The Cumulative Normal Distribution for Dimensions up to 3 using the qfloat floating-point Library from LCC-WIN32

This implementation gives exactness over almost the 104 digits which the library provides.

The system library `qfloat.dll` should be in Window's system directory and the concurrent `cdfn123.dll` should be placed in the directory of this worksheet.

AVt, Jan 2006

```
> restart;
  kernelopts(version);
```

Maple 10.02, IBM INTEL NT, Nov 8 2005 Build ID 208934

```
> Digits_lcc:=105;
```

Digits_lcc := 105

```
> Digits:=2*Digits_lcc; # greater precision to check results
```

Digits := 210

For using the DLL locate its directory to call external functions from there:

```
> currentdir(): myDLL:=cat(%,`\\cdfn123.dll`);
```

myDLL := "C:_Work\other\LCC_Work\cdfn123\lcc\cdfn123.dll"

Accessing the DLL functions is through strings:

```
> lccstr:=proc(var)
    convert( evalf(parse(convert(var,string)),Digits_lcc + 10), string);
  end proc:
```

The cumulative normal distribution

Define the cumulative normal distribution within Maple

```
> cdfN := x -> 1/2+1/2*erf(1/2*x*2^(1/2));
  pdfN := x -> 1/2*1/Pi^(1/2)*exp(-1/2*x^2)*2^(1/2);
```

$$\mathrm{cdfN} := x \to \frac{1}{2} + \frac{1}{2}\,\mathrm{erf}\left(\frac{1}{2}\,x\,\sqrt{2}\right)$$

$$\mathrm{pdfN} := x \to \frac{1}{2}\,\frac{\mathbf{e}^{(-1/2\,x^2)}\,\sqrt{2}}{\sqrt{\pi}}$$

```
> fct_cdfN := define_external(
    'cdfN_string_Maple',
    'C',
    'x_str'::string[],
    'ret_str'::string[],
    RETURN::integer[4],
    LIB=myDLL):

  cdfN_lcc:=proc(x)
    local X::string, result::string;
    result:=StringTools:-Fill( `0` , Digits_lcc+10);
    X:=lccstr(x):
    if type(parse(X),numeric) then
      fct_cdfN(X,result);
      return parse(result);
    else
```

```
      return 'cdfN_lcc(x)';
    end if;
  end proc: #maplemint(%);
```

This will provide the DLL with memory space (given as a string y_str) to store the results in the DLL. Since update is 'inplace' this will modify the string and its length. As Maple is a symbolic system one should never call this result directly, since this inconsistency for the same object will crash it (just try it and restart ...). So a simple procedure is used as interface.

Test that for inputs (first display yL from DLL and second show Maple's result yM):

```
> xTst:= sqrt(2);
  yL:= cdfN_lcc(xTst):
  evalf(cdfN(xTst)): yM:=evalf(%,105):
  yL; yM;
  `absolute error`=yL-yM;
  evalf((yL-yM)/yM,105):
  `relative error`=evalf(%,16);
```

$$xTst := \sqrt{2}$$

```
0.921350396474857434670610317541304629648033498983151454229968948917358627048005420630\
    991662674072444227077
0.921350396474857434670610317541304629648033498983151454229968948917358627048005420630\
    991662674072444227079
```

$$absolute\ error = -0.2\ 10^{-104}$$

$$relative\ error = -0.2170726802367613\ 10^{-104}$$

... which is the exactness qfloat will provide directly ...

some more test values:

```
> xTst:= -4.2;
  yL:= cdfN_lcc(xTst):
  evalf(cdfN(xTst)): yM:=evalf(%,105):
  yL; yM;
  `absolute error`=yL-yM;
  evalf((yL-yM)/yM,105):
  `relative error`=evalf(%,16);
```

$$xTst := -4.2$$

```
0.000013345749015906338353092117785627370250712739167976443620720867880513553093414456\
    8658209720285026030765360
0.000013345749015906338353092117785627370250712739167976443620720867880513553093414456\
    8658209720285026030765359
```

$$absolute\ error = 0.1\ 10^{-108}$$

$$relative\ error = 0.7493022675670991\ 10^{-104}$$

```
> xTst:= 12.2;
  yL:=cdfN_lcc(xTst):
  evalf(cdfN(xTst)): yM:=evalf(%,105):
  yL; yM;
  `absolute error`=yL-yM;
  evalf((yL-yM)/yM,105):
  `relative error`=evalf(%,16);
```

$$xTst := 12.2$$

```
0.99999999999999999999999999999999999844588021361040649038854144264270487085213522836944\
    894249894745980968391
0.99999999999999999999999999999999999844588021361040649038854144264270487085213522836944\
    894249894745980968391
                              absolute error = 0.
                              relative error = 0.
```

```
> remDigits:=Digits: Digits:=1000:
  xTst:= -32;
  yL:=cdfN_lcc(xTst):
  evalf(cdfN(xTst)): evalf(%): yM:=evalf(%,105):
  yL; yM;
  `absolute error`= yL-yM;
  evalf((yL-yM)/yM,105): `relative error`=evalf(%,16);
  Digits:=remDigits:
```

```
                              xTst := -32
0.54520806035123960919623525038697080788735754690472374940491371852481214290332676 9728\
    9595722169376220837 89 10^-224
```

$$xTst := -32$$

```
0.545208060351239609196235250386970807887357546904723749404913718524812142903326769728\
    95957221693762208378 9 10^-224
```

$$0.545208060351239609196235250386970807887357546904723749404913718524812142903326769728\backslash$$
$$95957221693762208378 9\ 10^{-224}$$

$$0.545208060351239609196235250386970807887357546904723749404913718524812142903326769728\backslash$$
$$9595722169376220837 96\ 10^{-224}$$

$$\text{absolute error} = -0.7\ 10^{-328}$$

$$\text{relative error} = -0.1283913520187208\ 10^{-103}$$

The probability density is provided for completeness only:

```
> fct_pdfN := define_external(
    'pdfN_string_Maple',
    'C',
    'x_str'::string[],
    'ret_str'::string[],
    RETURN::integer[4],
    LIB=myDLL):

  pdfN_lcc:=proc(x)
    local X::string, result::string;
    result:=StringTools:-Fill( `0` , Digits_lcc+10);
    X:=lccstr(x):
    if type(parse(X),numeric) then
      fct_pdfN(X,result);
      return parse(result);
    else
      return 'pdfN_lcc(x)';
    end if;
  end proc:
```

Again some test value (as it is used within the DLL):

```
> xTst:= 52.2;
  yL:=pdfN_lcc(xTst):
  evalf(pdfN(xTst)): yM:=evalf(%,105):
  yL; yM;
  `absolute error`=yL-yM;
  evalf((yL-yM)/yM,105):
  `relative error`=evalf(%,16);
```

$$xTst := 52.2$$

```
0.811749502616212771335375813794655876538297077012935699484149462076766187927225963048\
    9469055235240561306 57 10^-592
0.81174950261621277133537581379465587653829707701293569948414946207676618792722596304 8\
    94690552352405613069 6 10^-592
```

absolute error = $-0.39\ 10^{-695}$

relative error = $-0.4804437806774834\ 10^{-103}$

The Bivariate Case

The bivariate normal distribution can be written as:

```
> pdfN2:= (x,y,rho) ->
    1/sqrt(1-rho^2)/(2*Pi)*exp(-(x^2-2*rho*x*y+y^2)/(2*(1-rho^2)));
  ``;
  cdfN2:= (x,y,rho) ->
    Int(Int( pdfN2(xi,eta,rho),   eta=-infinity..y),xi=-infinity..x);
```

$$\mathrm{pdfN2} := (x, y, \rho) \to \frac{1}{2}\,\frac{\mathbf{e}^{\left(-\frac{x^2-2\rho x y+y^2}{2-2\rho^2}\right)}}{\sqrt{1-\rho^2}\,\pi}$$

$$\mathrm{cdfN2} := (x, y, \rho) \to \int_{-\infty}^{x}\int_{-\infty}^{y} \mathrm{pdfN2}(\xi, \eta, \rho)\, d\eta\, d\xi$$

It is coded within the DLL giving 104 decimal points of precision and can be accessed as follows:

```
> fct_cdfN2 := define_external(
    'cdfN2_string_Maple',
    'C',
    'x_str'::string[],
    'y_str'::string[],
    'r_str'::string[],
    'ret_str'::string[],
    RETURN::integer[4],
    LIB=myDLL):

  cdfN2_lcc:=proc(x,y,r)
    local X::string, Y::string, R::string, result;

    result:=StringTools:-Fill( `0` , Digits_lcc+10);
    X:=lccstr(x);
    Y:=lccstr(y);
    R:=lccstr(r);

    if type(parse(X),numeric) and type(parse(Y),numeric) and
  type(parse(R),numeric) then
      fct_cdfN2(X,Y,R,result);
      return parse(result);
    else
      return 'cdfN2_lcc(x,y,r)';
    end if;
  end proc:
```

Look at some test cases:

```
> xTst:= -16.0;
  yTst:= 6;
  rTst:= 0.11;
  st:=time():
  cdfN2_lcc(xTst,yTst,rTst);
  `seconds`=time()-st;
```

$$xTst := -16.0$$

$$yTst := 6$$

$$rTst := 0.11$$

```
0.63887544005380695933132973424379624257388206411665910862579602740385656013256222125 9\
     5358373116315028940 20 10^-57
```

$$0.638875440053806959331329734243796242573882064116659108625796027403856560132562221259\backslash 535837311631502894020\ 10^{-57}$$

$$seconds = 0.011$$

```
> xTst:= 18.0;
  yTst:= -8;
  rTst:= 0.91;

  st:=time():
  cdfN2_lcc(xTst,yTst,rTst);
  `seconds`=time()-st;
```

$$xTst := 18.0$$

$$yTst := -8$$

$$rTst := 0.91$$

$$0.622096057427178412351599517258818842248871727890027580152376352656860350375808906994\backslash 866020457716697680836\ 10^{-15}$$

$$seconds = 0.016$$

Obviously this is fast.

But difficult to test. For that use a re-formulation:

```
> 'cdfN2(x,y,rho)'= 'Int(pdfN(tau)*cdfN((y-rho*tau)/sqrt(1-rho^2)),tau =
  -infinity .. x)';
```

$$\mathrm{cdfN2}(x, y, \rho) = \int_{-\infty}^{x} \mathrm{pdfN}(\tau)\,\mathrm{cdfN}\left(\frac{y-\rho\,\tau}{\sqrt{1-\rho^2}}\right) d\tau$$

We will check the implementation against moderate test data:

```
> xTst:= 2.0;
  yTst:= -1.0;
  rTst:= 0.61;
```

$$xTst := 2.0$$

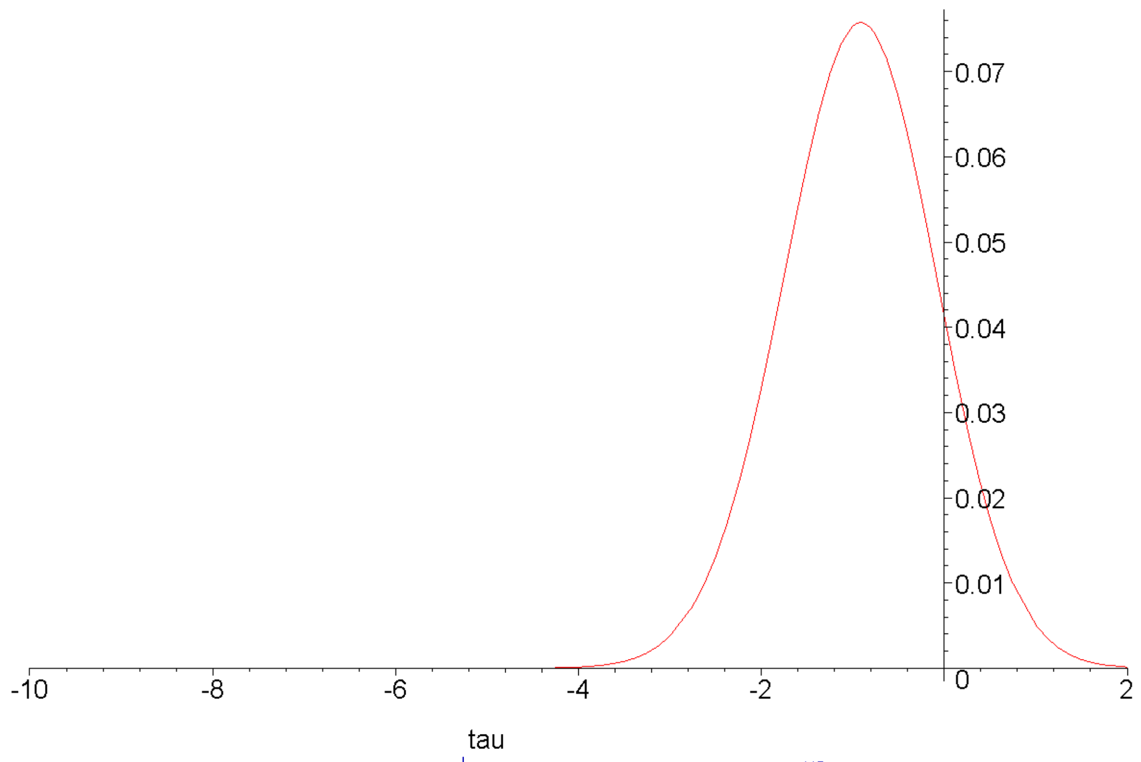
$$yTst := -1.0$$

$$rTst := 0.61$$

The integrand will below `1E-115` if τ is smaller than -23 and looks like a Gaussian:

```
> myIntegrand:='pdfN(tau)*cdfN((y-rho*tau)/sqrt(1-rho^2))';
  myIntegrand:= subs(x=xTst,y=yTst,rho=rTst, myIntegrand):
  plot(myIntegrand, tau=-10..xTst);
  'eval(myIntegrand, tau=-23)': evalf(%): '%%' = evalf(%,15);;
```

$$myIntegrand := \mathrm{pdfN}(\tau)\,\mathrm{cdfN}\left(\frac{y-\rho\,\tau}{\sqrt{1-\rho^2}}\right)$$



$$\left.\text{myIntegrand}\right|_{\tau=-23} = 0.537056036502059\ 10^{-115}$$

Since everything is positive and cdfN is at most 1 the error through cutting off below is at most

```
> 'int(pdfN(tau), tau = -infinity .. cutoff)': '%' =%;
```

$$\int_{-\infty}^{\text{cutoff}} \text{pdfN}(\tau)\, d\tau = \frac{1}{2} + \frac{1}{2}\,\text{erf}\left(\frac{\sqrt{2}\ \text{cutoff}}{2}\right)$$

which is a cdfN and the error will be below 116 decimal points for cutting off at τ = -23:

```
> 'eval(cdfN(cutoff),cutoff=-23.0)': '%'= evalf(%);
```

$$\left.\text{cdfN}(\text{cutoff})\right|_{\text{cutoff}=-23.0} = 0.2330637006220648798573584670433245931588847271077801841105462904 7\backslash$$
$$9012597264225949260981892635 2\ 10^{-116}$$

To compute the integral the interval is split and a Gauss quadrature is used (with 210 digits of exactness,
so be patient on running it, Maple will do it carefully):

```
> Int(myIntegrand, tau=-10 ..xTst, method = _Gquad): I1:=evalf[105](%);
  Int(myIntegrand, tau=-23 .. -10, method = _Gquad): I2:=evalf[105](%);
```

$$I1 := 0.158626030793176988582939568634296108059942154307177990036873067888299192076584 22\backslash$$
$$5100361199185662048268939$$

$$I2 := 0.761985302384975855555181566313263220787565390125863490652578827544721647587220 74\backslash$$
$$8213960590420541087840444\ 10^{-23}$$

Now compare that with the DLL (which gives it in 10 msec):

```
> `I1 + I2 =`;evalf(I1+I2,105);
  `DLL =`;cdfN2_lcc(xTst,yTst,rTst);
  `error =`; cdfN2_lcc(xTst,yTst,rTst) - evalf(I1+I2,105);
```

$$I1 + I2 =$$

```
0.15862603079317698858294718848731995781849770612284112266908094354220045071149075088\
      636646402137920476421
                                    DLL =
0.15862603079317698858294718848731995781849770612284112266908094354220045071149075088\
      636646402137920476422
                                    error =
```

$$0.1\ 10^{-104}$$

Note: within the series approach for the bivariate a decomposition of $\mathrm{cdfN2}(x, y, \rho)$ is made into two terms $\mathrm{cdfN2}(a, 0, r)$ with "`y=0`". Thus the total result usually has limited exactness by the qfloat system exactness given through `QFLT_EPSILON` which is around `1.09E-106`.

One more test case:

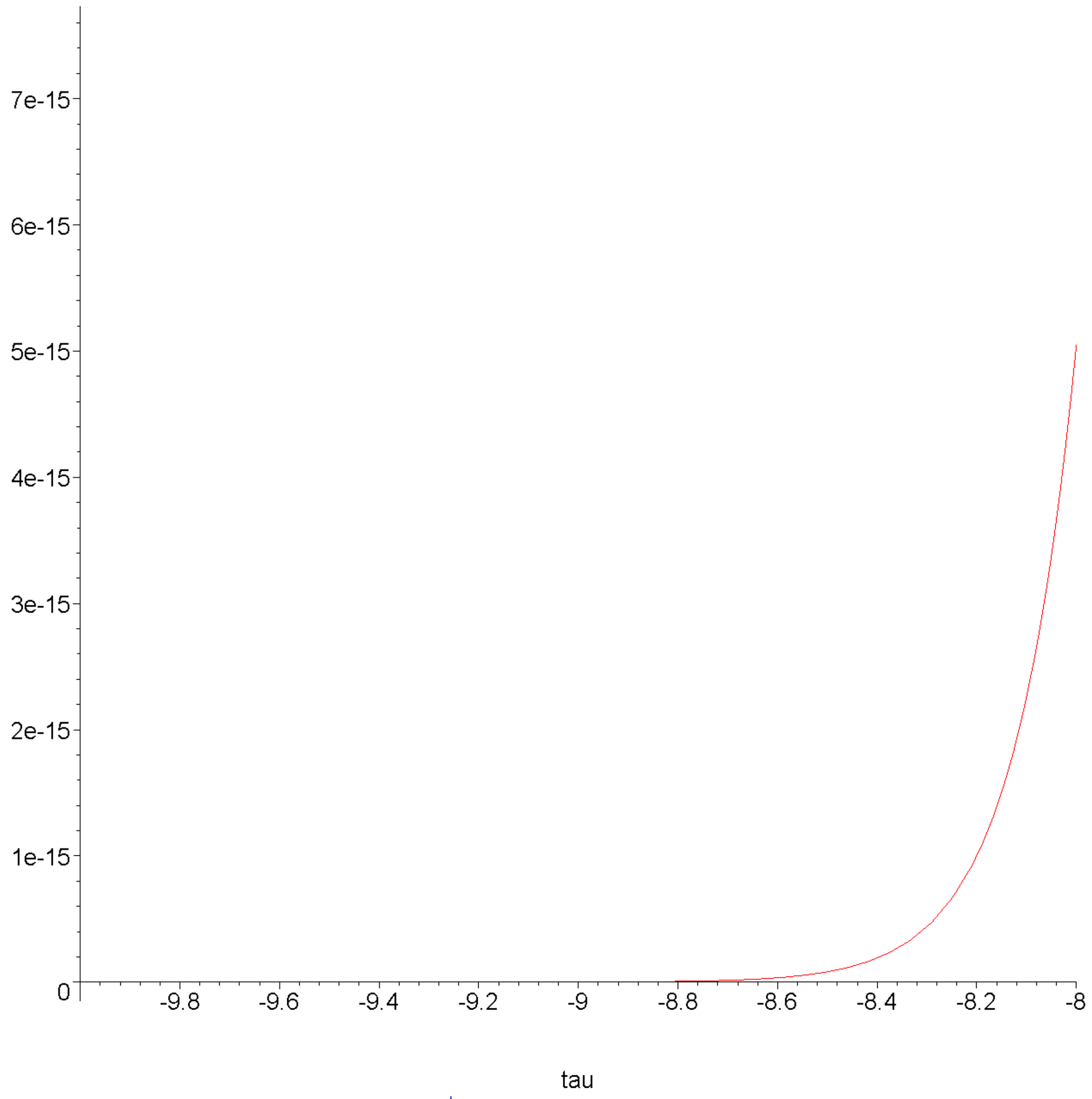
```
> xTst:= -8.0;
  yTst:= 4.0;
  rTst:= 0.4;

  myIntegrand:='pdfN(tau)*cdfN((y-rho*tau)/sqrt(1-rho^2))';
  myIntegrand:= subs(x=xTst,y=yTst,rho=rTst, myIntegrand):
  plot(myIntegrand, tau=-10..xTst);
  'eval(myIntegrand, tau=-23)': evalf(%): '%%' = evalf(%,15);

  Int(`myIntegrand `, tau=-23 ..xTst);
  evalf[105](Int(myIntegrand, tau=-23 ..xTst, method = _Gquad));
  GaussQuadrature:= %:
  `DLL =`;cdfN2_lcc(xTst,yTst,rTst);
  `error =`; cdfN2_lcc(xTst,yTst,rTst) - evalf(GaussQuadrature,105);
```

$$xTst := -8.0$$

$$yTst := 4.0$$

$$rTst := 0.4$$

$$myIntegrand := \mathrm{pdfN}(\tau)\,\mathrm{cdfN}\left(\frac{y-\rho\,\tau}{\sqrt{1-\rho^2}}\right)$$



$$\left.\mathrm{myIntegrand}\right|_{\tau=-23} = 0.537056036502059\ 10^{-115}$$

$$\int_{-23}^{-8.0} \mathrm{myIntegrand}\ d\tau$$

0.622096057427177546423473443317522621574390356104069879434127067456401503552434035773\
890939550080799620000 10^{-15}

DLL =

0.622096057427177546423473443317522621574390356104069879434127067456401503552434035773\
890939549518028029805 10^{-15}

error =

$-0.562771590195\ 10^{-108}$

Certainly I will not do that for the trivariate case: it is not unlikely that a precise computation for the trivariate will need some thousand evaluations for the bivariate normal if it is done through some integration over a bivariate normal, even for a runtime of 10 msec each this will need very much patients using Maple for an exact chross check.

Dimension = 3

The trivariate case can be handled through semi-definite integrals using bivariate normal distributions, cf Alan Genz, *Numerical Computation of Rectangular Bivariate and Trivariate ...* http://www.sci.wsu.edu/math/faculty/genz/homepage

The proper way is to use a beautiful solution due to Plackett, which is given in the above paper and is coded in the DLL for high precision. That needs only integrals over univariate arguments.

To see that the used integration routine is highly exact as an example just compare what happens for integrating over the usual pdfN (that is coded in the DLL only for a test):

```
> fct_int_inf := define_external(
    'integral_infinity_qfloat_Maple',
    'C',
    'x_str'::string[],
    'ret_str'::string[],
    RETURN::integer[4],
    LIB=myDLL):

  # interface
  int_inf:=proc(x)
  local X::string, result;

  result:=StringTools:-Fill( `0` , Digits_lcc+10);
  X:=convert(x,string);

  if type(parse(X),numeric) then
    fct_int_inf(X,result);
    return parse(result);
  else
    return 'x';
  end if;
  end proc:
```

```
> xTst:= 9.0;
  st:=time():
  Int(pdfN(xi), xi=x .. infinity); value(%);: subs(x=xTst,%): evalf(%,800):
  evalf(%,105):
  res1:=%:
  `seconds Maple`=time()-st;
  st:=time():
  res2:=int_inf(xTst):
  `seconds DLL`=time()-st;
  res1;
  res2;
  `difference`=res2-res1;
```

$$xTst := 9.0$$

$$\int_{x}^{\infty} \frac{1}{2} \frac{e^{\left(-\frac{\xi^2}{2}\right)} \sqrt{2}}{\sqrt{\pi}} \, d\xi$$

$$-\frac{1}{2} \operatorname{erf}\left(\frac{\sqrt{2}\ x}{2}\right) + \frac{1}{2}$$

$$\text{seconds Maple} = 0.407$$

$$\text{seconds DLL} = 0.470$$

```
0.11285884059538406477355020759687472579800419008181649488873492287215539304878813487\
    562991729139759339015 10^-18
0.11285884059538406477355020759687472579800419008181649488873492287215539304878813487\
    562991729139362299428 10^-18
                              difference = -0.397039587 10^-114
```

One can see: that is very exact and quite fast (as 104 digits are supported and note, that Maple here does not integrate, it evaluates the error function with 210 digits and needs time for initialization).

But as one can guess from the runtime above it will need much time if the integrand is expensive and I just give two test cases (without checking exactness through Maple here).

For directly inputting constants it not clear that they belong to a correlation matrix, so provide a check:

```
> check_coefficients:=proc(_r12,_r13,_r23)
  local R, det, ev, remDigits;

  remDigits:=Digits;
  Digits:=18;
  R := Matrix(
    [[1,_r12,_r13],
         [1,_r23],
            [1]],
    shape=symmetric,
    scan=triangular[upper]);

  if (LinearAlgebra:-IsDefinite(R, query=positive_definite)) then
    det:=evalf(LinearAlgebra:-Determinant(R), 2*Digits);
    det:=evalf(det,6);
    print(`valid correlation matrix`, `determinant `=det);
    Digits:=remDigits;
    return 0;
  else
    print(`not a correlation matrix!`);
    Digits:=remDigits;
    return -1;
  end if;
  Digits:=remDigits;
  return;
  end proc:
```

Provide an interface to the DLL ...

```
> fct3 := define_external(
    'cdfN3_string_Maple',
    'C',
    'x1_str'::string[], 'x2_str'::string[], 'x3_str'::string[],
    'r12_str'::string[], 'r13_str'::string[], 'r23_str'::string[],
    'ret_str'::string[],
    RETURN::integer[4],
    LIB=myDLL):

  cdfN3_lcc:=proc(x1,x2,x3,r12,r13,r23)
  #local X::string, Y::string, R::string, result;
  local X1, X2, X3, R12, R13, R23, result, lccstr;

  lccstr:=proc(var)
    convert( evalf(parse(convert(var,string)),105), string);
  end proc:

  result:=StringTools:-Fill( `0` , Digits_lcc+10);
  #X1:=convert(x1,string); X2:=convert(x2,string); X3:=convert(x3,string);
  #R12:=convert(r12,string); R13:=convert(r13,string);
  R23:=convert(r23,string);
  X1:=lccstr(x1): X2:=lccstr(x2): X3:=lccstr(x3):
  R12:=lccstr(r12): R13:=lccstr(r13): R23:=lccstr(r23):

  if type(parse(X1),numeric) then
    fct3(X1, X2, X3, R12, R13, R23,result);
    return parse(result);
  else
    return 'x';
  end if;
  end proc:
```

... and let it run with test values:

```
> x1 := 1.000000000;
  x2 := 0.330000000;
  x3 := -0.500000000;

  r12 := 0.9;
  r13 := 0.7;
  r23 := 0.8;

  st:=time():
  if 0 <= check_coefficients(r12,r13,r23) then
    cdfN3_lcc(x1,x2,x3,r12,r13,r23);
    #evalf(%,16);
  end if;
  `seconds`=time()-st;
```

```
                              x1 := 1.000000000
                              x2 := 0.330000000
                             x3 := -0.500000000
                                  r12 := 0.9
                                  r13 := 0.7
                                  r23 := 0.8
                   valid correlation matrix, determinant  = 0.068
0.296114833713582696020468935168923545697048449068019828037433866742829296013568740952\
    543110441612268973184
                               seconds = 4.517
```

If the determinant is close to zero then even more time is needed and for this example
the correlation coefficients are choosen from a valid matrix R (cf the Genz paper):

```
> theta1 := 0.01;
  theta2 := 1 - 1e-6;
  theta3 := -0.02;

  R:='R': A:='A': `R`= A*A^t;
  `A`=Matrix('[[1,0,0],
    [cos(theta1*Pi),sin(theta1*Pi),0],

  [cos(theta2*Pi)*cos(theta3*Pi),cos(theta2*Pi)*sin(theta3*Pi),sin(theta2*Pi
  )]]');
  rhs(%):
  A:=evalf(%):
  LinearAlgebra:-Multiply(A, LinearAlgebra:-Transpose(A)):
  R:=evalf(evalhf(%),15); A:='A':
  ``;
  r12:=R[1,2];
  r13:=R[1,3];
  r23:=R[2,3];
```

$$\theta1 := 0.01$$

$$\theta2 := 0.999999$$

$$\theta3 := -0.02$$

$$R = A\ A^t$$

$$A = \begin{bmatrix} 1 & 0 & 0 \\ \cos(\theta1\ \pi) & \sin(\theta1\ \pi) & 0 \\ \cos(\theta2\ \pi)\cos(\theta3\ \pi) & \cos(\theta2\ \pi)\sin(\theta3\ \pi) & \sin(\theta2\ \pi) \end{bmatrix}$$

$$R := \begin{bmatrix} 1. & 0.999506560365732 & -0.998026728423346 \\ 0.999506560365732 & 1. & -0.995561964598167 \\ -0.998026728423346 & -0.995561964598167 & 1. \end{bmatrix}$$

$$r12 := 0.999506560365732$$

$$r13 := -0.998026728423346$$

$$r23 := -0.995561964598167$$

Here the computation needs not longer to achieve the desired accuracy:

```
> x1 := 1.000000000;
  x2 := 0.330000000;
  x3 := -0.400000000;

  st:=time():
  if 0 <= check_coefficients(r12,r13,r23) then
    print(`cdfN3 =`);
    cdfN3_lcc(x1,x2,x3,r12,r13,r23);
    #evalf(%,16);
  end if;
  `seconds`=time()-st;
```

$$x1 := 1.000000000$$

$$x2 := 0.330000000$$

$$x3 := -0.400000000$$

$$\text{valid correlation matrix, determinant} = 0.973497\ 10^{-14}$$

cdfN3 =

0.00467092361857449107072174226248922727816424154838584910872167864590894871393193533 6\
    70700834723695037797094

$$\text{seconds} = 4.781$$

The result through that approach would be almost immediate if the usual type double is used.