MAIN
!
!       is a driver program that calls PDFR2 and produces output
!
      INTEGER SIZE

   10 WRITE (*,11)
   11 FORMAT (/1X,'ENTER Y, M (>1), N (>M), RHO2 (BETWEEN 0 AND 1)', &
             /1X,'FOR PDFR2 ==> ')
      READ (*,*) Y, M, SIZE, RHO2
      PDF = PDFR2(Y, M, SIZE, RHO2, IFAULT)
      IF (IFAULT .EQ. 0) THEN
        WRITE (*,21) Y, M, SIZE, RHO2, PDF
   21   FORMAT (/1X, 'PDFR2(', E10.4, ',', 2(I3, ','), E10.4, ') =', &
               E12.6)
      ELSE
        WRITE (*,31)
   31   FORMAT (/1X, 'THE INPUT VALUE IS ILLEGAL!')
      ENDIF
      WRITE(*,41)
   41 FORMAT (/1X, 'ENTER 1 TO CONTINUE OR 0 TO QUIT ==> ')
      READ (*,*)K
      IF (K .EQ. 1) GOTO 10
      STOP
      END
```

Part IV
RANDOM GENERATORS

Chapter 41
Beta distribution

Function genbet(aa, bb, idum, ier)
File : genbet.f90

```
function genbet( aa, bb, idum, ier )

!----------------------------------------------------------------------
!
!     Returns a random variable distributed according to a Beta
!     distribution with parameters aa and bb.
!
!     aa    - Input . 1st parameter of the Beta distribution   - Real
!                   (1e-37 < aa )
!     bb    - Input . 2nd parameter of the Beta distribution  - Real
!                   (1e-37 < bb )
!     idum  - Input . A negative integer to initialize the    - Integer
!           Output. random sequence. Thereafter do not
!                   alter this argument between successive
!                   deviates in a sequence
!     ier   - Output. Return code :                           - Integer
!                   0 = normal
!                   1 = invalid input argument aa or bb
!                      (then genbet = -1.0)
!
!     External functions called:
!       RAN1
!     Fortran  functions called:
!       EXP  LOG  MAX  MIN  SQRT
!
!----------------------------------------------------------------------
!     From
!     RANLIB
!          Library of Fortran Routines for Random Number Generation
!                        Compiled and Written by:
!                             Barry W. Brown
!                              James Lovato
!                  Department of Biomathematics, Box 237
!                  The University of Texas, M.D. Anderson Cancer Center
!                  1515 Holcombe Boulevard
!                  Houston, TX      77030
!
!**********************************************************************
!
!     REAL FUNCTION GENBET( A, B )
!              GeNerate BETa random deviate
!
!
!                           Function
!
!
!     Returns a single random deviate from the beta distribution with
!     parameters A and B.  The density of the beta is
!              x^(a-1) * (1-x)^(b-1) / B(a,b) for 0 < x < 1
!
!
!                           Arguments
```

```
!
!
!     A --> First parameter of the beta distribution
!                   REAL A
!
!     B --> Second parameter of the beta distribution
!                   REAL B
!
!
!                        Method
!
!
!     R. C. H. Cheng
!     Generating Beta Variate with Nonintegral Shape Parameters
!     Communications of the ACM, 21:317-322  (1978)
!     (Algorithms BB and BC)
!
!*****************************************************************

    implicit none

!     Function
!     --------
    real(kind=4) :: genbet

!     Arguments
!     ---------
    real(kind=4), intent(in) :: aa, bb
    integer, intent(inout) :: idum
    integer, intent(out) :: ier

!     Local declarations
!     ------------------
    real(kind=4), external :: ran1

    real(kind=4), parameter :: expmax=89.0
    !  Close to the largest number that can be exponentiated
    real(kind=4), parameter :: infnty=1.0e38
    !  Close to the largest representable single precision number

    real(kind=4) :: a, b, delta, r, s, t, u1, u2, v, w, y, z
    real(kind=4), save :: alpha, beta, gamma, k1, k2
    real(kind=4), save :: olda=-1.0, oldb=-1.0
    logical :: qsame

    !-----------------------------------------------------------------

    !  Test for valid input arguments
    if ( aa <= 0.0 .or. bb <= 0.0 ) then
      ier = 1
      genbet = -1.0
      return
    end if
```

```
ier = 0
qsame = (olda == aa) .and. (oldb == bb)
olda = aa
oldb = bb

if ( min(aa,bb) > 1.0 ) then

  ! Algorithm BB

  ! Initialize
  if ( .not. qsame ) then
    a = min(aa,bb)
    b = max(aa,bb)
    alpha = a + b
    beta = sqrt((alpha-2.0)/ (2.0*a*b-alpha))
    gamma = a + 1.0/beta
  end if

  do
    u1 = ran1( idum )

    ! Step 1
    u2 = ran1( idum )
    v = beta*log(u1/ (1.0-u1))
    if ( v > expmax ) then
      w = infnty
    else
      w = a*exp(v)
    end if
    z = u1**2*u2
    r = gamma*v - 1.3862944
    s = a + r - w

    ! Step 2
    if ( s+2.609438 >= 5.0*z ) exit

    ! Step 3
    t = log(z)
    if ( s > t ) exit

    ! Step 4
    if ( (r+alpha*log(alpha/(b+w))) >= t ) exit
  end do

  ! Step 5
  if ( aa == a ) then
    genbet = w/(b+w)
  else
    genbet = b/(b+w)
  end if

else
```

```
! Algorithm BC

! Initialize
if ( .not. qsame ) then
   a = max(aa,bb)
   b = min(aa,bb)
   alpha = a + b
   beta = 1.0/b
   delta = 1.0 + a - b
   k1 = delta*(0.0138889+0.0416667*b)/(a*beta-0.777778)
   k2 = 0.25 + (0.5+0.25/delta)*b
end if

do
   u1 = ran1( idum )

   ! Step 1
   u2 = ran1( idum )
   if ( u1 < 0.5 ) then
      ! Step 2
      y = u1*u2
      z = u1*y
      if ( 0.25*u2+z-y >= k1 ) cycle
   else
      ! Step 3
      z = u1**2*u2
      if ( z <= 0.25 ) then
         v = beta*log(u1/ (1.0-u1))
         if ( v > expmax ) then
            w = infnty
         else
            w = a*exp(v)
         end if
         exit
      end if
      if ( z >= k2 ) cycle

   end if
   ! Step 4
   ! Step 5
   v = beta*log(u1/ (1.0-u1))
   if ( v > expmax ) then
      w = infnty
   else
      w = a*exp(v)
   end if
   if ( alpha*(log(alpha/(b+w))+v)-1.3862944 >= log(z) ) exit
end do

! Step 6
if ( a == aa ) then
   genbet = w/(b+w)
```

```
      else
        genbet = b/(b+w)
      end if

    end if

  end function genbet
```

Chapter 42
Binormal distribution

Subroutine binormdev(idum, rho, x1, x2)
File : binormdev.f90

```
subroutine binormdev( idum, rho, x1, x2 )

    Returns a random vector distributed according to a binormal
    distribution with zero means, unit variances and rho correlation.

!-----------------------------------------------------------------------
!
!    Returns a random vector distributed according to a binormal
!    distribution with zero means, unit variances and rho correlation.
!
!    idum  - Input . A negative integer to initialize the    - Integer
!            Output. random sequence. Thereafter do not
!                    alter this argument between successive
!                    deviates in a sequence
!    rho   - Input . Correlation             (-1<= rho <=1) - Real
!    x1    - Output. First coordinate                       - Real
!    x2    - Output. Second coordinate                      - Real
!
!    External functions called:
!      GAUSSBM
!    Fortran  functions called:
!      SQRT
!
!-----------------------------------------------------------------------

  implicit none

  ! Arguments
  ! ---------

  real(kind=8), intent(in) :: rho
  integer, intent(inout) :: idum
  real(kind=8), intent(out) :: x1, x2

  ! Local declarations
  ! ------------------

  real(kind=8), external :: gaussbm

  real(kind=8), parameter :: one=1.0_8

  !-----------------------------------------------------------------------

  x1 = gaussbm(idum)
  x2 = rho*x1 + sqrt(one-rho*rho)*gaussbm(idum)

end subroutine binormdev
```

Chapter 43
Exponential distribution

Function expdv1(idum)
File : expdv1.f90

```
function expdv1( idum )

!-----------------------------------------------------------------------
!
!     Returns a random variable distributed according to an exponential
!     distribution with parameter one.
!
!     idum  - Input . A negative integer to initialize the     - Integer
!             Output. random sequence. Thereafter do not
!                     alter this argument between successive
!                     deviates in a sequence
!
!     External functions called:
!       RAN1
!     Fortran  functions called:
!       LOG
!
!-----------------------------------------------------------------------

   implicit none

!     Function
!     --------

   real :: expdv1

!     Arguments
!     ---------

   integer, intent(inout) :: idum

!     Local declarations
!     ------------------

   real, external :: ran1
   real, parameter :: zero=0.0
   real :: dum

   !--------------------------------------------------------------

   do
     dum = ran1(idum)
     if ( dum > zero ) exit
   end do
   expdv1 = -log(dum)

end function expdv1
```

Chapter 44
Exponential distribution (Real*8 version)

Function expdv2(idum)
File : expdv2.f90

Chapter 45
Gamma distribution

Function gamdvr(ia, idum, ier)
File : GAMDVR.f90

```
function gamdvr( alpha, idum, ier )

!-----------------------------------------------------------------------
!
!     Returns a random variable distributed according to a Gamma
!     distribution with parameter aplha.
!
!     alpha - Input . Parameter of the Gamma distribution      - Real
!                     (epsilon < alpha < 1/epsilon )
!     idum  - Input . A negative integer to initialize the     - Integer
!             Output. random sequence. Thereafter do not
!                     alter this argument between successive
!                     deviates in a sequence
!     ier   - Output. Return code :                            - Integer
!                     0 = normal
!                     1 = invalid input argument alpha
!                        (then gamdvr = -1.0)
!                     2 = underflow
!
!     External functions called:
!       GAMDVI  RAN12
!     Fortran  functions called:
!       EPSILON  HUGE  LOG  MOD  NINT
!
!-----------------------------------------------------------------------
!
!     REMARK :
!
!       For integral values of alpha, the function gamdev is used
!       (Press et al. (1992), Numerical Recipes in Fortran, Cambridge
!       University Press).
!       For non-integral values of alpha, the ratio method is used
!       when alpha > 1, and the acceptance-rejection method is used
!       when alpha < 1 (see K. Lange (1999), Numerical Analysis for
!       Statisticians, Springer).
!
!-----------------------------------------------------------------------

   implicit none

!     Function
!     --------
   real(kind=8) :: gamdvr

!     Arguments
!     ---------
   real(kind=8), intent(in) :: alpha
   integer, intent(inout) :: idum
   integer, intent(out) :: ier

!     Local declarations
!     ------------------
```

```
real(kind=8), external :: gamdvi, ran12

real(kind=8), parameter :: zero=0.0_8, half=0.5_8, one=1.0_8, two=2.0_8
real(kind=8), parameter :: e=2.7182818284590452_8
real(kind=8) :: am1, ap1, r, u, uk, v, vk, w, x

!-----------------------------------------------------------------

ier = 0
! Test for valid input arguments
if ( alpha < epsilon(alpha) .or. one < alpha*epsilon(alpha) ) then
  ier = 1
  gamdvr = -one
  return
end if

! If alpha is an integral value, use Gamdev algorithm
if ( mod(alpha,one) <= epsilon(alpha) .and. alpha <= huge(1) ) then
  gamdvr = gamdvi( nint(alpha), idum, ier )
  return
end if

if ( alpha > one ) then
  ! Use ratio method (see K. Lange (1999), p277)

  ap1 = alpha + one
  am1 = one/(alpha - one)
  r = ((ap1*am1)**(half*alpha))*sqrt(ap1/am1)/e
  do
    u = ran12( idum )
    v = ran12( idum )
    x = r*v/u
    w = x * am1
    if ( two*am1*log(u)-one-log(w)+w <= zero ) exit
  end do

else
  ! use acceptance-rejection method (see k. lange (1999), p283)

  am1 = alpha - one
  vk = one/alpha
  uk = e/(e+alpha)
  do
    u = ran12( idum )
    if ( u < uk ) then
      x = ran12(idum)**vk
      if ( x <= zero ) then
        ier = 2
        gamdvr = zero
        return
      end if
      w = exp(-x)
    else
```

```
            x = one - log(ran12(idum))
            w = x**am1
          end if
          v = ran12( idum )
          if ( v <= w ) exit
        end do

      end if

      gamdvr = x

    end function gamdvr
```

Chapter 46
Gamma distribution (Real*8 version)

Function gamdvi(ia, idum, ier)
File : GAMDVI.f90

```
function gamdvi( ia, idum, ier )

!-----------------------------------------------------------------------
!
!    Returns a random variable distributed according to a Gamma
!    distribution with integer parameter ia.
!    This is the real*8 version of the Gamdev function
!    from Numerical Recipes in Fortran by Press et al. (1992).
!
!    ia    - Input . Parameter of the Gamma distribution     - Integer
!                  (ia >= 1)
!    idum  - Input . A negative integer to initialize the    - Integer
!            Output. random sequence. Thereafter do not
!                  alter this argument between successive
!                  deviates in a sequence
!    ier   - Output. Return code :                           - Integer
!                  0 = normal
!                  1 = invalid input argument alpha
!                     (then gamdvi = -1.0)
!
!    External functions called:
!      RAN12
!    Fortran  functions called:
!      EXP  LOG  SQRT
!
!-----------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: gamdvi

   ! Arguments
   ! ---------

   integer, intent(in) :: ia
   integer, intent(inout) :: idum
   integer, intent(out) :: ier

   ! Local declarations
   ! ------------------

   real(kind=8), external :: ran12

   real(kind=8), parameter :: zero=0.0_8, one=1.0_8, two=2.0_8
   real(kind=8) :: am, e, s, v1, v2, x, y
   integer :: j

   !-----------------------------------------------------------------------

   ier = 0
```

```
    if ( ia < 1 ) then
      ier = 1
      gamdvi = -one
      return
    end if
    if ( ia < 6 ) then
      x = one
      do j = 1, ia
        x = x*ran12(idum)
      end do
      x = -log(x)
    else
      do
        v1 = two*ran12(idum)-one
        v2 = two*ran12(idum)-one
        if ( v1**2+v2**2 > one ) cycle
        y = v2/v1
        am = ia-1
        s = sqrt(two*am+one)
        x = s*y+am
        if ( x <= zero ) cycle
        e = (one+y**2)*exp(am*log(x/am)-s*y)
        if ( ran12(idum) <= e ) exit
      end do
    end if
    gamdvi = x

  end function gamdvi
```

Chapter 47
Normal distribution

Function gaussbm(idum)
File : gaussBM.f90

```
function gaussbm( idum )

!-----------------------------------------------------------------------
!
!    Returns a random variable distributed according to a Normal
!    distribution using Box & Muller (1958) method.
!
!    idum  - Input . A negative integer to initialize the    - Integer
!            Output. random sequence. Thereafter do not
!                    alter this argument between successive
!                    deviates in a sequence
!
!    External functions called:
!      RAN12
!    Fortran  functions called:
!      COS  LOG  SIN  SQRT
!
!-----------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: gaussbm

   ! Arguments
   ! ---------

   integer, intent(inout) :: idum

   ! Local declarations
   ! ------------------

   real(kind=8), external :: ran12

   real(kind=8), parameter :: twopi=6.2831853071795865_8
   real(kind=8), save :: gset
   real(kind=8) :: fac, v
   logical, save :: iset=.true.

   !-----------------------------------------------------------------------

   if ( iset ) then
      fac = sqrt(-2.0_8*log(ran12(idum)))
      v = twopi*ran12(idum)
      gset = fac*sin(v)
      gaussbm = fac*cos(v)
      iset = .false.
   else
      gaussbm = gset
      iset = .true.
   end if

end function gaussbm
```

Chapter 48
Normal distribution (other)

Function gasdev(idum)
File : GASDEV.f90

```
      FUNCTION gasdev(idum)
            INTEGER idum
            REAL gasdev
      !U    USES ran1
            LOGICAL iset
            REAL fac,gset,rsq,v1,v2,ran1
            SAVE iset,gset
            DATA iset/.true./
            if (iset) then
      1       v1=2.*ran1(idum)-1.
              v2=2.*ran1(idum)-1.
              rsq=v1**2+v2**2
              if(rsq.ge.1..or.rsq.eq.0.)goto 1
              fac=sqrt(-2.*log(rsq)/rsq)
              gset=v1*fac
              gasdev=v2*fac
              iset=.false.
            else
              gasdev=gset
              iset=.true.
            end if
      END
```

Chapter 49
Normal distribution (other modified version)

Function gasdv2(idum)
File : GASDV2.f90

```
      FUNCTION gasdv2(idum)
!     Modified version of gasdev for double precision
      INTEGER idum
      DOUBLE PRECISION gasdv2
!U    USES ran12
      LOGICAL iset
      DOUBLE PRECISION fac,gset,rsq,v1,v2,ran12
      SAVE iset,gset
      DATA iset/.true./
      if (iset) then
1       v1=2.0d0*ran12(idum)-1.0d0
        v2=2.0d0*ran12(idum)-1.0d0
        rsq=v1**2+v2**2
        if(rsq.ge.1.0d0.or.rsq.eq.0.0d0)goto 1
        fac=sqrt(-2.0d0*log(rsq)/rsq)
        gset=v1*fac
        gasdv2=v2*fac
        iset=.false.
      else
        gasdv2=gset
        iset=.true.
      end if
      END
```

Chapter 50
Truncated normal distribution

Function gausstrunc(xlow, xupp, idum)
File : gaussTrunc.f90

```
function gausstrunc( xlow, xupp, idum )

!----------------------------------------------------------------------
!
!    Returns a random variable distributed according to a truncated
!    Normal distribution.
!
!    XLOW  - Input . Lower limit of the variable             - Real
!    XUPP  - Input . Upper limit of the variable             - Real
!    IDUM  - Input . A negative integer to initialize the    - Integer
!            Output. random sequence. Thereafter do not
!                    alter this argument between successive
!                    deviates in a sequence
!
!    External functions called:
!      RAN12
!    Fortran  functions called:
!      LOG
!
!----------------------------------------------------------------------

   implicit none

   ! Function
   ! --------

   real(kind=8) :: gausstrunc

   ! Arguments
   ! ---------

   real(kind=8), intent(in) :: xlow, xupp
   integer, intent(inout) :: idum

   ! Local declarations
   ! ------------------

   real(kind=8), external :: ran12

   real(kind=8), parameter :: zero=0.0_8, two=2.0_8
   real(kind=8) :: e, y, z

   !----------------------------------------------------------------------

   if ( xlow <= zero .and . xupp >= zero ) then
      z = zero
   else if ( xlow > zero ) then
      z = -xlow*xlow
   else
      z = -xupp*xupp
   end if
   e = xupp - xlow
   do
```

```
      y = xlow + e*ran12(idum)
      if ( two*log(ran12(idum)) <= -y*y-z ) exit
    end do
    gausstrunc = y

  end function gausstrunc
```

Chapter 51
Uniform distribution

Function ran1(idum)
File : RAN1.f90

```
function ran1( idum )

!    "Minimal" random number generator of Park and Miller with
!    Bays-Durham shuffle and added safeguards.
!    Returns a uniform random deviate between 0.0 and 1.0 (exclusive
!    of the endpoint values). Call with idum a negative integer to
!    initialize; thereafter, do not alter idum between successive
!    deviates in a sequence. RNMX should approximate the largest
!    floating value that is less than 1.

  implicit none

  ! Function
  ! --------

  real :: ran1

  ! Arguments
  ! ---------

  integer, intent(inout) :: idum

  ! Local declarations
  ! ------------------

  integer, parameter :: ia=16807, im=2147483647, iq=127773, ir=2836
  integer, parameter :: ntab=32, ndiv=1+(im-1)/ntab
  real, parameter :: am=1.0/im
  real, parameter :: eps=0.12e-6, rnmx=1.0-eps

  integer :: j, k
  integer, save :: iy=0
  integer, dimension(ntab), save :: iv=(/ (0,j=1,ntab) /)

  if ( idum <= 0 .or. iy == 0 ) then
    idum = max(-idum,1)
    do j = ntab+8, 1, -1
      k = idum/iq
      idum = ia*(idum-k*iq)-ir*k
      if ( idum < 0 ) idum = idum+im
      if ( j <= ntab ) iv(j) = idum
    end do
    iy = iv(1)
  end if
  k = idum/iq
  idum = ia*(idum-k*iq)-ir*k
  if ( idum < 0 ) idum = idum+im
  j = 1 + iy/ndiv
  iy = iv(j)
  iv(j) = idum
  ran1 = min(am*iy,rnmx)

end function ran1
```

Chapter 52
Uniform distribution (modified version

Function ran12(idum)
File : RAN12.f90

```
function ran12( idum )

!    "Minimal" random number generator of Park and Miller with
!    Bays-Durham shuffle and added safeguards.
!    Returns a uniform random deviate between 0.0 and 1.0 (exclusive
!    of the endpoint values). Call with idum a negative integer to
!    initialize; thereafter, do not alter idum between successive
!    deviates in a sequence. RNMX should approximate the largest
!    floating value that is less than 1.
!    Modified version of ran1, returning a real*8 number.

  implicit none

  ! Function
  ! --------

  real(kind=8) :: ran12

  ! Arguments
  ! ---------

  integer, intent(inout) :: idum

  ! Local declarations
  ! ------------------

  integer, parameter :: ia=16807, im=2147483647, iq=127773, ir=2836
  integer, parameter :: ntab=32, ndiv=1+(im-1)/ntab
  real(kind=8), parameter :: am=1.0_8/im
  real(kind=8), parameter :: eps=0.223e-15_8, rnmx=1.0_8-eps

  integer :: j, k
  integer, save :: iy=0
  integer, dimension(ntab), save :: iv=(/ (0,j=1,ntab) /)

  if ( idum <= 0 .or. iy == 0 ) then
    idum = max(-idum,1)
    do j = ntab+8, 1, -1
      k = idum/iq
      idum = ia*(idum-k*iq)-ir*k
      if ( idum < 0 ) idum = idum+im
      if ( j <= ntab ) iv(j) = idum
    end do
    iy = iv(1)
  end if
  k = idum/iq
  idum = ia*(idum-k*iq)-ir*k
  if ( idum < 0 ) idum = idum+im
  j = 1 + iy/ndiv
  iy = iv(j)
  iv(j) = idum
  ran12 = min(am*iy,rnmx)

end function ran12
```