```
   Help
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion <
     (2007+2) //The "#else" part of the code will be freely av
    ailable after the (year of creation of this file + 2)
#else

#include <iostream>

using namespace std;

#include "math_andersen.h"

static double TWO_PI = 6.28318530717958623;
static double SQRT_TWO_PI_INV = 1/sqrt(TWO_PI);

const double GOLD = 1.618034;


////////////////////////////////////////////////////////
    ////////////
//
// computes E(f(X)), where X is normally distributed N(mea
    n,var)
// and f is a function double->double

//
// method: Riemann-type sum

// (the integration is restricted to the interval [mean-l,
    mean+l]
// and then discretized by stepnumber steps)
//
////////////////////////////////////////////////////////
    ////////////
double Normal (double mean, double var, double f(double),
    double intervallength, int stepnumber)
{
  double result = 0;
  double l = intervallength/2;
  double h = intervallength/stepnumber;
  double oldfvalue = f(mean-l);
```

```
  double newfvalue;

  for (int j=0; j<stepnumber; j++)
    {
      newfvalue = f(mean-l+(j+1)*h);
      result    = result + (oldfvalue+newfvalue)/2 * exp( -
     SQR(l-j*h)/(2*var) );
      oldfvalue = newfvalue;
    }

  return SQRT_TWO_PI_INV * h / sqrt(var) * result;
}

////////////////////////////////////////////////////////////
    /////////////////////////
//
// computes E(f(X)), where X is normally distributed N(mea
    n,var)
// and f is of the type discrete_fct (i.e. we only have th
    e values
// f(xleft + j*xstep) = f.val[j] for j=0,...,f.xnumber-1

//
// method: Riemann-type sum

// (the integration is restricted to the interval [xleft, x
    left+(xnumber-1)*xstep]
// and then discretized in the points xleft + j*xstep)
//
////////////////////////////////////////////////////////////
    /////////////////////////
double NormalTab (double mean, double var, discrete_fct *f)
{
  double result = 0;
  double expterm, newterm;

  for (int j=0; j<f->xnumber-1; j++)
    {
      expterm = exp( - SQR(mean-f->xleft-j*f->xstep)/(2*
    var) );
      if (expterm<0) {printf("exp yields a negative result
```

```
    !{n"); exit(1);}
      newterm = (f->val[j] + f->val[j+1])/2 * expterm;
      result  = result + newterm;
    }

  return SQRT_TWO_PI_INV * f->xstep / sqrt(var) * result;
}
```

```
void Set_discrete_fct (discrete_fct *f, double xleft,
    double xstep, int xnumber)
{
  f->xleft = xleft;
  f->xstep = xstep;
  f->xnumber = xnumber;
  f->val = (double*)malloc(xnumber*sizeof(double));
}
```

```
void SetNf (discrete_fct *g, double var, discrete_fct *f)
// Sets g = NormalTab( ř, var, f) such that its domain is
    the set [RHS>eps]
{
  double xleft=-20.,xright,eps=0.0000001,xstep=f->xstep;
  int i;
  //  int xnumber=1;

  while ( (NormalTab(xleft, var, f) <= eps) && (xleft<=20)
    ) xleft+=0.25;
  if (NormalTab(xleft, var, f) <= eps )
    {
      printf("Problem in SetNf !{n"); exit(1);
    }

  if (xstep < 0.001)
    {
      while (NormalTab(xleft, var, f) >  eps)  xleft-=0.025
    ;
```

```
      while (NormalTab(xleft, var, f) <= eps)  xleft+=0.002
    5;
    }

  while (NormalTab(xleft, var, f) > eps)  xleft-=xstep;

  xleft+=xstep;
  // Now we have xleft = min{x; Normal(x,var,f)>eps }


  xright=xleft;
  while (NormalTab(xright, var, f) > eps)  xright+=0.25;


  if (xstep < 0.001)
    {
      while (NormalTab(xright, var, f) <= eps)  xright-=0.0
    25;
      while (NormalTab(xright, var, f) >  eps)  xright+=0.0
    025;
    }

  while (NormalTab(xright, var, f) <= eps) xright-=xstep;

  // Now we have xright = max{x; Normal(x,var,f)>eps }

  Set_discrete_fct( g, xleft, (xright-xleft)/(double)(f->x
    number-1), f->xnumber );
  for (i=0; i<g->xnumber; i++) g->val[i] = NormalTab( g->x
    left+i*g->xstep, var, f);

}




/*
double NfUpBound (discrete_fct *f, double var, double vmax)
// returns the minimum of all x>=f.xleft such that NormalTa
    b(0,var,f*1_{(x,infty)}) < vmax
{
```

```
  double x=f->xleft;
  int j;
  discrete_fct g;

  Set_discrete_fct( &g, f->xleft, f->xstep, f->xnumber);
  for (j=0; j<g.xnumber; j++) g.val[j] = f->val[j];
  // Now g is a copy of f !!

  g.val[0]=0.;
  j=1;
  while (( NormalTab(0.,var,&g) >= vmax ) && (j<f->xnumber)
    )
    {
      g.val[j]=0.; j++; x+=f->xstep;
    }

  if ( NormalTab(0.,var,&g) >= vmax ) printf("Problem in Nf
    UpBounds !{n");

  Delete_discrete_fct(&g);
  return x;
}
*/
```

```
double NfUpBound (discrete_fct *f, double var, double vmax)
// returns the minimum of all x>=f.xleft such that NormalTa
    b(0,var,f*1_{(x,infty)}) < vmax
{
  int i,j=0;
  discrete_fct g;

  if (vmax<0)
    {
      printf("Stupid call of NfUpBounds !{n");
      return 20.;
    }
```

```
  Set_discrete_fct( &g, f->xleft, f->xstep, f->xnumber);
  for (i=0; i<g.xnumber; i++) g.val[i] = f->val[i];
  // Now g is a copy of f !!

  while (( NormalTab(0.,var,&g) >= vmax ) && (j+99<f->xnumb
    er))
    {
      for (i=0; i<100; i++)  g.val[j+i]=0.;
      j+=100;
    }

  if ( NormalTab(0.,var,&g) >= vmax ) {j-=100; printf("%d
    Problem in NfUpBounds !{n",j);}

  while (( NormalTab(0.,var,&g) < vmax ) && (j-10>=0))
    {
      j-=10;
      for (i=0; i<10; i++)  g.val[j+i]=f->val[j+i];
    }

  while (( NormalTab(0.,var,&g) >= vmax ) && (j<f->xnumber)
    )
    {
      g.val[j]=0.; j++;
    }

  Delete_discrete_fct(&g);
  return f->xleft + j*f->xstep;
}




double NfLoBound (discrete_fct *f, double var, double vmin)
// returns the minimum of all x<=f.xleft+(f.xnumber-2)*f.x
    step
// such that NormalTab(0,var,f*1_{(x,infty)}) > vmin
{
  double x=f->xleft + (f->xnumber-2)*f->xstep;
  int j;
  discrete_fct g;
```

```
  Set_discrete_fct( &g, f->xleft, f->xstep, f->xnumber);
  for (j=0; j<g.xnumber; j++) g.val[j] = 0;

  g.val[g.xnumber-1] = f->val[g.xnumber-1];
  j = g.xnumber-2;
  while (( NormalTab(0,var,&g) <= vmin ) && (j>=0))
    {
      g.val[j]=f->val[j]; j--; x-=f->xstep;
    }

  if ( NormalTab(0.,var,&g) <= vmin ) printf("Problem in Nf
    LoBounds !{n");

  Delete_discrete_fct(&g);
  return x;
}



void ShowDiscreteFct(discrete_fct *f)
{
  printf("xleft   = %f{n", f->xleft);
  printf("xstep   = %f{n", f->xstep);
  printf("xnumber = %d{n", f->xnumber);
  printf("(xright = %f){n{n", f->xleft + (f->xnumber-1)*f->
    xstep);
}



double InterpolDiscreteFct(discrete_fct *f, double x)
// returns f(x) via LINEAR interpolation
{
  double xleft=f->xleft, xstep=f->xstep;
  int i=1, xnumber=f->xnumber;
  double x_i, x_iminus1, c;

  if (x<f->xleft) return 0.;
  if (x>xleft+(xnumber-1)*xstep) return 0.;
  if (x==xleft+(xnumber-1)*xstep) return f->val[xnumber-1];
```

```
  while (xleft+i*xstep<=x) i++;
  // Now we have  x_{i-1} <= x < x_i  and  0<i<xnumber
  // Here we denote x_i = xleft+i*xstep

  x_iminus1 = xleft+(i-1)*xstep;
  x_i       = xleft+    i*xstep;
  c         = (x_i-x)/(x_i-x_iminus1);
  // Now we have  x = c*x_iminus1 + (1-c)*x_i  and  0<c<=1

  return c*f->val[i-1] + (1-c)*f->val[i];
}



void ShowDiscreteFctVal(discrete_fct *f)
{
  int j;

  printf("xleft   = %f{n", f->xleft);
  printf("xstep   = %f{n", f->xstep);
  printf("xnumber = %d{n{n", f->xnumber);


  for(j=0; j<f->xnumber; j++)
    {
      printf("val[%d] = %e{n", j, f->val[j]);
      if ((j>0) && (j%100==0)) getchar();
    }

}




void SaveDiscreteFctToFile( discrete_fct *f, char *name)
{
  double x;
  int j;
  FILE *ff;
```

```
  ff = fopen( name, "w");

  for (j=0; j<f->xnumber; j++)
    {
      x = f->xleft + j*f->xstep;
      fprintf( ff, "%f  %f{n", x, f->val[j] );
    }

  fclose(ff);
}
```

```
void SaveArrayToFile( double *tab, int n, char *name)
{
  int j;
  FILE *ff;

  ff = fopen( name, "w");
  for (j=0; j<n; j++)  fprintf( ff, "%d  %f{n", j, tab[j] )
    ;
  fclose(ff);
}
```

```
void Delete_discrete_fct (discrete_fct *f)
{
  free(f->val);
}
```

```
//////////////////////////////////////////
```

```
//                                          //
// Minimization/Maximization of functions //
//                                          //
//////////////////////////////////////////////


void SHFT( double &a, double &b )
{
  double c=a; a=b; b=c;
}


void SHFT( double &a, double &b, double &c, double d )
{
  a=b; b=c; c=d;
}


double SiGn( double a, double b )
{
  if (b>0.) return fabs(a); else return -fabs(a);
}


void InitialMinBracketSB( NumFct1D &f, double &ax, double &
    bx, double &cx )
{
  double h=0.002;
  int ok=0;

  while (ok==0)
    {
      h/=2.; // cout<<"h="<<h<<endl;
      bx=ax;
      while ((f.Eval(bx+h) <= f.Eval(bx)) && (bx<0.1))  bx+
    =h;
      if (bx<0.2)  ok=1;
      cx=bx+h;
    }

  h=0.0001;
```

```
  if ( f.Eval(ax)==f.Eval(bx) )
    {
      ax=bx;
      while ( f.Eval(ax+h) >= f.Eval(ax) )  ax+=h;
      bx=ax+h;
      while ( f.Eval(bx+h) <= f.Eval(bx) )  bx+=h;
      cx=bx+h;
    }


  if ( (f.Eval(bx)>=f.Eval(ax)) || (f.Eval(bx)>=f.Eval(cx))
     )
    {
      cout << "Pbm in InitialMinBracket: f(b) is not the
    strict min. !!"
           << endl;
      cout << "a = " << ax << "   f(a) = " << f.Eval(ax) <<
     endl;
      cout << "b = " << bx << "   f(b) = " << f.Eval(bx) <<
     endl;
      cout << "c = " << cx << "   f(c) = " << f.Eval(cx) <<
     endl;
    }

  //  cout << "InitialMinBracketSB finished !" << endl;

}



void InitialMinBracket( NumFct1D &f, double &ax, double &bx
    , double &cx )
// given f,ax,bx, this routine returns new points ax,bx,cx
    which bracket
// a minimum of f:  ax<bx<cx  and  f(bx)<min(f(ax),f(cx))
{
  double fa,fb,fc,r,q,u,fu,ulim,aux;

  fa=f.Eval(ax);
  fb=f.Eval(bx);
```

```
// assure that f(b)<=f(a)
if (fb>fa)
  {
    SHFT(ax,bx);
    SHFT(fa,fb);
  }

// first guess for c
cx = bx + GOLD*(bx-ax);
fc = f.Eval(cx);

// main loop
while (fb>fc)
  {
    ulim = bx + 1.2*(cx-bx);

    r = (bx-ax)*(fb-fc);
    q = (bx-cx)*(fb-fa);
    u = bx - ((bx-cx)*q - (bx-ax)*r)/
(2.*SiGn( MAX(fabs(q-r),1.0e-15 ) , q-r ));

    if ((bx-u)*(u-cx)>0.)     // u is between b and c
{
  fu=f.Eval(u);
  if (fu<fc)
    {
      ax=bx; bx=u; fa=fb; fb=fu;
      return;
    }
  else if (fu>fb)
    {
      cx=u; fc=fu;
      return;
    }

  u = cx + GOLD*(cx-bx);
  fu=f.Eval(u);
}

    else if ((cx-u)*(u-ulim)>0.)
{
```

```
    fu=f.Eval(u);
    if (fu<fc)
      {
        aux=cx+GOLD*(cx-bx); SHFT(bx,cx,u,aux);
        aux=f.Eval(u);      SHFT(fb,fc,fu,aux);
      }
  }

      else if ((u-ulim)*(ulim-cx)>=0.)
  {
    u=ulim;
    fu=f.Eval(u);
  }
      else
  {
    u = cx+GOLD*(cx-bx);
    fu=f.Eval(u);
  }

      SHFT(ax,bx,cx,u);

    } // end of while-loop


  if (ax>cx) SHFT(ax,cx);
  if ( (ax>=bx) || (bx>=cx) )
    {
      cout << "Pbm in InitialMinBracket: (a<b<c) is false !
    !"
           << endl;
      cout << "  a = " << ax << endl;
      cout << "  b = " << bx << endl;
      cout << "  c = " << cx << endl;
    }
  if ( (f.Eval(bx)>=f.Eval(ax)) || (f.Eval(bx)>=f.Eval(cx))
     )
    {
      cout << "Pbm in InitialMinBracket: f(b) is not the
    strict min. !!"
           << endl;
      cout << "a = " << ax << "   f(a) = " << f.Eval(ax) <<
```

```
     endl;
      cout << "b = " << bx << "   f(b) = " << f.Eval(bx) <<
     endl;
      cout << "c = " << cx << "   f(c) = " << f.Eval(cx) <<
     endl;
    }

}



void GoldenSectionMin1D( NumFct1D &f, double ax, double bx,
     double &xmin )
// given f,ax,bx, this routine computes at first new points
     ax,bx,cx which
// bracket a minimum of f: ax<bx<cx and f(b)<min(f(a),f(c))

// then it performs a Golden Section search for xmin
{
  const double R=0.61803399;
  const double C=1.-R;
  const double tol=0.0001;

  double cx,x0,x1,x2,x3,f0,f1,f2,f3;

  InitialMinBracketSB( f, ax, bx, cx );
  //  cout << "InitialMinBracket finished !" << endl;


  // Initialization of x0,x1,x2,x3
  x0=ax;
  x3=cx;
  if ( fabs(cx-bx) > fabs(bx-ax) )  // b is closer to a th
    an to c
    {
      x1 = bx;
      //      x2 = bx + C*(cx-bx);
      x2 = R*x1 + C*x3;
    }
  else
    {
```

```
      x2 = bx;
      //      x1 = bx - C*(bx-ax);
      x1 = R*x2 + C*x0;
    }
  // Observe that we have x0<x1<x2<x3


  // Initialization of f1,f2
  f1=f.Eval(x1);
  f2=f.Eval(x2);


  // main loop (observe that x0<x1<x2<x3 remains always tru
    e !!)
  while ( fabs(x3-x0) > tol*(fabs(x1)+fabs(x2)) )
    {
      if (f2<f1)
  {
    SHFT(x0,x1,x2,R*x2+C*x3);
    SHFT(f0,f1,f2,f.Eval(x2));
  }
      else
  {
      SHFT(x3,x2,x1,R*x1+C*x0);
    SHFT(f3,f2,f1,f.Eval(x1));
  }
    }


  if (f1<f2) xmin=x1; else xmin=x2;

  // cout << "Golden Section finished !" << endl;
}
```

```
////////////////////////////
//                        //
// Matrices and valarrays //
//                        //
////////////////////////////


double ScalarProd( valarray<double> &x, valarray<double> &
    y )
{
  return (x*y).sum();
}



void VectorProd( valarray<double> &x, valarray<double> &
    mat )
{
  int i,j,dim=x.size();

  for (i=0; i<dim; i++)
    for (j=0; j<dim; j++)
      mat[i*dim+j] = x[i] * x[j];
}



valarray<double> MatrixVectorProd( valarray<double> &M, val
    array<double> &x )
// M is a matrix with D lines and d columns; M_{i,j} = M[i*
    d+j]
// x is a column vector with d entries
// the result M*x is a vector with D entries
{
  int d=x.size(), D=M.size()/d;
  valarray<double> res(0.,D);
```

```
  for (int i=0; i<D; i++)
    for (int j=0; j<d; j++)  res[i]+=M[i*d+j]*x[j];
  return res;
}

#endif //PremiaCurrentVersion
```

References