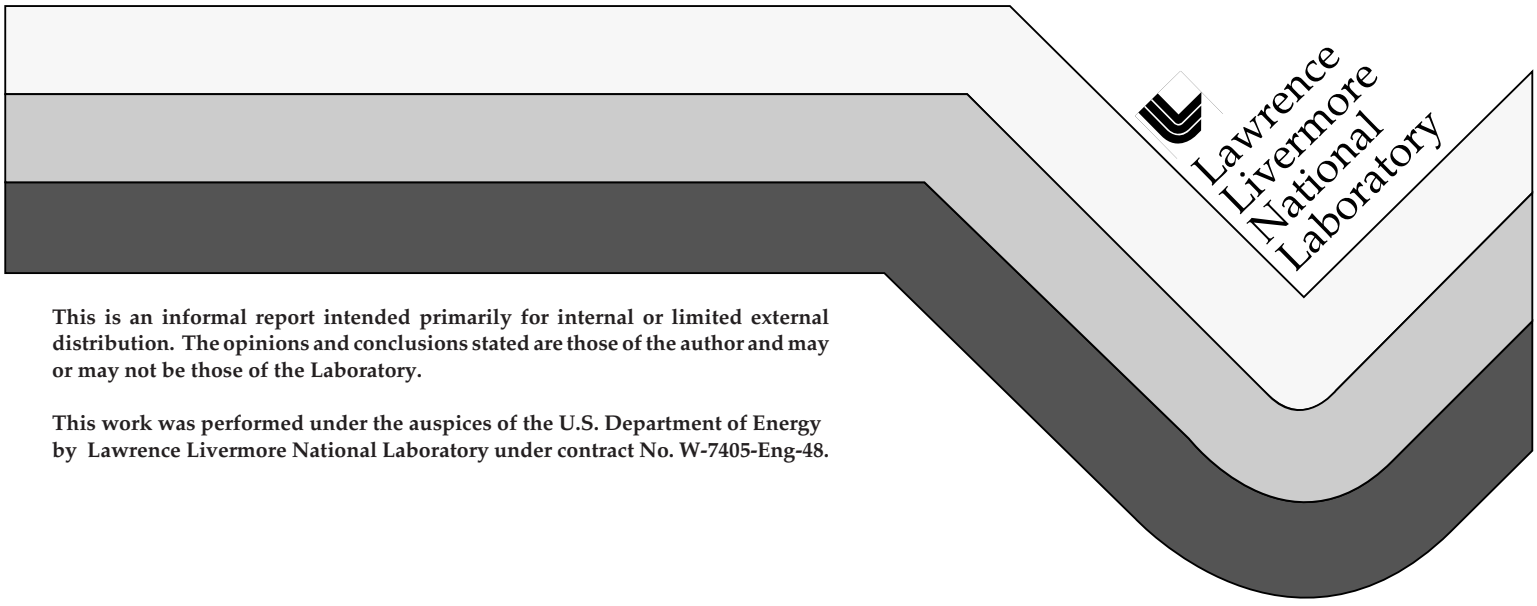
UCRL-ID-125798

Monte Carlo Simulation of Scenario Probability Distributions

Ron Glaser

October 23, 1996



Monte Carlo Simulation of Scenario Probability Distributions

R. E. Glaser, Lawrence Livermore National Laboratory
October 23, 1996

Introduction

Suppose a scenario of interest can be represented as a series of events. For example, consider the following scenario that may occur in the W48 dismantlement process. In the activity of heating the DMSO vessel by hot water, if the water sustains a sufficiently high temperature (say 180 °F) for a long enough time (say one hour), and this problem is not detected because of a temperature sensing and controlling system malfunction, the pit may overheat and crack, causing a mechanical release. The mechanical release R caused by heating the DMSO vessel by hot water may be viewed then as the intersection of three events, A, B, and C, where A is the event that the hot water temperature exceeds 180 °F for an hour, B is the event that the temperature sensing and controlling system malfunctions for longer than one hour, and C is the event that the pit overheats and cracks. The probability of release P(R) in this case is the product $P(R) = P(A)\,P(B \mid A)\,P(C \mid A \cap B)$. An expert may be reluctant to estimate P(R) as a whole yet agree to supply his notions of the component probabilities P(A), P(B|A), and $P(C \mid A \cap B)$ in the form of prior distributions. Each component prior distribution may be viewed as the stochastic characterization of the expert's uncertainty regarding the true value of the component probability. Mathematically, the component probabilities are treated as independent random variables and P(R) as their product; the induced prior distribution for P(R) is determined which characterizes the expert's uncertainty regarding P(R). It is typically difficult if not impossible to obtain analytically a closed form expression for the probability distribution of the product of a series of random variables. Furthermore, in applications such as the above W48 process example, the component priors supplied by experts may be "ball park" in character. Consequently, it may be both convenient and adequate to approximate the desired distribution by Monte Carlo simulation.

Software (see the Appendix) has been written for this task that allows a variety of component priors that experts with good engineering judgment might feel comfortable with. The priors are mostly based on the so-called

likelihood classes, L1 through L5, which essentially span the realm of the possible in general terms as L1: very likely; L2: likely; L3: possible; L4: unlikely; and L5: highly unlikely. For the sake of quantification in the W48 dismantlement setting, the likelihood classes have been defined in $\log_{10}$ space to be L1: -2 to 0; L2: -3 to -2; L3: -5 to -3; L4: -6 to -5; and L5: under -6. Thus the L2 class, for instance, concerns probabilities between 10^{-3} and 10^{-2}, i.e., between 0.001 and 0.01. That this range represents something "likely" must be understood in the context of an extremely undesirable event, such as a cracked pit. The software permits an expert to choose for a given component event probability one of six types of prior distributions, and the expert specifies the parameter value(s) for that prior. Each prior is unimodal. The expert essentially decides where the mode is, how the probability is distributed in the vicinity of the mode, and how rapidly it attenuates away. Limiting and degenerate applications allow the expert to be vague or precise. The six types of priors are described in the next section.

In cases where there are several (or at least two) scenarios of interest, it may be desirable to characterize the uncertainty that any of the scenarios occur. For instance, there may be a number of scenarios that lead to mechanical release, or to combustion release, or to violent reaction, or that involve manual handling, or that involve a spill, etc. Thus, there is a collection, R_1, ..., R_n, of scenarios, each of which is a series of events, and it is desired to characterize the probability $P(\cup_i R_i)$. The software of the Appendix handles this general case as well as the single scenario case, n=1.

Component prior distributions

The following six families of probability distributions have been built into the software of the Appendix to model the uncertainty an expert feels about the probability of a component event in the series sequence of events, A, B, C, ..., that depict a scenario, R. The software may of course be enhanced to include additional or modified prior distribution types if this is mandated.

Type 1: Triangular on a single class. To select this type of prior, the expert must believe the component event probability is confined to a particular likelihood class. The distribution is determined by specifying the class number and the modal value within the class: for example, class L2, modal probability 0.002, i.e., modal value $\log_{10}(0.002)$

= -2.70. The resultant density is shown in Figure 1. If class L5 is selected, a left limit of the class must be specified, e.g. -8. If the expert does not wish to specify a left limit, the default value -10 is used.

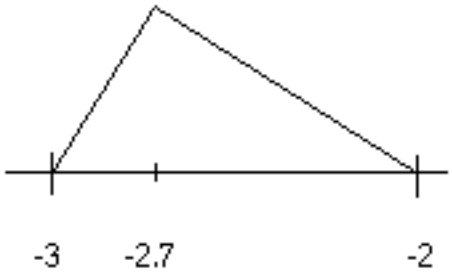


Figure 1. A Type 1 density.

Type 2: Triangular over more than one class. This prior, similar to that of Type 1, spreads the uncertainty over a larger range. Here the expert specifies the leftmost class, the rightmost class, and the modal value. If the leftmost class is L5, the left limit of the class must also be specified (or a default value of -10 will be used). To illustrate this type of prior, suppose the expert selects a modal probability of 0.0005 (which implies the modal value $\log_{10}(0.0005) = -3.30$ and modal class L3) and leftmost class L4 and rightmost class L3. The requested density is shown in Figure 2.

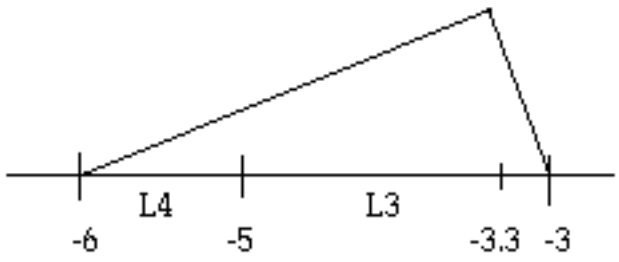


Figure 2. A Type 2 density.

Type 3: Quasi-triangular distribution. For this prior the expert selects a modal class, the probability content q_1 of the class, and the

modal value m within the class. In addition, he gives the total probability q_2 to the left of the modal class and specifies the leftmost class having positive probability. Likewise, he gives the total probability q_3 to the right of the modal class and specifies the rightmost class having positive probability. Necessarily $q_1 + q_2 + q_3 = 1$. As in Types 1 and 2, if L5 is involved, a left limit must be specified or the default value -10 will be used. Shown in Figure 3 is the Type 3 prior density corresponding to the following specifications: modal class = L3; modal value -3.30; $q_1 = 0.7$; $q_2 = 0.1$ with leftmost class L4; and $q_3 = 0.2$ with rightmost class L2.

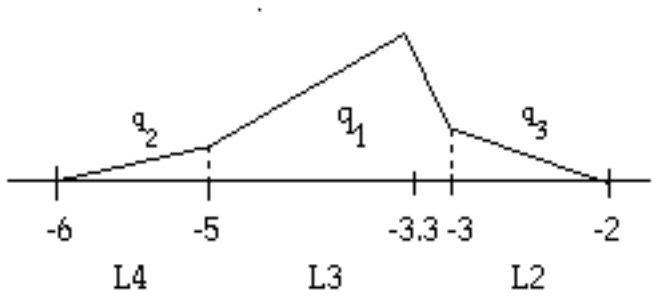


Figure 3. A Type 3 density.

The q_2 and q_3 portions are right triangles, and the central q_1 portion is polygonal with a triangular top. A pair of simple inequalities must be satisfied in order that the density be unimodal. If the expert wishes, q_2 or q_3 may be set to 0, which causes the density to drop to 0 at the left or right limit of the modal class, respectively.

Type 4: Histogram. Here the expert characterizes his prior notion of the component event probability by specifying for each class Li the chance (or belief) p_i that the probability resides in this class. Any of the p_i may be 0, although necessarily $p_1 + p_2 + p_3 + p_4 + p_5 = 1$. Moreover, within a likelihood class the density is constant, which means the expert places equal credibility on each point in the class. Simple inequalities involving the p_i must hold to ensure that the histogram density is unimodal. If $p_5 > 0$, a left limit of L5 must be specified, e.g. -8. If the expert does not wish to specify a left limit, the default value -10 is used. An example of a Type 4 prior is the case shown in Figure 4 in

which the expert specifies the probabilities $(p_1, p_2, p_3, p_4, p_5) = (0.181, 0.600, 0.181, 0.033, 0.005)$. This is the prior that corresponds to having L2 as the modal class, 0.6 as the modal class probability, and having neighboring class probabilities fall off geometrically.

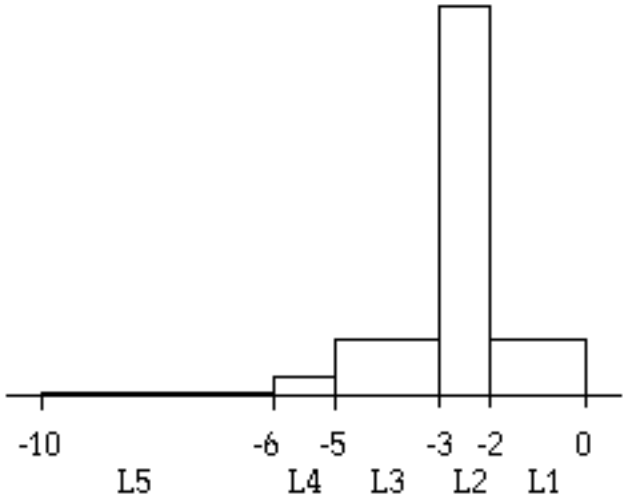


Figure 4. A Type 4 density.

Type 5: Lognormal density. Consider the component event that the operator fails to install properly the protective cover on the sample hose. The probability of this event depends on the operator: some operators are more skilled and dependable and consequently have a lower failure frequency than others. A lognormal distribution on operator failure probabilities may be an adequate model. To illustrate, suppose the median failure frequency of operators is 0.01, and the fifth percentile failure frequency is 0.00143 (i.e. only 5 percent of operators have a failure frequency below 0.00143). The lognormal fit to these parameters, displayed in Figure 5, is a normal distribution in $\log_{10}$ space with mean $\log_{10}(0.01) = -2.0$ and standard deviation $[\log_{10}(0.01) - \log_{10}(0.00143)]/1.645 = 0.513$. To implement this type of prior, the expert must specify the mean and standard deviation of the distribution in $\log_{10}$ space. These parameters can be determined

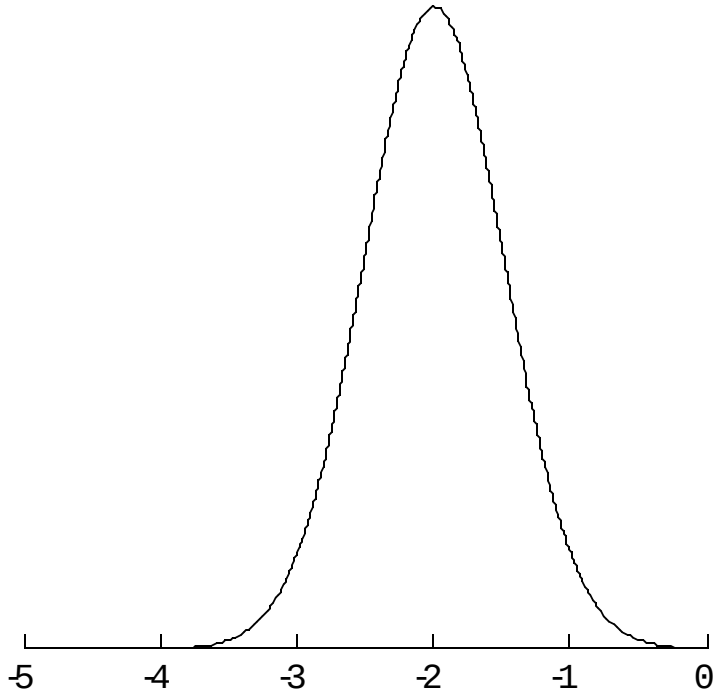


Figure 5. The normal density corresponding to a Type 5 density.

algebraically from any two percentiles, as illustrated above for the median and fifth percentile probabilities. In general, if the expert desires the quantile probabilities p_α and p_β for quantiles α and β (e.g., $p_\alpha = 0.01$ and $p_\beta = 0.00143$ for $\alpha = 0.5$ and $\beta = 0.05$), then the expert must supply the corresponding mean μ and standard deviation σ, which are calculated as $\sigma = [\log_{10}(p_\alpha) - \log_{10}(p_\beta)]/(z_\alpha - z_\beta)$ and $\mu = \log_{10}(p_\alpha) - z_\alpha \sigma$, where z_θ is defined by $\theta = \Phi(z_\theta)$, Φ being the standard normal CDF.

Type 6: Known probability. If there is sufficient data regarding the event in question, the expert may be willing to invoke a degenerate prior that puts probability one at a particular value. For example, consider the component event that the hoist drops the weapon assembly. If this type of hoist failure has occurred historically with relative frequency 1/2000, it may be reasonable to assume a probability of 1/2000 for such a failure on a given attempted hoist operation. Since computations are made in $\log_{10}$ space, a prior with probability one at $\log_{10}(0.0005) = -3.30$ is appropriate.

Monte Carlo Simulation

A Monte Carlo simulation code has been written by the author (see the Appendix) to generate, in the single scenario (n=1) case, the prior distribution of the annualized scenario probability, $P_a(R)$, and in the general case, the annualized probability, $P_a(\cup_i R_i)$, that any of the n scenarios occur. Inputs include the expert-supplied component priors. The annualized probability of a scenario is the probability that the scenario would occur at least once in a year's time assuming that the activity that gives rise to the scenario (e.g., heating the DMSO vessel) would take place on M independent occasions during the year. Thus, $P_a(R) = 1 - [1 - P(R)]^M$, and in general, $P_a(\cup_i R_i) = 1 - \prod_i [1 - P(R_i)]^{M_i}$. [M is the variable "yrfac" in the code. Of course, setting yrfac = 1 gives P(R).] For a given scenario (collection of scenarios), one million random values of $P_a(R)$ ($P_a(\cup_i R_i)$) are created by simulation. These values are summarized in the output files, and as demonstrated in the Appendix, graphical approximations of the cumulative distribution function (CDF) and probability density function (pdf) of $P_a(R)$ ($P_a(\cup_i R_i)$) are readily obtained.

The code employs a random number generator ("ran1") of Park and Miller as referenced in Press et al. (1992). For Types 1 through 4, the inverse of the cumulative distribution function is used to simulate the component prior. Specifically, if X has the CDF F, then $F^{-1}(U)$ has the same distribution as X, where U is a standard uniform variate over [0, 1] (Mood, Graybill, and Boes, p. 202, 1974). Consequently, from a random sample of uniform deviates U_1, ..., U_N, a random sample of deviates with the distribution of X is generated, namely $F^{-1}(U_1)$, ..., $F^{-1}(U_N)$. The Box-Muller transformation (Parzen, p. 334, 1960) is used to generate the Gaussian variates for component Type 5. For Type 6, no randomness is generated since an unchanging component probability value is used in each simulation.

Because of the short-tailed character of all the prior types, it appears that Monte Carlo sampling with only 1000000 iterations provides sufficient accuracy in approximating the distribution of $P_a(R)$ or $P_a(\cup_i R_i)$.

References

Mood, A. M., Graybill, F. A., & Boes, D. C. 1974. *Introduction to the Theory of Statistics*, 3rd ed. New York: McGraw-Hill.

Parzen, E. 1960. *Modern Probability Theory and Its Applications*. New York: John Wiley & Sons.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 1992. *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge, England: Cambridge University Press.

Appendix

A listing of the Monte Carlo simulation code is presented here, along with a tutorial on construction of the input file and the manipulation of the output files to produce graphical approximations of the scenario CDF and pdf.

The code. The following C language code implements the Monte Carlo simulation to characterize the expert-induced prior probability distribution of $P_a(R)$ ($P_a(U_i R_i)$). This code may be compiled by a Think C compiler for the Power Macintosh. A copy of the Macintosh application is available from the author.

```
# include <stdlib.h>
# include <math.h>
# include <stdio.h>
# include <time.h>
# define N 10000
# define NN 40
# define JJ 10
# define NIT 1000000
# define ten 10.
# define IA 16807
# define IM 2147483647
# define AM (1.0/IM)
# define IQ 127773
# define IR 2836
# define NTAB 32
# define NDIV (1+(IM-1)/NTAB)
# define EPS 1.2e-7
# define RNMX (1.0-EPS)

int *okay, inow=0, idum, *numk, ss, jj6;
double *y, *p, *lefts, *left, *lwidth, *right, *rwidth,
      *pl, *ql, *q2k, *ck, *ack, *fmk, *ak,
      *con1, *con2, *con3, *con4, *con5, *con6,
      *p3k, *bk, *dk, *dbk, *q3k, *range, *un,
      *aaa, *rat, *conp, *lmean, *lsd, *yrfac,
      pi2, gauss1, gauss2, lten;

void initialize();
void findroom();
void punt();
int coded();
int chkmod();
int norse();
int unimod();
void simulate();
double getrany();
double randy();
double l10();
void sim2();
double gauss();
void ggrand();
```

```
void hpsort();
double ran1();

main()
      {
       initialize();
       simulate();
       hpsort();
       sim2();
      }

void initialize()
      {
       int j, bigm, itype, i1, ll, rr, i, ichk, idef, k, ii, ii6,
           ij, it, iz, ir, i4, i5, iq;
       unsigned int starter;
       char scene[20];
       double m, lt, rt, p1, lwid, rwid, xi, q2, q3, x,
            aa[6]={0., -10., -6., -5., -3., -2.},
            rang[6]={0., 4., 1., 2., 1., 2.},
            lefty[6]={0., -2., -3., -5., -6., -10.};
       FILE *in;
       in=fopen("inrisky.c", "r");
       starter = (unsigned int) clock();
       idum= -starter; x=ran1();
       pi2=8.*atan(1.); lten=log(ten); jj6=6*JJ;
       fscanf(in, "%ld", &ss);
       if (ss>NN) {exit(1); printf("\ntoo many scenarios\n");}
       findroom();
       for (ii=0; ii<ss; ii++)
            {ii6=6*ii; ij=JJ*ii; iz=jj6*ii; i4=ii6+4; i5=i4+1;
             for (i=0; i<6; i++) lefts[ii6+i]=lefty[i];
             fscanf(in, "%s", scene); printf("%s", scene);
             printf("\n");
             fscanf(in, "%lf", yrfac+ii);
             fscanf(in, "%ld %ld", &k, &idef); numk[ii]=k;
             if (k>JJ) {exit(1); printf("\ntoo many
                        components\n");}
             for (j=0; j<k; j++)
                  {fscanf(in, "%ld %ld", &bigm, &itype);
                   i1=coded(itype); it=ij+j; iq=iz+6*j;
                   if (i1>3) {okay[it]=i1+1;
                              if (i1==4) fscanf(in, "%lf",
                                         conp+it);
                              else fscanf(in, "%lf %lf", lmean+it,
                                         lsd+it);}
                   else {if (i1>0)
                           {fscanf(in, "%lf", &m);
                            ichk=chkmod(ii, bigm, m);
                            if (ichk==0) okay[it]=0;
                            if (i1==1) ll=rr=bigm;
                            else fscanf(in, "%ld %ld", &ll, &rr);
                            if (idef==1 && ll==5) fscanf(in,
                                                 "%lf",&lt);
                            else lt=lefts[ii6+ll];
                            rt=lefts[ii6+rr-1];
                            if (i1<3)
                              {lwid=m-lt; rwid=rt-m;
```

```
                                p1=lwid/(rt-lt);
                                left[it]=lt; lwidth[it]=lwid;
                                right[it]=rt; rwidth[it]=rwid;
                                pl[it]=p1; ql[it]=1.-p1;
                                okay[it]=1;}
                             else {fscanf(in, "%lf %lf", &q2, &q3);
                        okay[it]=norse(ii, j, q2, q3, lt,
                        lefts[ii6+bigm], m, lefts[ii6+bigm-1], rt);}}
                             else {for (i=1; i<=5; i++) fscanf(in,
                                                  "%lf", &p[ii6+i]);
                                for (i=1; i<=5; i++) {ir=iq+i;
                                range[ir]=rang[i]; aaa[ir]=aa[i];}
                                if (idef==1 && p[i5]>0)
                                     {fscanf(in, "%lf", lefts+i5);
                                     range[iq+1]=lefts[i4]-
                                   lefts[i5]; aaa[iq+1]=lefts[i5];}
                               okay[it]=unimod(ii, j, i1, bigm);}}}
              i=0;
              for (j=0; j<k; j++)
                   {if (okay[ij+j]==0)
                       {printf("\noops %d %d",ii+1, j+1); i=1;}}
              if (i==1) exit(1);}
        }

  void findroom()
        {
         int sdub, ns, na, nb, nc, slin, sa;
         sdub=sizeof(double); ns=NN*sdub;
         slin=NN*sizeof(int); sa=slin*JJ;
         na=JJ*ns; nb=6*na; nc=6*ns;
         okay=(int*) malloc(sa);
         if (okay==NULL) punt();
         numk=(int*) malloc(slin);
         if (numk==NULL) punt();
         y=(double*) malloc((N+1)*sdub);
         if (y==NULL) punt();
         p=(double*) malloc(nc);
         if (p==NULL) punt();
         lefts=(double*) malloc(nc);
         if (lefts==NULL) punt();
         left=(double*) malloc(na);
         if (left==NULL) punt();
         lwidth=(double*) malloc(na);
         if (lwidth==NULL) punt();
         right=(double*) malloc(na);
         if (right==NULL) punt();
         rwidth=(double*) malloc(na);
         if (rwidth==NULL) punt();
         pl=(double*) malloc(na);
         if (pl==NULL) punt();
         ql=(double*) malloc(na);
         if (ql==NULL) punt();
         q2k=(double*) malloc(na);
         if (q2k==NULL) punt();
         ck=(double*) malloc(na);
         if (ck==NULL) punt();
         ack=(double*) malloc(na);
         if (ack==NULL) punt();
```

```
       fmk=(double*) malloc(na);
       if (fmk==NULL) punt();
       ak=(double*) malloc(na);
       if (ak==NULL) punt();
       con1=(double*) malloc(na);
       if (con1==NULL) punt();
       con2=(double*) malloc(na);
       if (con2==NULL) punt();
       con3=(double*) malloc(na);
       if (con3==NULL) punt();
       con4=(double*) malloc(na);
       if (con4==NULL) punt();
       con5=(double*) malloc(na);
       if (con5==NULL) punt();
       con6=(double*) malloc(na);
       if (con6==NULL) punt();
       p3k=(double*) malloc(na);
       if (p3k==NULL) punt();
       bk=(double*) malloc(na);
       if (bk==NULL) punt();
       dk=(double*) malloc(na);
       if (dk==NULL) punt();
       dbk=(double*) malloc(na);
       if (dbk==NULL) punt();
       q3k=(double*) malloc(na);
       if (q3k==NULL) punt();
       range=(double*) malloc(nb);
       if (range==NULL) punt();
       un=(double*) malloc(nb);
       if (un==NULL) punt();
       aaa=(double*) malloc(nb);
       if (aaa==NULL) punt();
       rat=(double*) malloc(nb);
       if (rat==NULL) punt();
       conp=(double*) malloc(na);
       if (conp==NULL) punt();
       lmean=(double*) malloc(na);
       if (lmean==NULL) punt();
       lsd=(double*) malloc(na);
       if (lsd==NULL) punt();
       yrfac=(double*) malloc(ns);
       if (yrfac==NULL) punt();
      }

void punt()
      {
       printf("need more memory\n"); exit(1);
      }

int coded(itype)  int itype;
      {
       if (itype<4 || itype==5) return itype;
       if (itype==4) return 0;
       return 4;
      }

int chkmod(ii, bigm, m)  int ii, bigm; double m;
      {
```

```
       int ib;
       ib=6*ii+bigm-1;
       if (m>lefts[ib]) return 0;
       if (bigm==5) return 1;
       return (m<lefts[ib+1] ? 0 : 1);
      }

int norse(ii, j, q2, q3, c, a, m, b, d)  int ii, j; double q2, q3,
                                                    c, a, m, b, d;
      {
       int it;
       double ac, db, xi2, c1, c2, ma, bm, p3;
       ac=a-c; c1=(q2>0. ? q2/ac : 0.);
       db=d-b; c2=(q3>0. ? q3/db : 0.);
       ma=m-a; bm=b-m; p3=1.-q3;
       xi2=(p3-q2-ma*c1-bm*c2)/(b-a);
       if (xi2<c1 || xi2<c2) return 0;
       it=ii*JJ+j;
       q2k[it]=q2; ck[it]=c; ack[it]=ac;
       fmk[it]=q2+ma*(xi2+c1); ak[it]=a; con1[it]=c1;
       con2[it]=c1*c1; con3[it]=(xi2-c1)/ma;
       p3k[it]=p3; bk[it]=b; con4[it]=c2;
       con5[it]=c2*c2; con6[it]=(xi2-c2)/bm;
       dk[it]=d; dbk[it]=db; q3k[it]=q3;
       return 2;
      }

int unimod(ii, j, i1, m)  int ii, j, i1, m;
      {
       int i, iok, i6, i66, iq, iq6, iqi;
       double z=0., value[6], w;
       i6=6*ii; i66=i6+6; iq=jj6*ii+6*j; iq6=iq+6;
       un[iq]=0.;
       for (i=1; i<=5; i++)
            {iqi=iq+i; w=p[i66-i]; z+=w; un[iqi]=z;
             if (w>0.) rat[iqi]=range[iqi]/w;}
       if (fabs(z-1.)>1.e-6) return 0;
       for (i=1; i<=5; i++) value[i]=p[i6+i]/range[iq6-i];
       if (i1== -1) value[5]*=2.;
       iok=3-i1;
       for (i=1; i<=4; i++)
            {if (i<m)
                  {if (value[i]>value[i+1]) iok=0;}
             else {if (value[i]<value[i+1]) iok=0;}}
       return iok;
      }

void simulate()
      {
       int i, ip=1, prin, prin0;
       prin0=prin=N/10;
       for (i=1; i<=N; i++)
            {if (i==prin) {printf("\n%ld", ip++);
                           prin+=prin0;}
             y[i]=getrany();}
      }

double getrany()
```

```
      {
       int ii, k, j; double zz=0., z;
       for (ii=0; ii<ss; ii++)
            {k=numk[ii]; z=0.;
             for (j=0; j<k; j++) z+=randy(ii, j);
             zz+=(yrfac[ii]*log(1.-pow(ten, z)));}
             return l10(1.-exp(zz));
      }

double randy(ii, j)  int ii, j;
      {
       int i, it, ir, is; double u;
       it=JJ*ii+j; i=okay[it];
       if (i==5) return conp[it];
       if (i==6) return gauss(ii, j);
       u=ran1();
       if (i==1)
            return (u<pl[it] ? left[it]+lwidth[it]*sqrt(u/pl[it])
            : right[it]-rwidth[it]*sqrt((1.-u)/ql[it]));
       if (i==2)
            {if (u<q2k[it]) return ck[it]+ack[it]*sqrt(u/q2k[it]);
            if (u<fmk[it])
            return ak[it]+(sqrt(con2[it]+con3[it]*(u-q2k[it]))-
                   con1[it])/con3[it];
            if (u<=p3k[it])
            return bk[it]-(sqrt(con5[it]+con6[it]*(p3k[it]-u))-
                   con4[it])/con6[it];
            return dk[it]-dbk[it]*sqrt((1.-u)/q3k[it]);}
       is=ir=jj6*ii+6*j+1;
       while (u>un[ir]) ir++;
       return aaa[ir]+(is==ir && i==4 ? range[ir]*sqrt(u/un[ir]) :
              rat[ir]*(u-un[ir-1]));
      }

double l10(x)  double x;
      {
       return log(x)/lten;
      }

void sim2()
      {
       int i, j, kk=0, prin0, prin, ip=1, ktr[102];
       double a, b, c, d, x, c1, c2, c3, zn, cdf, pdf, xp, xx;
       FILE *out1, *out2;
       out1=fopen("scdf.c", "w"); out2=fopen("spdf.c", "w");
       printf("\nend of mini simulation");
       prin0=NIT/100; prin=prin0-1;
       a=y[1]; d=y[N];
       j=N/4; b=y[j];
       j*=3; c=y[j];
       c1=25./(b-a); c2=50./(c-b); c3=25./(d-c);
       for (j=0; j<102; j++) ktr[j]=0;
       for (i=0; i<NIT; i++)
            {if (i==prin)
                {printf("\n%ld", ip++); prin+=prin0;}
             x=getrany();
             j=(x<c ? (x<b ? (x<a ? 0 : c1*(x-a)+1.) :
               c2*(x-b)+26.) : (x<d ? c3*(x-c)+76. : 101));
```

```
            ktr[j]++;}
        zn=NIT;
        for (j=0; j<101; j++)
            {xp=x;
             kk+=ktr[j]; cdf=kk/zn;
             x=(j<=75 ? (j<=25 ? a+j/c1 : b+(j-25)/c2) :
               c+(j-75)/c3);
             fprintf(out1, "\n %10.5f %8.5f", x, cdf);
             if (j>0)
                 {pdf=(ktr[j]/zn)/(x-xp); xx=0.5*(x+xp);
                  fprintf(out2, "\n %10.5f %8.5f", xx, pdf);}}
        fclose(out1); fclose(out2);
       }

double gauss(ii, j)  int ii, j;
       {
        double z; int it;
        inow=1-inow;
        if (inow==1) ggrand();
        it=JJ*ii+j;
        z=lmean[it]+lsd[it]*(inow==1 ? gauss1 : gauss2);
        return (z<0. ? z : 0.);
       }

void ggrand()
       {
        double z, w;
        z=sqrt(-2.*log(ran1()));
        w=pi2*(ran1());
        gauss1=z*cos(w); gauss2=z*sin(w);
       }

void hpsort()
       {
        unsigned int i, ir, j, l, nm;
        double rra;
        nm=N;
        l=(nm >> 1)+1; ir=nm;
        for ( ; ; )
            {if (l>1) rra=y[--l];
             else {rra=y[ir]; y[ir]=y[1];
                   if (--ir == 1) {y[1]=rra; break;}}
             i=l; j=l+l;
             while (j<=ir) {if (j<ir && y[j]<y[j+1]) j++;
                            if (rra<y[j]) {y[i]=y[j]; i=j;
                                           j <<= 1;}
                            else j=ir+1;}
             y[i]=rra;}
       }

double ran1()
       {
        int j, k;
        static int iy=0, iv[NTAB];
        double temp;
        if (idum<=0 || !iy)
            {if (-idum<1) idum=1;
             else idum= -idum;
```

```
        for (j=NTAB+7; j>=0; j--)
             {k=idum/IQ; idum=IA*(idum-k*IQ)-IR*k;
              if (idum<0) idum+=IM;
              if (j<NTAB) iv[j]= idum;}
        iy=iv[0];}
 k=idum/IQ; idum=IA*(idum-k*IQ)-IR*k;
 if (idum<0) idum+=IM;
 j=iy/NDIV; iy=iv[j]; iv[j]= idum;
 if ((temp=AM*iy)>RNMX) return RNMX;
 else return temp;
}
```

The input file. To implement the simulation, the code requires a user-supplied input file, called "inrisky.c".

The following is an example "inrisky.c" that treats the single scenario case, n=1. The scenario probability P(R) is seen to be the product of six components, one for each type of prior. In fact, these are the very same priors that were given as examples earlier.

```
1
example
90.
6 0
2 1
-2.70
3 2
-3.30
4 3
3 3
-3.30
4
.1 .2
2 4
.181 .6 .181 .033 .005
2 5
-2.0 0.513
3 6
-3.30
```

The portion of the code which dictates the form of the input is "initialize()". The first entry in “inrisky.c” is ss, which is the number of scenarios of interest, denoted n earlier. [ss=n=1 in this example.] A segment of data then follows, beginning with scene, the name of the scenario. [scene=example here.] The next entry is yrfac, the number of times in a year the activity that gives rise to the scenario is expected to take place. [Setting yrfac = 1.0 produces a simulation of P(R) rather than $P_a(R)$.] The next line gives k, the number of components in P(R), i.e., the number of series events

in the scenario, and idef, the indicator (1 = "yes", 0 = "no") of whether the user wishes to use, for at least one of the k components, a lower limit in L5 different from the default value, -10. In the example, the scenario consists of k = 6 events, and, since idef = 0, the user will accept the default value of -10 throughout.

The next k blocks of data describe the k individual components. The order in which the blocks appear is irrelevant to the simulation.

Type 1 block (triangular on a single class). The first line gives the class number, bigm, and the value itype = 1 to indicate a type 1 prior. The next line gives the value of the mode, m, in $\log_{10}$ units. If bigm = 5 and idef = 1, the next line gives the user's preference, lt, for the left limit of L5.

Type 2 block (triangular over more than one class). The first line gives the modal class number, bigm, and the value itype = 2 to indicate a type 2 prior. The next line gives the value of the mode, m, in $\log_{10}$ units. The next line gives the leftmost class number, ll, and the rightmost class number, rr, where ll ≥ bigm ≥ rr. (Class numbers descend from left to right as probability increases.) If ll = 5 and idef = 1, the next line gives the user's preference, lt, for the left limit of L5.

Type 3 block (quasi-triangular). The first line gives the modal class number, bigm, and the value itype = 3 to indicate a type 3 prior. The next line gives the value of the mode, m, in $\log_{10}$ units. The next line gives the leftmost class number, ll, and the rightmost class number, rr, where ll ≥ bigm ≥ rr. If ll = 5 and idef = 1, the next line gives the user's preference, lt, for the left limit of L5. The next line gives the total probabilities to the left ($q_2 \geq 0$) and right ($q_3 \geq 0$) of the modal class. The modal class probability q_1 is not an input, since necessarily $q_1 = 1 - q_2 - q_3$.

Type 4 block (histogram). The first line gives the modal class number, bigm, and the value itype = 4 to indicate a type 4 prior. The next line gives the class probabilities, p_1, p_2, p_3, p_4, p_5. If $p_5 > 0$ and idef = 1, the next line gives the user's preference for the left limit of L5.

Type 5 block (lognormal). The first line gives the modal class number, bigm, and the value itype = 5 to indicate a type 5 prior. [In fact, any value for bigm may be inserted, since this quantity is not used in the simulation.] The next line gives the mean and standard deviation, in $\log_{10}$ units, of the corresponding normal distribution.

Type 6 block (known probability). The first line gives the modal class number, bigm, and the value itype = 6 to indicate a type 6 prior. [In fact, any value for bigm may be inserted, since this quantity is not used in the simulation.] The next line gives the value of the known probability, in $\log_{10}$ units.

It is important for the user to note that there are two constraints which must be satisfied for successful execution:

- The total number of scenarios (n in the discussion and "ss" in the code) must be no greater than 40.
- For each scenario the total number of components ("k" in the code) must be no greater than 10.

These are memory-driven constraints, which may be modified by changing NN and JJ in the #define section of the code, and recompiling.

The output files. Implementation of the code produces two output files, "scdf.c" and "spdf.c", which give values of the respective approximate CDF and pdf of $P_a(U_i R_i)$. To illustrate, consider the multi-scenario situation based on n=4 with the following input file, "inrisky.c".

```
4
P1
87.5
3 0
3 4
.008 .092 .8 .092 .008
1 5
-1.301 0.125
3 4
.008 .092 .8 .092 .008
P2
90.
6 0
2 1
-2.70
3 2
-3.30
4 3
3 3
-3.30
4 2
.1 .2
2 4
.181 .6 .181 .033 .005
2 5
-2.0 0.213
3 6
-3.30
```

```
P3
87.5
3 0
1 0
.8 .167 .028 .0045 .0005
1 5
-1.301 0.125
2 0
.095 .8 .095 .009 .001
P4
87.5
2 0
1 5
-1.301 0.125
4 3
-5.5
5 1
.2 .2
```

The following output files were obtained for this input. [Repeated executions will give slightly different results due to different random number seeds.]

"scdf.c". In each row there are two numbers, x F(x), where x is a value of the probability $P_a(U_i R_i)$ in $\log_{10}$ space, and F(x) is the corresponding value of the empirical simulated CDF, i.e., the approximate probability that $P_a(U_i R_i)$ is less than x.

```
-7.36864  0.00008
-7.22182  0.00014
-7.07500  0.00022
-6.92818  0.00034
-6.78135  0.00048
-6.63453  0.00073
-6.48771  0.00104
-6.34089  0.00147
-6.19406  0.00205
-6.04724  0.00274
-5.90042  0.00365
-5.75360  0.00476
-5.60678  0.00601
-5.45995  0.00757
-5.31313  0.00977
-5.16631  0.01336
-5.01949  0.01990
-4.87266  0.03086
-4.72584  0.04680
-4.57902  0.06797
-4.43220  0.09279
-4.28538  0.12025
-4.13855  0.14956
-3.99173  0.18138
-3.84491  0.21606
-3.69809  0.25455
```

```
-3.66858  0.26262
-3.63907  0.27108
-3.60956  0.27958
-3.58006  0.28820
-3.55055  0.29697
-3.52104  0.30590
-3.49153  0.31484
-3.46202  0.32404
-3.43252  0.33351
-3.40301  0.34318
-3.37350  0.35268
-3.34399  0.36256
-3.31448  0.37235
-3.28498  0.38217
-3.25547  0.39210
-3.22596  0.40210
-3.19645  0.41203
-3.16694  0.42216
-3.13744  0.43225
-3.10793  0.44234
-3.07842  0.45251
-3.04891  0.46280
-3.01940  0.47300
-2.98990  0.48331
-2.96039  0.49352
-2.93088  0.50366
-2.90137  0.51398
-2.87187  0.52427
-2.84236  0.53463
-2.81285  0.54492
-2.78334  0.55542
-2.75383  0.56592
-2.72433  0.57650
-2.69482  0.58694
-2.66531  0.59754
-2.63580  0.60813
-2.60629  0.61875
-2.57679  0.62921
-2.54728  0.63976
-2.51777  0.65049
-2.48826  0.66107
-2.45875  0.67159
-2.42925  0.68195
-2.39974  0.69247
-2.37023  0.70284
-2.34072  0.71299
-2.31121  0.72302
-2.28171  0.73318
-2.25220  0.74314
-2.22269  0.75270
-2.13382  0.78078
-2.04495  0.80658
-1.95607  0.83025
-1.86720  0.85162
-1.77833  0.87073
-1.68946  0.88733
-1.60058  0.90161
-1.51171  0.91359
```

```
-1.42284  0.92328
-1.33397  0.93129
-1.24509  0.93816
-1.15622  0.94414
-1.06735  0.94989
-0.97847  0.95513
-0.88960  0.96018
-0.80073  0.96498
-0.71186  0.96961
-0.62298  0.97403
-0.53411  0.97819
-0.44524  0.98217
-0.35637  0.98601
-0.26749  0.98964
-0.17862  0.99311
-0.08975  0.99656
-0.00088  0.99998
```

"spdf.c". In each row there are two numbers, x f(x), where x is a value of the probability $P_a(U_i R_i)$ in $\log_{10}$ space, and f(x) is the corresponding value of the simulated pdf, i.e., an approximation of f(x)=F′(x).

```
-7.29523  0.00036
-7.14841  0.00057
-7.00159  0.00083
-6.85476  0.00094
-6.70794  0.00168
-6.56112  0.00212
-6.41430  0.00296
-6.26748  0.00394
-6.12065  0.00468
-5.97383  0.00622
-5.82701  0.00754
-5.68019  0.00856
-5.53336  0.01057
-5.38654  0.01501
-5.23972  0.02444
-5.09290  0.04455
-4.94608  0.07468
-4.79925  0.10853
-4.65243  0.14422
-4.50561  0.16899
-4.35879  0.18703
-4.21196  0.19967
-4.06514  0.21670
-3.91832  0.23619
-3.77150  0.26219
-3.68333  0.27332
-3.65383  0.28674
-3.62432  0.28809
-3.59481  0.29223
-3.56530  0.29707
-3.53579  0.30263
-3.50629  0.30294
-3.47678  0.31195
-3.44727  0.32073
```

```
-3.41776  0.32771
-3.38825  0.32225
-3.35875  0.33455
-3.32924  0.33194
-3.29973  0.33269
-3.27022  0.33659
-3.24071  0.33872
-3.21121  0.33659
-3.18170  0.34326
-3.15219  0.34208
-3.12268  0.34191
-3.09317  0.34475
-3.06367  0.34862
-3.03416  0.34570
-3.00465  0.34930
-2.97514  0.34614
-2.94563  0.34364
-2.91613  0.34974
-2.88662  0.34862
-2.85711  0.35102
-2.82760  0.34892
-2.79810  0.35567
-2.76859  0.35584
-2.73908  0.35865
-2.70957  0.35374
-2.68006  0.35916
-2.65056  0.35909
-2.62105  0.35980
-2.59154  0.35445
-2.56203  0.35770
-2.53252  0.36360
-2.50302  0.35848
-2.47351  0.35638
-2.44400  0.35126
-2.41449  0.35638
-2.38498  0.35146
-2.35548  0.34408
-2.32597  0.33987
-2.29646  0.34425
-2.26695  0.33771
-2.23744  0.32398
-2.17825  0.31592
-2.08938  0.29026
-2.00051  0.26640
-1.91164  0.24040
-1.82276  0.21499
-1.73389  0.18684
-1.64502  0.16069
-1.55615  0.13475
-1.46727  0.10907
-1.37840  0.09013
-1.28953  0.07734
-1.20066  0.06728
-1.11178  0.06470
-1.02291  0.05895
-0.93404  0.05677
-0.84517  0.05404
-0.75629  0.05206
```

```
-0.66742  0.04972
-0.57855  0.04691
-0.48968  0.04478
-0.40080  0.04313
-0.31193  0.04090
-0.22306  0.03902
-0.13419  0.03883
-0.04531  0.03845
```

The approximate CDF and pdf are easily graphed by using software such as Excel. The shakiness is due to finiteness of the number of replications in the simulation (namely one million).

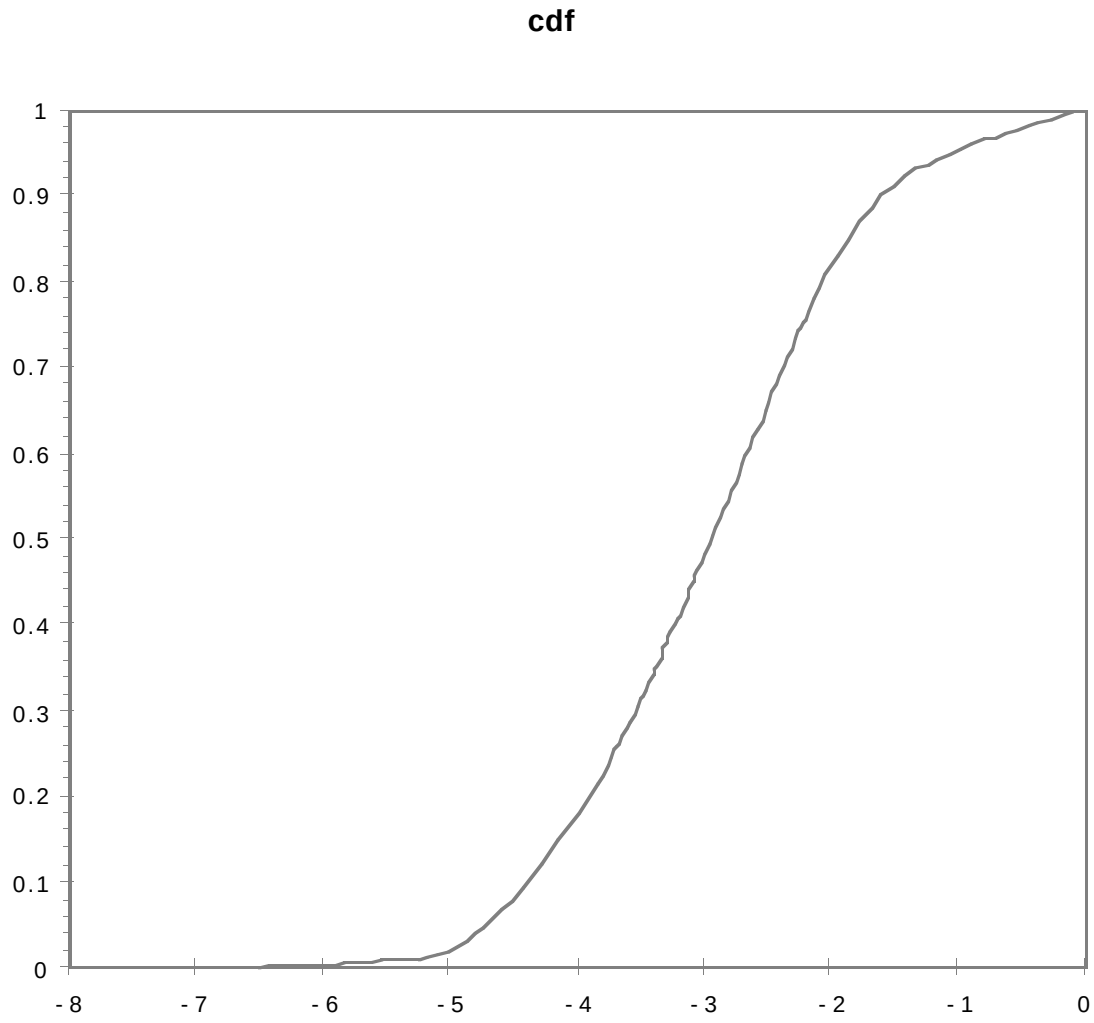


Figure 6. Excel graph of the output file, "scdf.c".

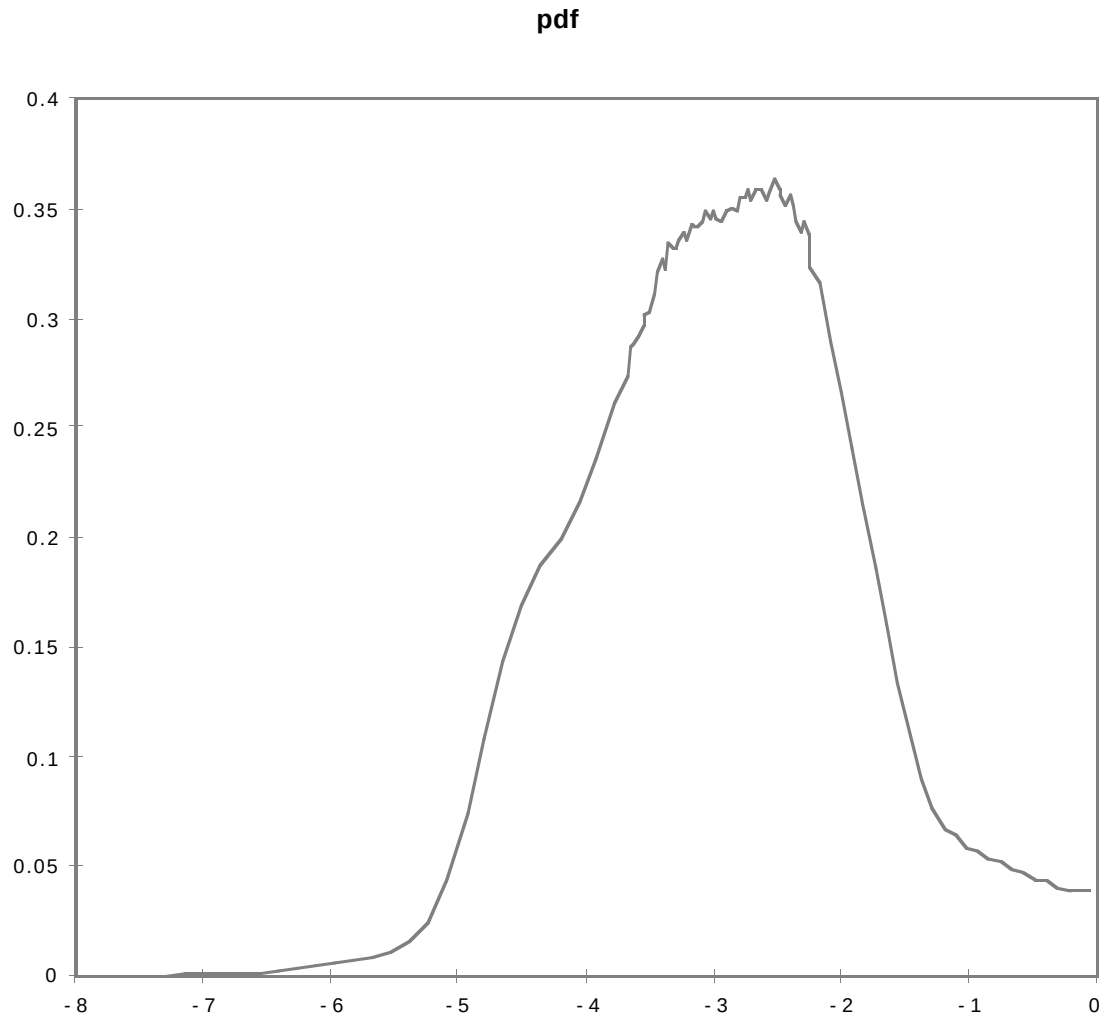


Figure 7. Excel graph of the output file, "spdf.c".

Technical Information Department • **Lawrence Livermore National Laboratory**
University of California • Livermore, California 94551