Probability in the Engineering and Informational Sciences, 8, 1994, 291–307. Printed in the U.S.A.

A SET OF ALGORITHMS FOR THE INCOMPLETE GAMMA FUNCTIONS

N. M. TEMME
CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands

This paper gives fast and reliable algorithms for the numerical evaluation of the incomplete gamma functions and for auxiliary functions, such as functions related with the gamma function and error function. All these functions are of basic importance in applied probability problems.

1. INTRODUCTION

The incomplete gamma integral, and its equivalent, the chi-squared distribution, is of basic importance in applied probability problems arising in inventory, queueing, and reliability. The principle idea behind this paper is to make available reliable, fast, and simple algorithms with limited accuracy (say, nine significant digits) that are easily implemented by the practitioner in applied probability. This note intends to fulfill a long-felt need for such algorithms. We give a basic set of algorithms for computing the incomplete gamma integral, including programs for the gamma function and the error functions.

2. AUXILIARY GAMMA FUNCTIONS

The Euler gamma function is defined by

$$\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}\,dt, \qquad x > 0. \tag{2.1}$$

 0269-9648/94 $5.00 + .00

We consider positive values of x. We give approximations for $\Gamma(x)$ and for auxiliary functions. As a rule, we give nearly-best Chebyshev rational approximations for the functions. The rational approximations all give answers accurate to nine significant digits. The coefficients are given in Pascal programs and are computed by the Remes algorithm, with accuracy of 19 significant digits.

2.1. The Function $\Gamma^*(x)$

We need an algorithm for the "tempered" gamma function $\Gamma^*(x)$ defined by

$$\Gamma(x) = \sqrt{2\pi}e^{-x}x^{x-\frac{1}{2}}\Gamma^*(x), \qquad x > 0. \tag{2.2}$$

This function is of fundamental importance in algorithms when x is large, because, on account of Stirling's formula, we have $\Gamma^*(x) = 1 + \mathcal{O}(1/x)$, as $x \to \infty$. For instance, when ratios of gamma functions are used, it is all-important to cancel the dominant parts in the fractions. The dominant part of the gamma function usually causes overflow, whereas the result of a combination of gamma functions may be representable.

For $x > 1$, we use rational approximations; for $x \in (0,1)$, we use the recursion

$$\Gamma^*(x) = e^{-1}\left(\frac{x+1}{x}\right)^{x+\frac{1}{2}}\Gamma^*(x+1).$$

In this way the algorithm is independent from other algorithms for the gamma function.

2.2. Another Auxiliary Function for the Gamma Function

In our algorithm for the incomplete gamma functions, we need the evaluation of $1/\Gamma(1+x) - 1$ for small values of $|x|$. In fact, we use the function $g(x)$ defined by

$$\frac{1}{\Gamma(1+x)} = 1 + x(x-1)g(x), \qquad 0 \le x \le 1. \tag{2.3}$$

This representation shows the vanishing of $1/\Gamma(1+x) - 1$ at $x = 0$, $x = 1$. On the interval $[0,1]$, the function $g(x)$ is computed by using a rational function. In the algorithm for the incomplete gamma function, we also need values on a neighboring interval, especially on $[1,1\frac{1}{4}]$, which can be obtained by a simple stable recursion. For $x > 2$, direct evaluation of $1/\Gamma(1+x) - 1$ is stable when the gamma function itself is used.

2.3. The Gamma Function

We use a rational approximation on the interval $[2,3]$. For $x \in (0,2)$ and $x \in (3,10)$, we use recursion. For $x \ge 10$, we use the asymptotic expansion for the

logarithm of the gamma function. We take care of large values, because overflow occurs very soon.

The special cases $x = n$ or $x = n + \frac{1}{2}$, n integer, occur very frequently in practical problems and the gamma function values can be given very easily. For these reasons we treat these values separately.

2.4. The Logarithm of the Gamma Function

This function is not needed in our programs. We like to draw attention to a robust algorithm given in Macleod [5]. A program in Fortran gives nine significant decimals, with good care for underflow and overflow. Also, care is taken for the vanishing of $\ln \Gamma(x)$ at $x = 1$, $x = 2$. The algorithm is based on rational approximations, as given in Cody and Hillstrom [2]. For convenience we include a Pascal version of Macleod's algorithm in our collection.

Remark: $\Gamma(x)$ and $\ln \Gamma(x)$ may be obtained directly from $\Gamma^*(x)$. However, in that case the computation of the exponential and/or logarithmic function is needed, which may not be as fast as the direct evaluation by using rational approximations. Another point is that the special values of $\ln \Gamma(x)$ at $x = 1$, $x = 2$, will be inaccurate when taking the logarithm of $\Gamma(x)$.

3. AUXILIARY ALGORITHMS FOR THE EXPONENTIAL AND LOGARITHMIC FUNCTIONS

3.1. The Function $\ln(1 + x) - x$

When we need $\ln(1 + x)$ for small values of $|x|$, straightforward use of the standard log-function gives bad relative accuracy. For instance, when $|x|$ is so small that $1 + x = 1$ (on the computer), $\ln(1 + x)$ is just equal to 0, while we may need the correct value, which is of order $x + \mathcal{O}(x^2)$. In the case of the incomplete gamma functions, we need the evaluation of $\ln(1 + x) - x$, which is of order $-\frac{1}{2}x^2 + \mathcal{O}(x^3)$ when $|x|$ is small. To obtain this quantity, we use a rational approximation on the interval [0,1.36]. On [−0.70,0) we use a symmetry rule, namely, the relation

$$\ln(1 + x) - x = -[\ln(1 + z) - z] + xz, \qquad z = -\frac{x}{x + 1}.$$

3.2. The Function $e^x - 1$

We also need an algorithm for the evaluation of $e^x - 1$ for small values of $|x|$. This can be done by the Taylor series of e^x, but again we use rational approximations—in this case on the interval $[\ln \frac{1}{2}, \ln \frac{3}{2}]$. Note that in $\sinh x = (e^x - e^{-x})/2$ also loss of accuracy occurs when $|x|$ is small. However, when we have $y = e^x - 1$, a stable representation is $\sinh x = \frac{1}{2}y(y + 2)/(y + 1)$.

3.3. The Function $x^a e^{-x}/\Gamma(a+1)$

The function

$$D(a,x) := \frac{x^a e^{-x}}{\Gamma(a+1)}$$

is the dominant part in many representations of the incomplete gamma functions. Especially when a and x are large, the computation of this quantity needs some care. We write $D(a,x)$ in the form

$$D(a,x) = \frac{e^{-a[\mu-\ln(1+\mu)]}}{\sqrt{2\pi a}\,\Gamma^*(a)}, \qquad \mu = \frac{x-a}{a}, \tag{3.1}$$

where $\Gamma^*(x)$ is defined in Eq. (2.2). When $|\mu|$ is small, the computation of $\mu - \ln(1+\mu)$ in the exponential function should be done, for instance, by using the algorithm mentioned in Section 3.1. We want to point out that, for large parameters a,x, the accuracy in the computations of the incomplete gamma functions strongly depends on the evaluation of the function $D(a,x)$. Representation (3.1) does not solve all problems, because the well-known loss of accuracy in evaluating the exponential function for a large argument is still a crucial point.

4. THE ERROR FUNCTIONS

We define

$$\operatorname{erf} z = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2}\,dt, \qquad \operatorname{erfc} z = 1 - \operatorname{erf} z = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-t^2}\,dt. \tag{4.1}$$

These functions are used in statistics and probability theory as the *normal distribution functions*, with somewhat different notation. For instance, we have

$$Z(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}, \qquad P(x) = \int_{-\infty}^x Z(t)\,dt, \qquad Q(x) = \int_x^\infty Z(t)\,dt. \tag{4.2}$$

It easily follows that

$$P(x) = \tfrac{1}{2}\operatorname{erfc}(-x/\sqrt{2}), \qquad Q(x) = \tfrac{1}{2}\operatorname{erfc}(x/\sqrt{2}). \tag{4.3}$$

We have the symmetry relations

$$\operatorname{erf}(-x) = -\operatorname{erf} x, \qquad \operatorname{erfc}(-x) = 2 - \operatorname{erfc} x.$$

The error functions are entire functions.

We base the algorithm on rational approximations of Cody [1]. The standard method for real x is to compute directly $\operatorname{erf} x$ and $\operatorname{erfc} x$ indirectly (as $1 - \operatorname{erf} x$) when $\operatorname{erf} x$ is smaller in value. Otherwise, $\operatorname{erfc} x$ is computed directly. We have $\operatorname{erf} x_0 = \operatorname{erfc} x_0$ when $x_0 \cong 0.47$, and we take 0.50 as the changeover point.

Another feature of our algorithm is that we can obtain $e^{x^2}\operatorname{erfc} x$ when x is positive. This is of great importance in numerical algorithms, because the computation of $\operatorname{erfc} x$ causes underflow for quite moderate values of x, due to the dominant term e^{-x^2} in the asymptotic behavior of $\operatorname{erfc} x$.

This all is handled in one master algorithm:

errorfunction (x: double; erfc, expo: boolean): double;

where the booleans *erfc* and *expo* mean the following:

- **if** *erfc* = **true and** *expo* = **false then** *errorfunction* computes $\operatorname{erfc} x$;
- **if** *erfc* = **true and** *expo* = **true and** $x > 0$ **then** *errorfunction* computes $e^{x^2}\operatorname{erfc} x$; and
- **if** *erfc* = **false then** *errorfunction* computes $\operatorname{erf} x$.

In three special-purpose algorithms, the use of *errorfunction* is explained further. The rational approximations are accurate to nine significant decimals.

5. THE INCOMPLETE GAMMA FUNCTIONS

We define

$$\gamma(a,x) = \int_0^x t^{a-1}e^{-t}\,dt, \qquad \Gamma(a,x) = \int_x^\infty t^{a-1}e^{-t}\,dt. \tag{5.1}$$

We assume that a and x are positive numbers. It is useful to define the normalizations

$$P(a,x) = \frac{\gamma(a,x)}{\Gamma(a)}, \qquad Q(a,x) = \frac{\Gamma(a,x)}{\Gamma(a)}, \tag{5.2}$$

of which the sum equals unity.

In statistics and probability theory, one is more familiar with the *chi-squared probability functions*, which are defined by

$$P(\chi^2 \mid \nu) = P(a,x), \qquad Q(\chi^2 \mid \nu) = Q(a,x), \qquad \nu = 2a, \qquad \chi^2 = 2x. \tag{5.3}$$

In other words,

$$\begin{aligned} P(\chi^2 \mid \nu) &= \frac{1}{2^{\nu/2}\Gamma(\frac{1}{2}\nu)} \int_0^{\chi^2} t^{\nu/2-1}e^{-t/2}\,dt, \\ Q(\chi^2 \mid \nu) &= \frac{1}{2^{\nu/2}\Gamma(\frac{1}{2}\nu)} \int_{\chi^2}^\infty t^{\nu/2-1}e^{-t/2}\,dt. \end{aligned} \tag{5.4}$$

When ν is even, we have the *Poisson distribution*, which reads

$$Q(\chi^2 \mid \nu) = \sum_{j=0}^{c-1} e^{-m}\frac{m^j}{j!}, \qquad c = \frac{1}{2}\nu, \qquad m = \frac{1}{2}\chi^2.$$

5.1. Gautschi's Algorithm

Our algorithm for the computation of $P(a,x), Q(a,x)$ is partly based on Gautschi [4]. Another interesting paper in the same spirit is DiDonato and Morris [3]. A rather short Fortran program appeared in the statistical literature in Shea [6], which also compared the results with Gautschi's Fortran program.

We will explain a few aspects of our algorithm. The quarter plane $\{a > 0, x > 0\}$ is divided in two parts, roughly by using the diagonal $a = x$ with small corrections at the origin. When $a \geq x$, $P(a,x)$ is computed directly, and $Q(a,x)$ indirectly as $1 - P(a,x)$. When $x > a$, $Q(a,x)$ is computed directly.

$P(a,x)$ is computed by the Taylor series:

$$P(a,x) = \frac{x^a e^{-x}}{\Gamma(a+1)} \sum_{n=0}^{\infty} x^n \frac{\Gamma(a+1)}{\Gamma(n+a+1)}, \tag{5.5}$$

which is stable and converges quite fast when $a \geq x$. When both parameters are large and of the same order, the convergence slows down. In that case we modify Gautschi's algorithm (see Section 5.2).

For small values of x, that is, when $x \leq 1$, $Q(a,x)$ is computed by using the representation

$$Q(a,x) = 1 - \frac{x^a}{\Gamma(a)} \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{(a+n)n!},$$

in which a different Taylor expansion of $P(a,x)$ is used. The crucial part of the algorithm is the computation of $1 - x^a/\Gamma(a+1)$ (obtained from the first term of the series). When a is small, loss of accuracy may occur in the subtraction; also, when x is near unity and a is near zero or unity, instabilities occur. Writing

$$1 - \frac{x^a}{\Gamma(a+1)} = \left[1 - \frac{1}{\Gamma(a+1)}\right] - \frac{[x^a - 1]}{\Gamma(1+a)},$$

we can control the instabilities by computing the terms between square brackets separately. For computing these terms, the auxiliary algorithms in the previous sections are used.

When $x > 1$, $x > a$, the function $Q(a,x)$ is computed by using Legendre's continued fraction:

$$Q(a,x) = \cfrac{x^a e^{-x}/\Gamma(a)}{x + \cfrac{1-a}{1 + \cfrac{1}{x + \cfrac{2-a}{1 + \cfrac{2}{x + \cfrac{3-a}{1 + \ddots}}}}}} \tag{5.6}$$

This expansion converges for all $x > 0$ and any value of a (also for complex values of a and x). The continued fraction is evaluated by turning the fraction into a series, as described in Gautschi [4]. Convergence is fast when $x \gg a$. Again, convergence slows down when both x and a are large and of the same order.

5.2. Evaluation Based on Uniform Expansions

When a is large, the functions $P(a,\lambda a), Q(a,\lambda a)$ change rapidly when λ crosses the value 1. This sudden change in behavior is the cause of bad convergence of standard expansions (5.5) and (5.6). The asymptotic behavior of the incomplete gamma functions can be described by using the error function (the normal distribution function). In Temme [7] uniform expansions of the incomplete gamma functions in terms of the error function are given; in Temme [8] a numerical algorithm is described based on the uniform representations. We summarize the basic elements of this algorithm.

Let η be the real number defined by

$$\tfrac{1}{2}\eta^2 = \lambda - 1 - \ln\lambda, \qquad \lambda > 0, \qquad \operatorname{sign}(\eta) = \operatorname{sign}(\lambda - 1).$$

Then we write, with $\lambda = x/a$,

$$\begin{aligned} Q(a,x) &= \tfrac{1}{2}\operatorname{erfc}(\eta\sqrt{a/2}) + R_a(\eta), \\ P(a,x) &= \tfrac{1}{2}\operatorname{erfc}(-\eta\sqrt{a/2}) - R_a(\eta). \end{aligned} \tag{5.7}$$

The error functions are the dominant terms that describe the transition at $a = x$. Observe that the property $P + Q = 1$ is reflected in the error functions, because $\operatorname{erfc} x + \operatorname{erfc}(-x) = 1$, and in having the same remainder $R_a(\eta)$. We write

$$R_a(\eta) = \frac{e^{-\frac{1}{2}a\eta^2}}{\sqrt{2\pi a}}\, S_a(\eta). \tag{5.8}$$

From our papers already quoted, it follows that $S_a(\eta)$ is slowly varying for all $\eta \in \mathbb{R}$, that is, for all $\lambda \in (0,\infty)$, where $\lambda = x/a$. We expand

$$S_a(\eta) = \frac{1}{\Gamma^*(a)} \sum_{m=0}^{\infty} b_m(a)\eta^m. \tag{5.9}$$

In Temme [8] we showed that the coefficients b_m can be computed by the recursion

$$b_{m-1}(a) = f_m + \frac{m+1}{a}\, b_{m+1}(a), \tag{5.10}$$

where the numbers f_m are given by

$$f_0 = 1, \qquad f_1 = -\tfrac{1}{3}, \qquad f_2 = \tfrac{1}{12}, \qquad f_3 = -\tfrac{2}{135}, \qquad f_4 = \tfrac{1}{864}, \qquad f_5 = \tfrac{1}{2835}.$$

Further coefficients can be generated by the recursion

$$f_m = -\frac{m+1}{m+2}\left[\frac{m-1}{3m} f_{m-1} + \sum_{j=3}^{m-1} \frac{f_{j-1} f_{m+1-j}}{m+2-j}\right], \qquad m \geq 4.$$

The backward recursion for b_m is stable; we can start with the false initial values

$$b_\nu(a) = f_{\nu+1}, \qquad b_{\nu-1}(a) = f_\nu, \tag{5.11}$$

for some value ν. In the algorithm we use $\nu = 14$. The backward recursion scheme converges better as a increases. Here, "convergence" means that the computed values of $b_m(a)$ obtained by recursion (5.10), and needed in the series in Eq. (5.9), are accurate enough for obtaining the desired precision.

We want to obtain nine-digit accuracy, and we use Eq. (5.9) for $\eta \in (-0.92, 0.19)$. This corresponds with (about) $\lambda \in (0.8, 1.2)$. Then, when we take $a \geq 25$ and only ten terms in the series of Eq. (5.9), we can obtain results that are accurate to nine decimal digits for the computation of $P(a,x), Q(a,x)$ by using the representations of Eq. (5.7).

References

1. Cody, W.J. (1969). Rational Chebyshev approximations for the error function. *Mathematics of Computation* 23: 631–637.
2. Cody, W.J. & Hillstrom, K.E. (1967). Chebyshev approximations for the natural logarithm of the gamma function. *Mathematics of Computation* 21: 198–203.
3. DiDonato, A.R. & Morris, A.H., Jr. (1986). Computation of the incomplete gamma functions. *ACM Transactions on Mathematical Software* 12: 377–393.
4. Gautschi, W. (1979). A computational procedure for incomplete gamma functions. *ACM Transactions on Mathematical Software* 5: 466–481.
5. Macleod, A.J. (1989). A robust and reliable algorithm for the logarithm of the gamma function; algorithm AS 245. *Applied Statistics* 38: 397–423.
6. Shea, B.L. (1988). Chi-squared and incomplete gamma integral. *Applied Statistics* 37: 466–473.
7. Temme, N.M. (1979). The asymptotic expansions of the incomplete gamma functions. *SIAM Journal on Mathematical Analysis* 10: 239–253.
8. Temme, N.M. (1987). On the computation of the incomplete gamma functions for large values of the parameters. In E.J.C. Mason & M.G. Cox (eds.), *Algorithms for approximation.* Proceedings of the IMA-Conference, Shrivenham, July 15–19, 1985, Oxford, Clarendon, pp. 479–489.

APPENDIX: PASCAL PROGRAMS

The Pascal programs were designed on a Macintosh in Think Pascal. No special features of this compiler are used; units are not used. We expect that the program will run on many other compilers and that no significant changes are needed.

The user has to specify three constants:

- *machtol* is the machine precision: the smallest constant such that $1 \pm$ *machtol* is different from unity.
- *giant* is the upper limit of machine-representable numbers (the "overflow" bound).
- *dwarf* is the lower limit of machine-representable numbers (the "underflow" bound).

Of course, these numbers may be slightly larger, smaller, and larger, respectively, than the exact machine constants of the user's computer.

```
program incomgamma;
  const
    machtol = 1.0e−10;
    dwarf = 1.0e−300;
    giant = 1.0e+300
    pi = 3.14159265358979323 85;
  type
    arrcoeff = array[0..15] of double;
  var
    a, h, x, y: double; k: integer; eps: real; ak, bk, bm, fm: arrcoeff;
    sqrtgiant, sqrtdwarf, lndwarf, lnmachtol, sqrtminlnmachtol:double;
    oneoversqrt2mt, explow, sqrtminexplow, exphigh, sqrttwopi: double;
    lnsqrttwopi, sqrtpi, oneoversqrtpi: double;

  procedure computeconstants;
  begin
    sqrtgiant := sqrt(giant); sqrtdwarf := sqrt(dwarf); lndwarf := ln(dwarf);
    lnmachtol := ln(machtol); sqrtminlnmachtol := sqrt(−lnmachtol);
    oneoversqrt2mt := 1 / sqrt(2 * machtol); explow := lndwarf;
    sqrtminexplow := sqrt(−explow); exphigh := ln(giant); sqrttwopi := sqrt(2 * pi);
    lnsqrttwopi := ln(sqrttwopi); sqrtpi := sqrt(pi); oneoversqrtpi := 1 / sqrtpi;
  end; {computeconstants}

  function ratfun (x: double; n, m: integer): double;
    var n1, m1, k: integer; num, den: double;
  begin
    n1 := n − 1; m1 := m − 1;
    num := ak[n]; for k := n1 downto 0 do num := num * x + ak[k];
    den := bk[m]; for k := m1 downto 0 do den := den * x + bk[k];
    ratfun := num / den
  end; {ratfun}
```

```
function exmin1 (x: double): double;
  var y: double;
begin
  if x < lnmachtol then y := −1 else if x > exphigh then y := giant else
  if (x < −0.69) or (x > 0.41) then y := exp(x) − 1.0 else
  if abs(x) < machtol then y := x else
    begin
      ak[0] := 9.999999998390e−1;        bk[0] := 1.000000000000e+0;
      ak[1] := 6.652950247674e−2;        bk[1] := −4.334704979491e−1;
      ak[2] := 2.331217139081e−2;        bk[2] := 7.338073943202e−2;
      ak[3] := 1.107965764952e−3;        bk[3] := −5.003986850699e−3;
     y := x * ratfun(x, 3, 3)
    end;
  exmin1 := y
end; {exmin1}

function auxln (x: double): double;
  var y, z: double; n: integer;
begin
  if x ≦ −1 then y := −giant else if (x < −0.70) or (x > 1.36) then y := ln(1 + x) − x
  else if abs(x) < machtol then y := −0.5 * sqr(x) else
    begin
      ak[0] := −4.999999994526e−1;       bk[0] := 1.000000000000e+0;
      ak[1] := −5.717084236157e−1;       bk[1] := 1.810083408290e+0;
      ak[2] := −1.423751838241e−1;       bk[2] := 9.914744762863e−1;
      ak[3] := −8.310525299547e−4;       bk[3] := 1.575899184525e−1;
      ak[4] := 3.899341537646e−5;
     if x > 0 then y := sqr(x) * ratfun(x, 4, 3) else
      begin
        z := −x / (1 + x); if z > 1.36 then y := −(ln(1 + z) − z) + x * z
        else y := −sqr(z) * ratfun(z, 4, 3) + x * z
      end
    end;
  auxln := y
end; {auxln}

function gammastar (x: double): double;
  var a, g, s: double; j, k: integer;
begin
  if x > 1.0e10 then
    begin if x > 1 / (12 * machtol) then g := 1.0 else g := 1.0 + 1 / (12 * x)
    end else if x ≧ 12.0 then
    begin
      ak[0] := 1.000000000949e+0;        bk[0] := 1.000000000000e+0;
      ak[1] := 9.781658613041e−1;        bk[1] := 8.948328926305e−1;
      ak[2] := 7.806359425652e−2;
     a := 1 / x; g := (ak[0] + a * (ak[1] + a * ak[2])) / (bk[0] + a * bk[1])
    end else if x ≧ 1.0 then
```

```
    begin
        ak[0] := 5.115471897484e−2;      bk[0] := 1.544892866413e−2;
        ak[1] := 4.990196893575e−1;      bk[1] := 4.241288251916e−1;
        ak[2] := 9.404953102900e−1;      bk[2] := 8.571609363101e−1;
        ak[3] := 9.999999625957e−1;      bk[3] := 1.000000000000e+0;
      g := ratfun(x, 3, 3)
    end else if x > dwarf then
    begin
      a := 1.0 + 1.0 / x; g := gammastar(x + 1) * sqrt(a) * exp(−1.0 + x * ln(a))
    end else g := 1.0 / (sqrttwopi * sqrtdwarf);
  gammastar := g;
end; {gammastar}

function gamma (x: double): double;
  var a, g, s, dw: double; j, k, k1, m: integer;
begin
  if x ≦ dwarf then g := 1.0 / dwarf else
    begin
      k := round(x); m := trunc(x); k1 := k − 1;
      if k = 0 then dw := dwarf else dw := (1.0 + x) * machtol;
      if (abs(k − x) < dw) and (x ≦ 15) then
        begin g := 1.0; for j := 2 to k1 do g := g * j end
      else if (abs((x − m) − 0.5) < (1.0 + x) * machtol) and (x ≦ 15) then
        begin g := sqrtpi; for j := 1 to m do g := g * (j − 0.5) end
      else
        begin
          ak[0] := 1.000000000000e+0;      bk[0] := −1.345271397926e−1;
          ak[1] := −3.965937302325e−1;     bk[1] := 1.510518912977e+0;
          ak[2] := 2.546766167439e−1;      bk[2] := −6.508685450017e−1;
          ak[3] := −4.880928874015e−2;     bk[3] := 9.766752854610e−2;
          ak[4] := 9.308302710346e−3;      bk[4] := −5.024949667262e−3;
          if x < 1.0 then g := ratfun(x + 2, 4, 4) / (x * (x + 1.0))
          else if x < 2 then g := ratfun(x + 1.0, 4, 4) / x
          else if x < 3 then g := ratfun(x, 4, 4)
          else if x < 10 then
            begin g := 1.0; a := x;
              repeat a := a − 1.0; g := a * g until a < 3;
              g := g * ratfun(a, 4, 4)
            end else if x < exphigh then
          begin
            a := 1.0 / sqr(x);
            g := (1.0 + a * (−3.33333333333e−2 + a * 9.52380952381e−3)) /
            (12.0 * x); a := −x + (x − 0.5) * ln(x) + g + lnsqrttwopi;
            if a < exphigh then g := exp(a) else g := giant;
          end else g := giant
        end
    end;
  gamma := g
end; {gamma}
```

```
function auxgam (x: double): double;
{function g(x) in 1/Γ(1 + x) = 1 + x*(x − 1)*g(x), 0 ≦ x ≦ 1.0}
  var g: double;
begin
      ak[0] := −5.772156647338e−1;      bk[0] := 1.000000000000e+0;
      ak[1] := −1.087824060619e−1;      bk[1] := 3.247396119172e−1;
      ak[2] := 4.369287357367e−2;       bk[2] := 1.776068284106e−1;
      ak[3] := −6.127046810372e−3;      bk[3] := 2.322361333467e−2;
                                        bk[4] := 8.148654046054e−3;
  if x ≦ −1.0 then g := −0.5 else if x < 0 then
    g := −(1.0 + sqr(x + 1.0) * ratfun(x + 1.0, 3, 4)) / (1.0 − x)
  else if x ≦ 1.0 then g := ratfun(x, 3, 4)
  else if x ≦ 2.0 then g := ((x − 2.0)* ratfun(x − 1, 3, 4) − 1.0) / sqr(x)
  else g := (1 / gamma(x + 1.0) − 1) / (x*(x − 1.0));
  auxgam := g
end; {auxgam}


function lngamma (x: double): double;
{ln Γ(x); rational approximations from Cody & Hillstrom (1967)}
  var a, g, y: double;
begin if x > 12 then
    begin g := 1.0 / (12*x);
      a := −x + (x − 0.5) * ln(x) + lnsqrttwopi; y := a + g; if y = a then g := y
      else
        begin y := 1.0 / sqr(x);
          g := a + g*(1.0 + y*(−3.33333333333e−2 + y*9.52380952381e−3));
        end
    end
  else if x ≧ 4 then
    begin
      ak[0] := −2.12159572323e5;        bk[0] := −1.16328495004e5;
      ak[1] := 2.30661510616e5;         bk[1] := −1.46025937511e5;
      ak[2] := 2.74647644705e4;         bk[2] := −2.42357409629e4;
      ak[3] := −4.02621119975e4;        bk[3] := −5.70691009324e2;
      ak[4] := −2.29660729780e3;        bk[4] := 1.00000000000;
      g := ratfun(x, 4, 4)
    end
  else if x > 1.5 then
    begin
      ak[0] := −7.83359299449e1;        bk[0] := 4.70668766060e1;
      ak[1] := −1.42046296688e2;        bk[1] := 3.13399215894e2;
      ak[2] := 1.37519416416e2;         bk[2] := 2.63505074721e2;
      ak[3] := 7.86994924154e1;         bk[3] := 4.33400022514e1;
      ak[4] := 4.16438922228;           bk[4] := 1.00000000000;
      g := (x − 2) * ratfun(x, 4, 4)
    end
  else if x > 0 then
```

```
    begin
        ak[0] := −2.66685511495;          bk[0] := 6.07771387771e−1;
        ak[1] := −2.44387534237e1;        bk[1] := 1.19400905721e1;
        ak[2] := −2.19698958928e1;        bk[2] := 3.14690115749e1;
        ak[3] := 1.11667541262e1;         bk[3] := 1.52346874070e1;
        ak[4] := 3.13060547623;           bk[4] := 1.00000000000;
      if x ≧ 0.5 then g := (x − 1.0) * ratfun(x, 4, 4)
      else if x > machtol then g := −ln(x) + x * ratfun(x + 1.0, 4, 4)
      else if x > dwarf then g := −ln(x) else g := −lndwarf
    end;
  lngamma := g
end; {lngamma}

function errorfunction (x: double; erfc, expo: boolean): double;
{rational approximations from Cody (1969)}
  var xl, y, z: double; done: boolean;
begin {main of errorfunction}
  if erfc then
    begin
      if x < −sqrtminlnmachtol then y := 2 else if x < −machtol then
       y := 2 − errorfunction(−x, true, false) else if x < machtol then y := 1.0
      else if x < 0.5 then
       begin
         if expo then y := exp(x * x) else y := 1.0;
         y := y * (1.0 − errorfunction(x, false, false))
       end
      else if x < 4 then
       begin if expo then y := 1.0 else y := exp(−x * x);
            ak[0] := 7.3738883116;        bk[0] := 7.3739608908;
            ak[1] := 6.8650184849;        bk[1] := 1.5184908190e1;
            ak[2] := 3.0317993362;        bk[2] := 1.2795529509e1;
            ak[3] := 5.6316961891e−1;     bk[3] := 5.3542167949;
            ak[4] := 4.3187787405e−5;     bk[4] := 1.0000000000;
         y := y * ratfun(x, 4, 4)
       end
      else
       begin done := false; if expo then
           begin xl := 1 / (dwarf * sqrtpi);
             if x > xl then begin y := 0.0; done := true end
             else if x > oneoversqrt2mt then
               begin y := 1.0 / (x * sqrtpi); done := true end
             else begin z := x * x; y := 1.0 end
           end
         else
           begin xl := sqrtminexplow; if x < xl then
               begin z := x * x; y := exp(−z);
                 if x * dwarf > y * oneoversqrtpi then
                   begin y := 0; done := true end
```

```
                end
              else begin y := 0; done := true end
            end;
          if not done then
            begin z := 1.0 / z;
                  ak[0] := −4.25799643553e−2;   bk[0] := 1.50942070545e−1;
                  ak[1] := −1.96068973726e−1;   bk[1] := 9.21452411694e−1;
                  ak[2] := −5.16882262185e−2;   bk[2] := 1.00000000000;
              y := y * (oneoversqrtpi + z * ratfun(z, 2, 2)) / x
            end
        end
    end
  else
    begin
      if x = 0 then
       y := 0
      else if abs(x) > sqrtminlnmachtol then y := x / abs(x)
      else if x > 0.5 then y := 1.0 − errorfunction(x, true, false)
      else if x < −0.5 then y := errorfunction(−x, true, false) − 1.0
      else
       begin
            ak[0] := 2.13853322378e1;      bk[0] := 1.89522572415e1;
            ak[1] := 1.72227577039;        bk[1] := 7.84374570830;
            ak[2] := 3.16652890658e−1;     bk[2] := 1.00000000000;
         z := x * x; y := x * ratfun(z, 2, 2)
       end
    end;
  errorfunction := y
end; {errorfunction}

function erf (x: double): double;
begin erf := errorfunction(x, false, false) end; {erf}

function erfc (x: double): double;
begin erfc := errorfunction(x, true, false) end; {erfc}

function erfctamed (x: double): double;
begin erfctamed :=errorfunction(x, true, true) end; {erfctamed}

procedure incomgam (a, x: double; var p, q: double; eps: real);
  var dp, lnx, mu, auxlnmu: double;

  function alfa (x: double): double;
  begin if x > 0.25 then alfa := x + 0.25 else if x ≧ dwarf
    then alfa := −0.6931 / lnx else alfa := −0.6931 / lndwarf
  end; {alpha}

  function dax: double;
  begin
    mu := (x − a) / a; auxlnmu := auxln(mu); dp := a*auxlnmu − 0.5*ln(2*pi*a);
    if dp < explow then
```

```
      begin
       dp := 0;
       if mu < 0 then begin p := 0.0; q := 1.0 end
       else begin p := 1.0; q := 0.0 end
      end
    else dp := exp(dp) / gammastar(a);
    dax := dp
end; {dax}

procedure qtaylor;
    var r, s, t, u, v, w, xpowa: double;
    {Gautschi's algorithm for Q(a,x), when x < 1}
begin
    r := a * lnx;
    if (r < -0.69) or (r > 0.41) then
      begin xpowa := exp(r); q := xpowa - 1 end
    else
      begin q := exmin1(r); xpowa := q + 1 end;
    s := -a * (a - 1) * auxgam(a); u := s - q * (1 - s);
    p := a * x; q := a + 1.0; r := a + 3.0; t := 1.0; v := 1.0;
    repeat
      p := p + x; q := q + r; r := r + 2; t := -p * t / q; v := v + t
    until abs(t / v) < eps;
    v := a * (1.0 - s) * exp((a + 1.0) * lnx) * v / (a + 1.0);
    q := u + v; p := 1 - q
end; {qtaylor}

procedure ptaylor;
    var c, r: double;
    {Gautschi's algorithm for P(a,x)}
begin p := 1.0; c := 1.0; r := a;
    repeat r := r + 1.0; c := x * c / r; p := p + c until c / p < eps;
    p := p * dp; q := 1 - p
end; {ptaylor}

procedure qfraction;
    var g, r, s, t, tau, ro: double;
    {Gautschi's continued fraction algorithm for Q(a,x)}
begin p := 0; q := (x - 1.0 - a) * (x + 1.0 - a); r := 4 * (x + 1.0 - a);
    s := 1.0 - a; ro := 0; t := 1.0; g := 1.0;
    repeat
      p := p + s; q := q + r; r := r + 8.0; s := s + 2.0; tau := p * (1.0 + ro);
      ro := tau / (q - tau); t := ro * t; g := g + t
    until abs(t / g) < eps;
    q := (a / (x + 1.0 - a)) * g * dp; p := 1 - q
end; {qfraction}

procedure pqasymp;
    var s eta, y, t, u, v: double; m, s: integer; fm, bm: arrcoeff;
```

```
  begin
        fm[ 0] := 1.0000000000e+0;      fm[ 1] := -3.3333333333e-1;
        fm[ 2] := 8.3333333333e-2;      fm[ 3] := -1.4814814815e-2;
        fm[ 4] := 1.1574074074e-3;      fm[ 5] := 3.5273368607e-4;
        fm[ 6] := -1.7875514403e-4;     fm[ 7] := 3.9192631785e-5;
        fm[ 8] := -2.1854485107e-6;     fm[ 9] := -1.8540622107e-6;
        fm[10] := 8.2967113410e-7;      fm[11] := -1.7665952737e-7;
        fm[12] := 6.7078535434e-9;      fm[13] := 1.0261809784e-8;
        fm[14] := -4.3820360185e-9;     fm[15] := 9.1476995822e-10;
    y := -auxlnmu; eta := sqrt(2 * y);
    v := 0.5 * errorfunction(sqrt(a * y), true, true) * gammastar(a) * sqrt(2 * pi * a);
    s := 1; if mu < 0 then s := -1; eta := s * eta;
    bm[14] := fm[15]; bm[13] := fm[14]; u := 0;
    for m := 13 downto 1 do
      begin
        t := fm[m] + (m + 1) * bm[m + 1] / a; u := eta * u + t; bm[m - 1] := t;
      end; u := s * u; p := (u + v) * dp;
    if s = 1 then begin q := p; p := 1.0 - q end else q := 1.0 - p
  end; {pqasymp}

begin {main of incomgam}
  if (a = 0.0) and (x = 0.0) then begin p := 0.5; q := 0.5 end
  else if x = 0.0 then begin p := 0.0; q := 1.0 end
  else if a = 0.0 then begin p := 1.0; q := 0.0 end
  else
    begin
      if x ≦ dwarf then lnx := lndwarf else lnx := ln(x);
      dp := dax; if dp ≧ dwarf then
       begin
         if (a > 25.0) and (abs(mu) < 0.2) then pqasymp else
         if a > alfa(x) then ptaylor else
           begin if x < 1.0 then qtaylor else qfraction end
       end else begin if a > x then p := 0.0 else p := 1.0; q := 1.0 - p end
    end
end; {incomgam}

function checkincgam (a, x: double; eps: real): double;
  {checks the relative accuracy in the recursions}
  {Q(a + 1,x) = Q(a,x) + x^a * exp(-x)/Γ(a + 1)}
  {P(a + 1,x) = P(a,x) - x^a * exp(-x)/Γ(a + 1)}
  var dp, p, q, p1, q1, mu, y: double;
begin mu := (x - a) / a; dp := a * auxln(mu) - 0.5 * ln(2 * pi * a);
  if dp < explow then dp := 0 else dp := exp(dp) / gammastar(a);
  incomgam(a + 1, x, p1, q1, eps); incomgam(a, x, p, q, eps);
  if dp > 0 then
    begin
      if x > a then y := q1 / (q + dp) - 1 else y := (p1 + dp) / p - 1
    end else y := 0;
```

```
    checkincgam:= y
  end; {checkincgam}

begin {main}
  computeconstants; eps := 1.0e-10; k := 0; h := 0;
  repeat {random returns an integer in the range [-32768, 32767]}
    a := machtol + abs(random / 300);
    x := machtol + abs(random / 320);
    y := abs(checkincgam(a, x, eps)); k := k + 1;
    if y > h then begin h := y; writeln(a : 16, x : 16, h, '', k) end;
  until k = 1000;
end.
```