Wetenschappelijk Programmeren

Tijdens de hoorcolleges 1 t/m 6 hebben wij de basis van de taal C geleert.

Vandaag en volgende week willen wij dit toepassen en enkele wetenschappelijke problemen oplossen.



Programmeren 1
Programming 1

2014/2015

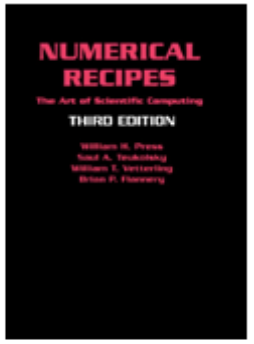
Online Help:
The GNU C reference manual
The C Library Reference Guide
UNIX Tutorial for Beginners
GNU plot

NP033B
1e jaar, kwartaal II, 3 ec

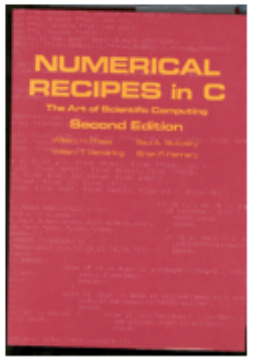
Literatuur:

B.W. Kernighan, D.M. Ritchie
The C Programming Language
Prentice Hall Software

W.H. Press et al.
Numerical Recipes 3rd edition
Cambridge University Press



W.H. Press et al.
Numerical Recipes in C 2nd edition
Cambridge University Press

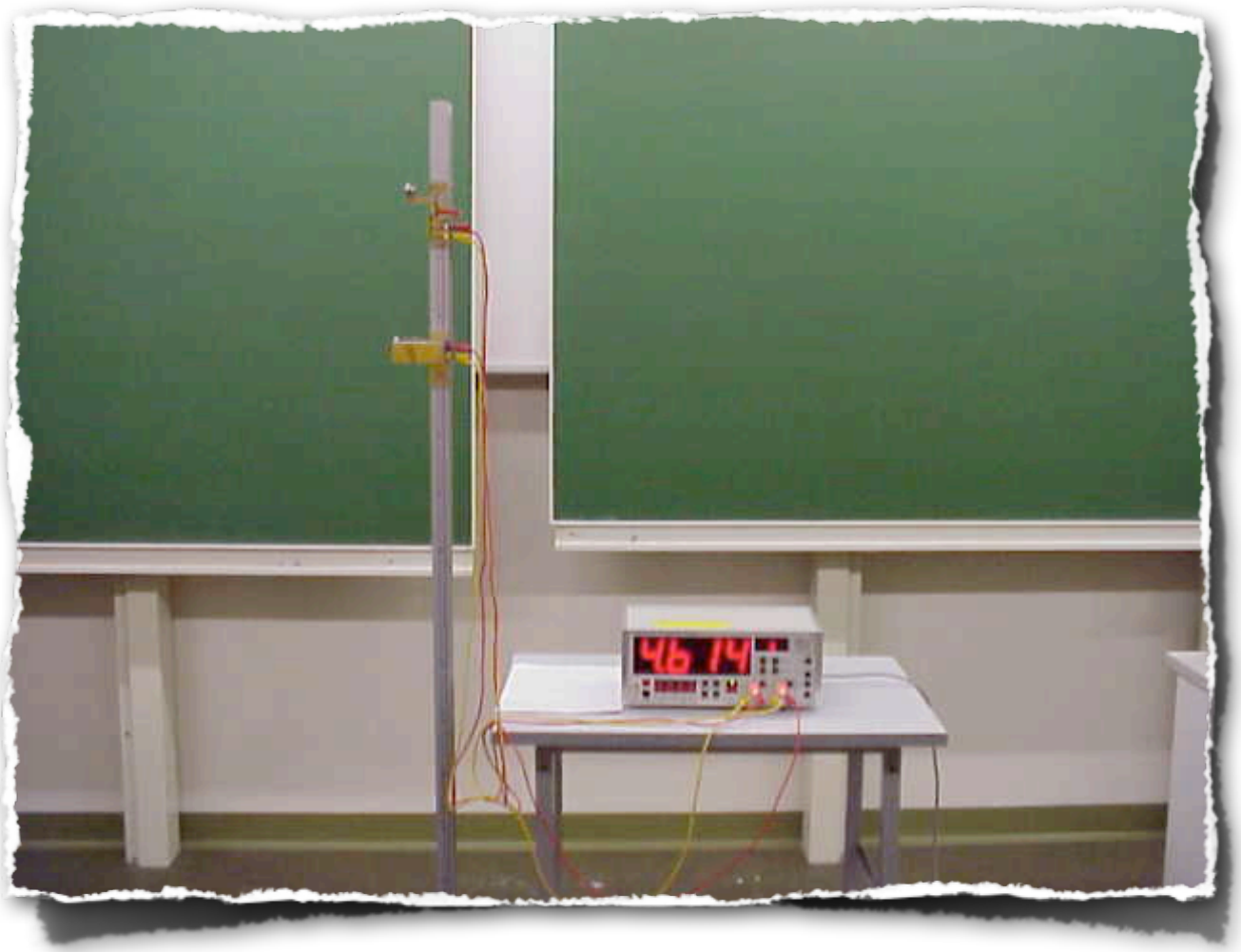


http://particle.astro.ru.nl/goto.html?prog1415

straight line fit

In het natuurkunde practicum meten wij de zwaartekracht parameter *g*.

$$s = \frac{1}{2} \cdot g \cdot t^2$$



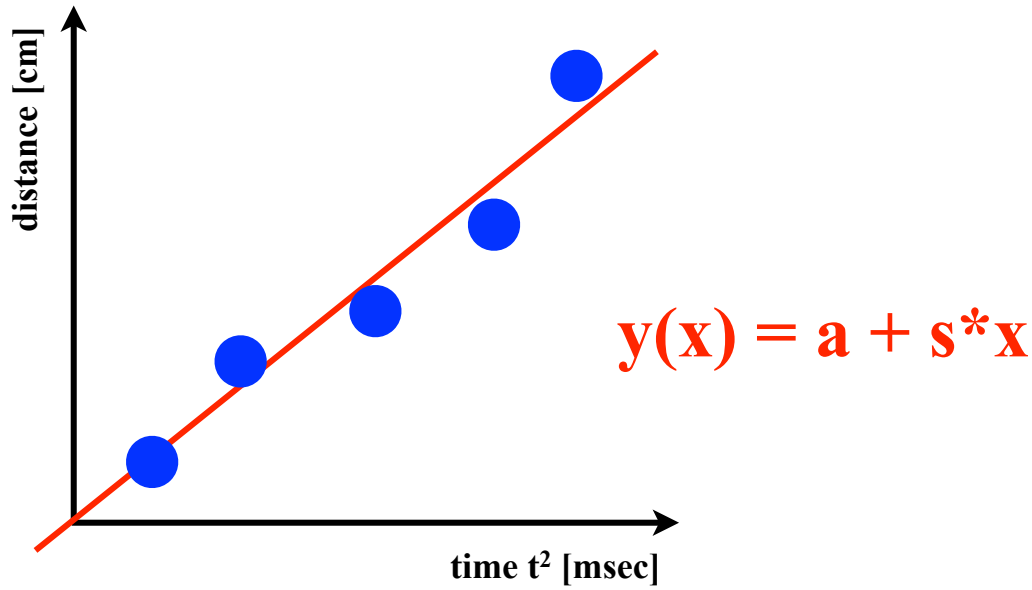
straight line fit

In het natuurkunde practicum meten wij de zwaartekracht parameter *g*.

$$s = \frac{1}{2} \cdot g \cdot t^2$$

Wij meten de waarden:

s [cm]	t^2 [msec^2]
10	22
20	38
50	103
100	200
150	310



Wij berekenen een straight line fit en de stijging *s* van de rechte lijn geeft de parameter *g = 2 * s*.

A concrete example will make the considerations of the previous section more meaningful. We consider the problem of fitting a set of N data points (x_i, y_i) to a straight-line model

$$y(x) = y(x; a, b) = a + bx \tag{15.2.1}$$

This problem is often called *linear regression*, a terminology that originated, long ago, in the social sciences. We assume that the uncertainty σ_i associated with each measurement y_i is known, and that the x_i's (values of the dependent variable) are known exactly.

To measure how well the model agrees with the data, we use the chi-square merit function (15.1.5), which in this case is

$$\chi^2(a, b) = \sum_{i=1}^{N} \left(\frac{y_i - a - bx_i}{\sigma_i} \right)^2 \tag{15.2.2}$$

If the measurement errors are normally distributed, then this merit function will give maximum likelihood parameter estimations of a and b; if the errors are not normally distributed, then the estimations are not maximum likelihood, but may still be useful in a practical sense. In §15.7, we will treat the case where outlier points are so numerous as to render the χ^2 merit function useless.

Equation (15.2.2) is minimized to determine a and b. At its minimum, derivatives of $\chi^2(a, b)$ with respect to a, b vanish.

$$\begin{aligned} 0 &= \frac{\partial \chi^2}{\partial a} = -2 \sum_{i=1}^{N} \frac{y_i - a - bx_i}{\sigma_i^2} \\ 0 &= \frac{\partial \chi^2}{\partial b} = -2 \sum_{i=1}^{N} \frac{x_i (y_i - a - bx_i)}{\sigma_i^2} \end{aligned} \tag{15.2.3}$$

These conditions can be rewritten in a convenient form if we define the following sums:

$$\begin{aligned} S &\equiv \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \quad S_x \equiv \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} \quad S_y \equiv \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} \\ S_{xx} &\equiv \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2} \quad S_{xy} \equiv \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2} \end{aligned} \tag{15.2.4}$$

With these definitions (15.2.3) becomes

$$\begin{aligned} aS + bS_x &= S_y \\ aS_x + bS_{xx} &= S_{xy} \end{aligned} \tag{15.2.5}$$

The solution of these two equations in two unknowns is calculated as

$$\begin{aligned} \Delta &\equiv SS_{xx} - (S_x)^2 \\ a &= \frac{S_{xx} S_y - S_x S_{xy}}{\Delta} \\ b &= \frac{SS_{xy} - S_x S_y}{\Delta} \end{aligned} \tag{15.2.6}$$

Equation (15.2.6) gives the solution for the best-fit model parameters a and b.

We are not done, however. We must estimate the probable uncertainties in the estimates of a and b, since obviously the measurement errors in the data must introduce some uncertainty in the determination of those parameters. If the data are independent, then each contributes its own bit of uncertainty to the parameters. Consideration of propagation of errors shows that the variance σ_f^2 in the value of any function will be

$$\sigma_f^2 = \sum_{i=1}^{N} \sigma_i^2 \left(\frac{\partial f}{\partial y_i} \right)^2 \tag{15.2.7}$$

For the straight line, the derivatives of a and b with respect to y_i can be directly evaluated from the solution:

$$\begin{aligned} \frac{\partial a}{\partial y_i} &= \frac{S_{xx} - S_x x_i}{\sigma_i^2 \Delta} \\ \frac{\partial b}{\partial y_i} &= \frac{S x_i - S_x}{\sigma_i^2 \Delta} \end{aligned} \tag{15.2.8}$$

Summing over the points as in (15.2.7), we get

$$\begin{aligned} \sigma_a^2 &= S_{xx}/\Delta \\ \sigma_b^2 &= S/\Delta \end{aligned} \tag{15.2.9}$$

which are the variances in the estimates of a and b, respectively. We will see in §15.6 that an additional number is also needed to characterize properly the probable uncertainty of the parameter estimation. That number is the *covariance* of a and b, and (as we will see below) is given by

$$\mathrm{Cov}(a, b) = -S_x/\Delta \tag{15.2.10}$$

The coefficient of correlation between the uncertainty in a and the uncertainty in b, which is a number between -1 and 1, follows from (15.2.10) (compare equation 14.5.1),

$$r_{ab} = \frac{-S_x}{\sqrt{SS_{xx}}} \tag{15.2.11}$$

A positive value of r_{ab} indicates that the errors in a and b are likely to have the same sign, while a negative value indicates the errors are anticorrelated, likely to have opposite signs.

We are *still* not done. We must estimate the goodness-of-fit of the data to the model. Absent this estimate, we have not the slightest indication that the parameters a and b in the model have any meaning at all! The probability Q that a value of chi-square as *poor* as the value (15.2.2) should occur by chance is

$$Q = \texttt{gammq}\left(\frac{N-2}{2}, \frac{\chi^2}{2} \right) \tag{15.2.12}$$

Here `gammq` is our routine for the incomplete gamma function $Q(a, x)$, §6.2. If Q is larger than, say, 0.1, then the goodness-of-fit is believable. If it is larger than, say, 0.001, then the fit *may* be acceptable if the errors are nonnormal or have been moderately underestimated. If Q is less than 0.001 then the model and/or estimation procedure can rightly be called into question. In this latter case, turn to §15.7 to proceed further.

If you do not know the individual measurement errors of the points σ_i, and are proceeding (dangerously) to use equation (15.1.6) for estimating these errors, then here is the procedure for estimating the probable uncertainties of the parameters a and b: Set $\sigma_i \equiv 1$ in all equations through (15.2.6), and multiply σ_a and σ_b, as obtained from equation (15.2.9), by the additional factor $\sqrt{\chi^2/(N-2)}$, where χ^2 is computed by (15.2.2) using the fitted parameters a and b. As discussed above, this procedure is equivalent to *assuming* a good fit, so you get no independent goodness-of-fit probability Q.

In §14.5 we promised a relation between the linear correlation coefficient r (equation 14.5.1) and a goodness-of-fit measure, χ^2 (equation 15.2.2). For unweighted data (all $\sigma_i = 1$), that relation is

$$\chi^2 = (1 - r^2)\text{NVar}\,(y_1 \ldots y_N) \tag{15.2.13}$$

where

$$\text{NVar}\,(y_1 \ldots y_N) \equiv \sum_{i=1}^{N} (y_i - \overline{y})^2 \tag{15.2.14}$$

For data with varying weights σ_i, the above equations remain valid if the sums in equation (14.5.1) are weighted by $1/\sigma_i^2$.

The following function, `fit`, carries out exactly the operations that we have discussed. When the weights σ are known in advance, the calculations exactly correspond to the formulas above. However, when weights σ are unavailable, the routine *assumes* equal values of σ for each point and *assumes* a good fit, as discussed in §15.1.

The formulas (15.2.6) are susceptible to roundoff error. Accordingly, we rewrite them as follows: Define

$$t_i = \frac{1}{\sigma_i}\left(x_i - \frac{S_x}{S}\right), \qquad i = 1, 2, \ldots, N \tag{15.2.15}$$

and

$$S_{tt} = \sum_{i=1}^{N} t_i^2 \tag{15.2.16}$$

Then, as you can verify by direct substitution,

$$b = \frac{1}{S_{tt}} \sum_{i=1}^{N} \frac{t_i y_i}{\sigma_i} \tag{15.2.17}$$

$$a = \frac{S_y - S_x b}{S} \tag{15.2.18}$$

$$\sigma_a^2 = \frac{1}{S}\left(1 + \frac{S_x^2}{S S_{tt}}\right) \tag{15.2.19}$$

$$\sigma_b^2 = \frac{1}{S_{tt}} \tag{15.2.20}$$

$$\text{Cov}(a, b) = -\frac{S_x}{S S_{tt}} \tag{15.2.21}$$

$$r_{ab} = \frac{\text{Cov}(a, b)}{\sigma_a \sigma_b} \tag{15.2.22}$$

straight line fit

```
#include <math.h>
#define NRANSI
#include "nrutil.h"

void fit(float x[], float y[], int ndata, float sig[], int mwt, float *a,
	float *b, float *siga, float *sigb, float *chi2, float *q)
{
	float gammq(float a, float x);
	int i;
	float wt,t,sxoss,sx=0.0,sy=0.0,st2=0.0,ss,sigdat;

	*b=0.0;
	if (mwt) {
		ss=0.0;
		for (i=1;i<=ndata;i++) {
			wt=1.0/SQR(sig[i]);
			ss += wt;
			sx += x[i]*wt;
			sy += y[i]*wt;
		}
	} else {
		for (i=1;i<=ndata;i++) {
			sx += x[i];
			sy += y[i];
		}
		ss=ndata;
	}
	sxoss=sx/ss;
	if (mwt) {
		for (i=1;i<=ndata;i++) {
			t=(x[i]-sxoss)/sig[i];
			st2 += t*t;
			*b += t*y[i]/sig[i];
		}
	} else {
		for (i=1;i<=ndata;i++) {
			t=x[i]-sxoss;
			st2 += t*t;
			*b += t*y[i];
		}
	}
	*b /= st2;
	*a=(sy-sx*(*b))/ss;
	*siga=sqrt((1.0+sx*sx/(ss*st2))/ss);
	*sigb=sqrt(1.0/st2);
	*chi2=0.0;
	if (mwt == 0) {
		for (i=1;i<=ndata;i++)
			*chi2 += SQR(y[i]-(*a)-(*b)*x[i]);
		*q=1.0;
		sigdat=sqrt((*chi2)/(ndata-2));
		*siga *= sigdat;
		*sigb *= sigdat;
	} else {
		for (i=1;i<=ndata;i++)
			*chi2 += SQR((y[i]-(*a)-(*b)*x[i])/sig[i]);
		*q=gammq(0.5*(ndata-2),0.5*(*chi2));
	}
}
#undef NRANSI
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

```
void fit(float x[], float y[], int ndata, float sig[], int mwt, float *a,
    float *b, float *siga, float *sigb, float *chi2, float *q)
```

Given a set of data points `x[1..ndata]`,`y[1..ndata]` with individual standard deviations `sig[1..ndata]`, fit them to a straight line $y = a + bx$ by minimizing χ^2. Returned are `a`,`b` and their respective probable uncertainties `siga` and `sigb`, the chi-square `chi2`, and the goodness-of-fit probability `q` (that the fit would have χ^2 this large or larger). If `mwt`=0 on input, then the standard deviations are assumed to be unavailable: `q` is returned as `1.0` and the normalization of `chi2` is to unit standard deviation on all points.

fit.c

straight line fit

```
#include <stdio.h>
#include "gammln.c"
#include "gser.c"
#include "nrerror.c"
#include "gcf.c"
#include "gammq.c"
#include "fit.c"

#define NDATA 5

int main()
 {
  int n=0;
  float y[NDATA];   // distance in meter
  float x[NDATA];   // time^2 in msec^2
  float sig[NDATA];
  float a, b, siga, sigb, chi2, q;
  char text[80];

  FILE* fp;

  fp=fopen("newton.dat","r");
   while (fgets(text,80,fp) != NULL)
    {
     sscanf(text,"%f %f",&y[n], &x[n]);
     printf("x=%f msec^2 y=%f m\n",x[n],y[n]);
     n++;
    }
  fclose(fp);

  fit(x,y, NDATA, sig, 0, &a, &b, &siga, &sigb, &chi2, &q);
  printf("a=%f b=%f\nparameter g = 2*b = %f +/- %f m/s^2\n",a,b, b*2, sigb*2);

 }
```

newton.c

gamma function

6.1 Gamma Function, Beta Function, Factorials, Binomial Coefficients

The gamma function is defined by the integral

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt \tag{6.1.1}$$

When the argument z is an integer, the gamma function is just the familiar factorial function, but offset by one,

$$n! = \Gamma(n+1) \tag{6.1.2}$$

The gamma function satisfies the recurrence relation

$$\Gamma(z+1) = z\Gamma(z) \tag{6.1.3}$$

If the function is known for arguments $z > 1$ or, more generally, in the half complex plane $\mathrm{Re}(z) > 1$ it can be obtained for $z < 1$ or $\mathrm{Re}\,(z) < 1$ by the reflection formula

$$\Gamma(1-z) = \frac{\pi}{\Gamma(z)\sin(\pi z)} = \frac{\pi z}{\Gamma(1+z)\sin(\pi z)} \tag{6.1.4}$$

Notice that $\Gamma(z)$ has a pole at $z = 0$, and at all negative integer values of z.

There are a variety of methods in use for calculating the function $\Gamma(z)$ numerically, but none is quite as neat as the approximation derived by Lanczos [1]. This scheme is entirely specific to the gamma function, seemingly plucked from thin air. We will not attempt to derive the approximation, but only state the resulting formula: For certain integer choices of γ and N, and for certain coefficients $c_1, c_2, \ldots, c_N$, the gamma function is given by

$$\begin{aligned}\Gamma(z+1) &= (z+\gamma+\tfrac{1}{2})^{z+\frac{1}{2}} e^{-(z+\gamma+\frac{1}{2})} \\ &\quad \times \sqrt{2\pi}\left[c_0 + \frac{c_1}{z+1} + \frac{c_2}{z+2} + \cdots + \frac{c_N}{z+N} + \epsilon\right] \quad (z > 0)\end{aligned} \tag{6.1.5}$$

You can see that this is a sort of take-off on Stirling's approximation, but with a series of corrections that take into account the first few poles in the left complex plane. The constant c_0 is very nearly equal to 1. The error term is parametrized by ϵ. For $\gamma = 5$, $N = 6$, and a certain set of c's, the error is smaller than $|\epsilon| < 2 \times 10^{-10}$. Impressed? If not, then perhaps you will be impressed by the fact that (with these same parameters) the formula (6.1.5) and bound on ϵ apply for the *complex* gamma function, *everywhere in the half complex plane Re* $z > 0$.

It is better to implement $\ln\Gamma(x)$ than $\Gamma(x)$, since the latter will overflow many computers' floating-point representation at quite modest values of x. Often the gamma function is used in calculations where the large values of $\Gamma(x)$ are divided by other large numbers, with the result being a perfectly ordinary value. Such operations would normally be coded as subtraction of logarithms. With (6.1.5) in hand, we can compute the logarithm of the gamma function with two calls to a logarithm and 25 or so arithmetic operations. This makes it not much more difficult than other built-in functions that we take for granted, such as $\sin x$ or e^x:

gamma function

```
#include <math.h>

float gammln(float xx)
{
        double x,y,tmp,ser;
        static double cof[6]={76.18009172947146,-86.50532032941677,
                24.01409824083091,-1.231739572450155,
                0.1208650973866179e-2,-0.5395239384953e-5};
        int j;

        y=x=xx;
        tmp=x+5.5;
        tmp -= (x+0.5)*log(tmp);
        ser=1.000000000190015;
        for (j=0;j<=5;j++) ser += cof[j]/++y;
        return -tmp+log(2.5066282746310005*ser/x);
}
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

`float gammln(float xx)`
Returns the value $\ln[\Gamma(\texttt{xx})]$ for `xx` > 0.

gammln.c

6.2 Incomplete Gamma Function, Error Function, Chi-Square Probability Function, Cumulative Poisson Function

The incomplete gamma function is defined by

$$P(a,x) \equiv \frac{\gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt \qquad (a > 0) \qquad (6.2.1)$$

It has the limiting values

$$P(a,0) = 0 \qquad \text{and} \qquad P(a,\infty) = 1 \qquad (6.2.2)$$

The incomplete gamma function $P(a,x)$ is monotonic and (for a greater than one or so) rises from "near-zero" to "near-unity" in a range of x centered on about $a-1$, and of width about $\sqrt{a}$ (see Figure 6.2.1).

The complement of $P(a,x)$ is also confusingly called an incomplete gamma function,

$$Q(a,x) \equiv 1 - P(a,x) \equiv \frac{\Gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)} \int_x^\infty e^{-t} t^{a-1} dt \qquad (a > 0) \qquad (6.2.3)$$

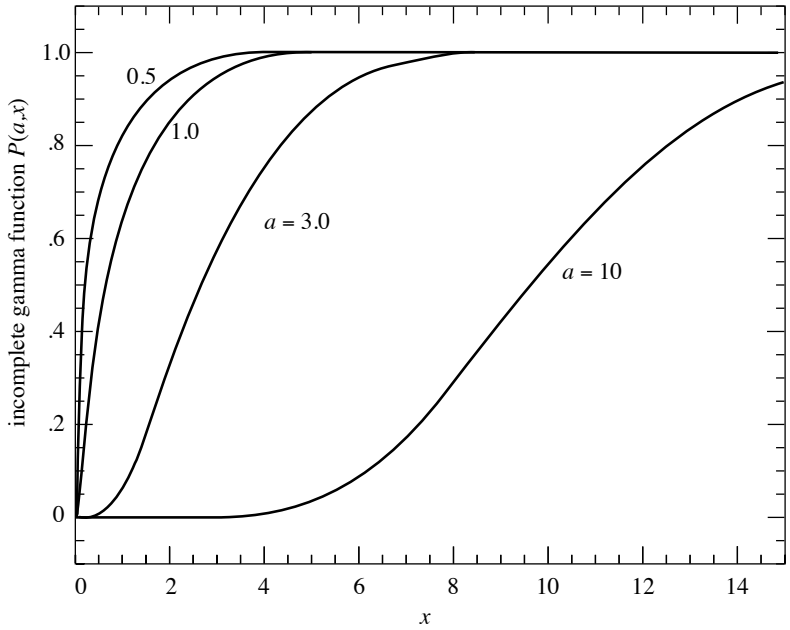


Figure 6.2.1. The incomplete gamma function $P(a,x)$ for four values of a.

It has the limiting values

$$Q(a,0) = 1 \qquad \text{and} \qquad Q(a,\infty) = 0 \qquad (6.2.4)$$

The notations $P(a,x), \gamma(a,x)$, and $\Gamma(a,x)$ are standard; the notation $Q(a,x)$ is specific to this book.

There is a series development for $\gamma(a,x)$ as follows:

$$\gamma(a,x) = e^{-x} x^a \sum_{n=0}^{\infty} \frac{\Gamma(a)}{\Gamma(a+1+n)} x^n \qquad (6.2.5)$$

One does not actually need to compute a new $\Gamma(a+1+n)$ for each n; one rather uses equation (6.1.3) and the previous coefficient.

A continued fraction development for $\Gamma(a,x)$ is

$$\Gamma(a,x) = e^{-x} x^a \left(\frac{1}{x+} \frac{1-a}{1+} \frac{1}{x+} \frac{2-a}{1+} \frac{2}{x+} \cdots \right) \qquad (x > 0) \qquad (6.2.6)$$

It is computationally better to use the even part of (6.2.6), which converges twice as fast (see §5.2):

$$\Gamma(a,x) = e^{-x} x^a \left(\frac{1}{x+1-a-} \frac{1\cdot(1-a)}{x+3-a-} \frac{2\cdot(2-a)}{x+5-a-} \cdots \right) \qquad (x > 0) \qquad (6.2.7)$$

gamma function

```
float gammq(float a, float x)
{
        void gcf(float *gammcf, float a, float x, float *gln);
        void gser(float *gamser, float a, float x, float *gln);
        void nrerror(char error_text[]);
        float gamser,gammcf,gln;

        if (x < 0.0 || a <= 0.0) nrerror("Invalid arguments in routine gammq");
        if (x < (a+1.0)) {
                gser(&gamser,a,x,&gln);
                return 1.0-gamser;
        } else {
                gcf(&gammcf,a,x,&gln);
                return gammcf;
        }
}
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

`float gammq(float a, float x)`
Returns the incomplete gamma function $Q(a, x) \equiv 1 - P(a, x)$.

gammq.c

gamma function

```
#include <math.h>
#define ITMAX 100
#define EPS 3.0e-7

void gser(float *gamser, float a, float x, float *gln)
{
        float gammln(float xx);
        void nrerror(char error_text[]);
        int n;
        float sum,del,ap;

        *gln=gammln(a);
        if (x <= 0.0) {
                if (x < 0.0) nrerror("x less than 0 in routine gser");
                *gamser=0.0;
                return;
        } else {
                ap=a;
                del=sum=1.0/a;
                for (n=1;n<=ITMAX;n++) {
                        ++ap;
                        del *= x/ap;
                        sum += del;
                        if (fabs(del) < fabs(sum)*EPS) {
                                *gamser=sum*exp(-x+a*log(x)-(*gln));
                                return;
                        }
                }
                nrerror("a too large, ITMAX too small in routine gser");
                return;
        }
}
#undef ITMAX
#undef EPS
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

gser.c

`void gser(float *gamser, float a, float x, float *gln)`
Returns the incomplete gamma function $P(a, x)$ evaluated by its series representation as `gamser`.
Also returns $\ln \Gamma(a)$ as `gln`.

gamma function

```
#include <math.h>
#define ITMAX 100
#define EPS 3.0e-7
#define FPMIN 1.0e-30

void gcf(float *gammcf, float a, float x, float *gln)
{
        float gammln(float xx);
        void nrerror(char error_text[]);
        int i;
        float an,b,c,d,del,h;

        *gln=gammln(a);
        b=x+1.0-a;
        c=1.0/FPMIN;
        d=1.0/b;
        h=d;
        for (i=1;i<=ITMAX;i++) {
                an = -i*(i-a);
                b += 2.0;
                d=an*d+b;
                if (fabs(d) < FPMIN) d=FPMIN;
                c=b+an/c;
                if (fabs(c) < FPMIN) c=FPMIN;
                d=1.0/d;
                del=d*c;
                h *= del;
                if (fabs(del-1.0) < EPS) break;
        }
        if (i > ITMAX) nrerror("a too large, ITMAX too small in gcf");
        *gammcf=exp(-x+a*log(x)-(*gln))*h;
}
#undef ITMAX
#undef EPS
#undef FPMIN
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

`void gcf(float *gammcf, float a, float x, float *gln)`
Returns the incomplete gamma function $Q(a, x)$ evaluated by its continued fraction representation as `gammcf`. Also returns $\ln \Gamma(a)$ as `gln`.

gcf.c

factorial function

How shall we write a routine for the factorial function $n!$? Generally the factorial function will be called for small integer values (for large values it will overflow anyway!), and in most applications the same integer value will be called for many times. It is a profligate waste of computer time to call `exp(gammln(n+1.0))` for each required factorial. Better to go back to basics, holding `gammln` in reserve for unlikely calls:

```
#include <math.h>

float factrl(int n)
{
        float gammln(float xx);
        void nrerror(char error_text[]);
        static int ntop=4;
        static float a[33]={1.0,1.0,2.0,6.0,24.0};
        int j;

        if (n < 0) nrerror("Negative factorial in routine factrl");
        if (n > 32) return exp(gammln(n+1.0));
        while (ntop<n) {
                j=ntop++;
                a[ntop]=a[j]*ntop;
        }
        return a[n];
}
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

A useful point is that `factrl` will be *exact* for the smaller values of `n`, since floating-point multiplies on small integers are exact on all computers. This exactness will not hold if we turn to the logarithm of the factorials. For binomial coefficients, however, we must do exactly this, since the individual factorials in a binomial coefficient will overflow long before the coefficient itself will.

`float factrl(int n)`
Returns the value `n`! as a floating-point number.

factrl.c

```
#include <stdio.h>

#include "nrutil.c"
#include "gammln.c"
#include "factrl.c"

int main()
 {
  int n;

  for(n=0; n<=34; n++)
   printf("%5i! = %50f \n",n,factrl(n));
 }
```

factorial.c

random numbers

Zijn de random numbers van rand() echt toevallig?

```
#include <stdio.h>

#define MAX 10

int main()
 {
  int i;

  // set seed value for random number generator
  srand(10);

  for(i=0; i<MAX; i++)
   printf("%20f \n", (double) rand() );
 }
```

random.c

The following routine, ran1, uses the Minimal Standard for its random value, but it shuffles the output to remove low-order serial correlations. A random deviate derived from the jth value in the sequence, I_j, is output not on the jth call, but rather on a randomized later call, $j + 32$ on average. The shuffling algorithm is due to Bays and Durham as described in Knuth [4], and is illustrated in Figure 7.1.1.

```
#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)

float ran1(long *idum)
"Minimal" random number generator of Park and Miller with Bays-Durham shuffle and added
safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint
values). Call with idum a negative integer to initialize; thereafter, do not alter idum between
successive deviates in a sequence. RNMX should approximate the largest floating value that is
less than 1.
{
    int j;
    long k;
    static long iy=0;
    static long iv[NTAB];
    float temp;

    if (*idum <= 0 || !iy) {                Initialize.
        if (-(*idum) < 1) *idum=1;          Be sure to prevent idum = 0.
        else *idum = -(*idum);
        for (j=NTAB+7;j>=0;j--) {           Load the shuffle table (after 8 warm-ups).
            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;                           Start here when not initializing.
    *idum=IA*(*idum-k*IQ)-IR*k;             Compute idum=(IA*idum) % IM without over-
    if (*idum < 0) *idum += IM;                 flows by Schrage's method.
    j=iy/NDIV;                              Will be in the range 0..NTAB-1.
    iy=iv[j];                               Output previously stored value and refill the
    iv[j] = *idum;                              shuffle table.
    if ((temp=AM*iy) > RNMX) return RNMX;   Because users don't expect endpoint values.
    else return temp;
}
```

The routine ran1 passes those statistical tests that ran0 is known to fail. In fact, we do not know of any statistical test that ran1 fails to pass, except when the number of calls starts to become on the order of the period m, say $> 10^8 \approx m/20$.

For situations when even longer random sequences are needed, L'Ecuyer [6] has given a good way of combining two different sequences with different periods so as to obtain a new sequence whose period is the least common multiple of the two periods. The basic idea is simply to add the two sequences, modulo the modulus of

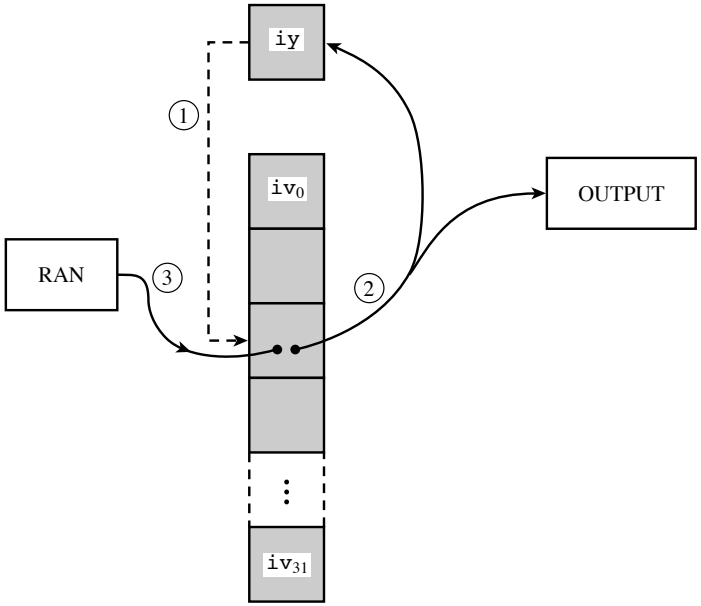


Figure 7.1.1. Shuffling procedure used in ran1 to break up sequential correlations in the Minimal Standard generator. Circled numbers indicate the sequence of events: On each call, the random number in iy is used to choose a random element in the array iv. That element becomes the output random number, and also is the next iy. Its spot in iv is refilled from the Minimal Standard routine.

either of them (call it m). A trick to avoid an intermediate value that overflows the integer wordsize is to subtract rather than add, and then add back the constant $m - 1$ if the result is ≤ 0, so as to wrap around into the desired interval $0, \ldots, m - 1$.

Notice that it is not necessary that this wrapped subtraction be able to reach all values $0, \ldots, m - 1$ from *every* value of the first sequence. Consider the absurd extreme case where the value subtracted was only between 1 and 10: The resulting sequence would still be no less random than the first sequence by itself. As a practical matter it is only necessary that the second sequence have a range covering *substantially* all of the range of the first. L'Ecuyer recommends the use of the two generators $m_1 = 2147483563$ (with $a_1 = 40014$, $q_1 = 53668$, $r_1 = 12211$) and $m_2 = 2147483399$ (with $a_2 = 40692$, $q_2 = 52774$, $r_2 = 3791$). Both moduli are slightly less than 2^{31}. The periods $m_1 - 1 = 2 \times 3 \times 7 \times 631 \times 81031$ and $m_2 - 1 = 2 \times 19 \times 31 \times 1019 \times 1789$ share only the factor 2, so the period of the combined generator is $\approx 2.3 \times 10^{18}$. For present computers, period exhaustion is a practical impossibility.

Combining the two generators breaks up serial correlations to a considerable extent. We nevertheless recommend the additional shuffle that is implemented in the following routine, ran2. We think that, within the limits of its floating-point precision, ran2 provides perfect random numbers; a practical definition of "perfect" is that we will pay $1000 to the first reader who convinces us otherwise (by finding a statistical test that ran2 fails in a nontrivial way, excluding the ordinary limitations of a machine's floating-point representation).

random numbers

```
#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)

float ran1(long *idum)
{
        int j;
        long k;
        static long iy=0;
        static long iv[NTAB];
        float 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;
}
#undef IA
#undef IM
#undef AM
#undef IQ
#undef IR
#undef NTAB
#undef NDIV
#undef EPS
#undef RNMX
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

ran1.c

`float ran1(long *idum)`
"Minimal" random number generator of Park and Miller with Bays-Durham shuffle and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). Call with `idum` a negative integer to initialize; thereafter, do not alter `idum` between successive deviates in a sequence. `RNMX` should approximate the largest floating value that is less than 1.

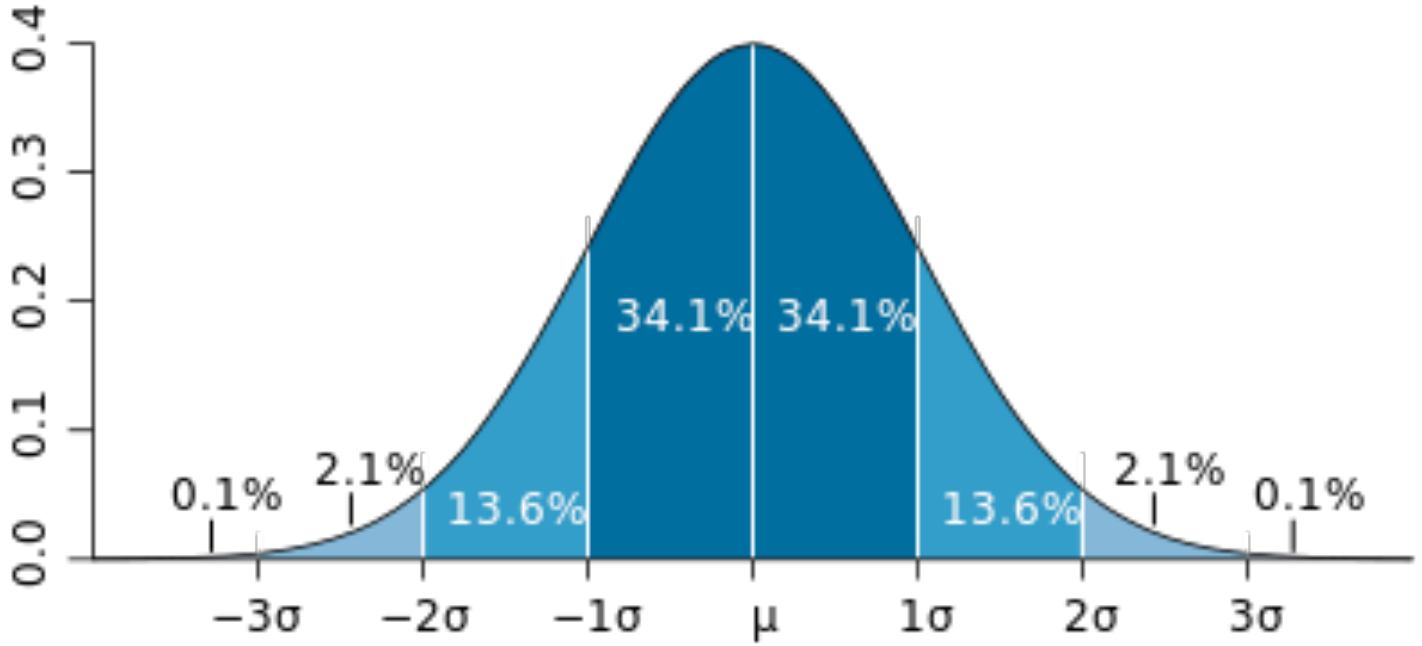
random numbers

```
#include <stdio.h>

#include "ran1.c"

#define MAX 10

int main()
 {
  int i;
  long x=-20;

  ran1(&x);

  for(i=0; i<MAX; i++)
   printf("%20f \n", ran1(&x) );
 }
```

random1.c

random numbers - Gaussian distribution

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy$$



probability density function of a normal distribution (Gaussian distribution)

7.2 Transformation Method: Exponential and Normal Deviates

In the previous section, we learned how to generate random deviates with a uniform probability distribution, so that the probability of generating a number between x and $x + dx$, denoted $p(x)dx$, is given by

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{7.2.1}$$

The probability distribution $p(x)$ is of course normalized, so that

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{7.2.2}$$

Now suppose that we generate a uniform deviate x and then take some prescribed function of it, $y(x)$. The probability distribution of y, denoted $p(y)dy$, is determined by the fundamental transformation law of probabilities, which is simply

$$|p(y)dy| = |p(x)dx| \tag{7.2.3}$$

or

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \tag{7.2.4}$$

Normal (Gaussian) Deviates

Transformation methods generalize to more than one dimension. If $x_1, x_2, \ldots$ are random deviates with a *joint* probability distribution $p(x_1, x_2, \ldots)\, dx_1 dx_2 \ldots$, and if $y_1, y_2, \ldots$ are each functions of all the x's (same number of y's as x's), then the joint probability distribution of the y's is

$$p(y_1, y_2, \ldots)dy_1 dy_2 \ldots = p(x_1, x_2, \ldots)\left|\frac{\partial(x_1, x_2, \ldots)}{\partial(y_1, y_2, \ldots)}\right| dy_1 dy_2 \ldots \tag{7.2.8}$$

where $|\partial(\quad)/\partial(\quad)|$ is the Jacobian determinant of the x's with respect to the y's (or reciprocal of the Jacobian determinant of the y's with respect to the x's).

An important example of the use of (7.2.8) is the *Box-Muller* method for generating random deviates with a normal (Gaussian) distribution,

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \tag{7.2.9}$$

Consider the transformation between two uniform deviates on (0,1), x_1, x_2, and two quantities y_1, y_2,

$$\begin{aligned} y_1 &= \sqrt{-2\ln x_1}\cos 2\pi x_2 \\ y_2 &= \sqrt{-2\ln x_1}\sin 2\pi x_2 \end{aligned} \tag{7.2.10}$$

Equivalently we can write

$$\begin{aligned} x_1 &= \exp\left[-\frac{1}{2}(y_1^2 + y_2^2)\right] \\ x_2 &= \frac{1}{2\pi}\arctan\frac{y_2}{y_1} \end{aligned} \tag{7.2.11}$$

Now the Jacobian determinant can readily be calculated (try it!):

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}} e^{-y_1^2/2}\right]\left[\frac{1}{\sqrt{2\pi}} e^{-y_2^2/2}\right] \tag{7.2.12}$$

Since this is the product of a function of y_2 alone and a function of y_1 alone, we see that each y is independently distributed according to the normal distribution (7.2.9).

One further trick is useful in applying (7.2.10). Suppose that, instead of picking uniform deviates x_1 and x_2 in the unit square, we instead pick v_1 and v_2 as the ordinate and abscissa of a random point inside the unit circle around the origin. Then the sum of their squares, $R^2 \equiv v_1^2 + v_2^2$ is a uniform deviate, which can be used for x_1, while the angle that (v_1, v_2) defines with respect to the v_1 axis can serve as the random angle $2\pi x_2$. What's the advantage? It's that the cosine and sine in (7.2.10) can now be written as $v_1/\sqrt{R^2}$ and $v_2/\sqrt{R^2}$, obviating the trigonometric function calls!

```
#include <math.h>

float gasdev(long *idum)
{
        float ran1(long *idum);
        static int iset=0;
        static float gset;
        float fac,rsq,v1,v2;

        if  (iset == 0) {
                do {
                        v1=2.0*ran1(idum)-1.0;
                        v2=2.0*ran1(idum)-1.0;
                        rsq=v1*v1+v2*v2;
                } while (rsq >= 1.0 || rsq == 0.0);
                fac=sqrt(-2.0*log(rsq)/rsq);
                gset=v1*fac;
                iset=1;
                return v2*fac;
        } else {
                iset=0;
                return gset;
        }
}
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

`float gasdev(long *idum)`
Returns a normally distributed deviate with zero mean and unit variance, using `ran1(idum)` as the source of uniform deviates.

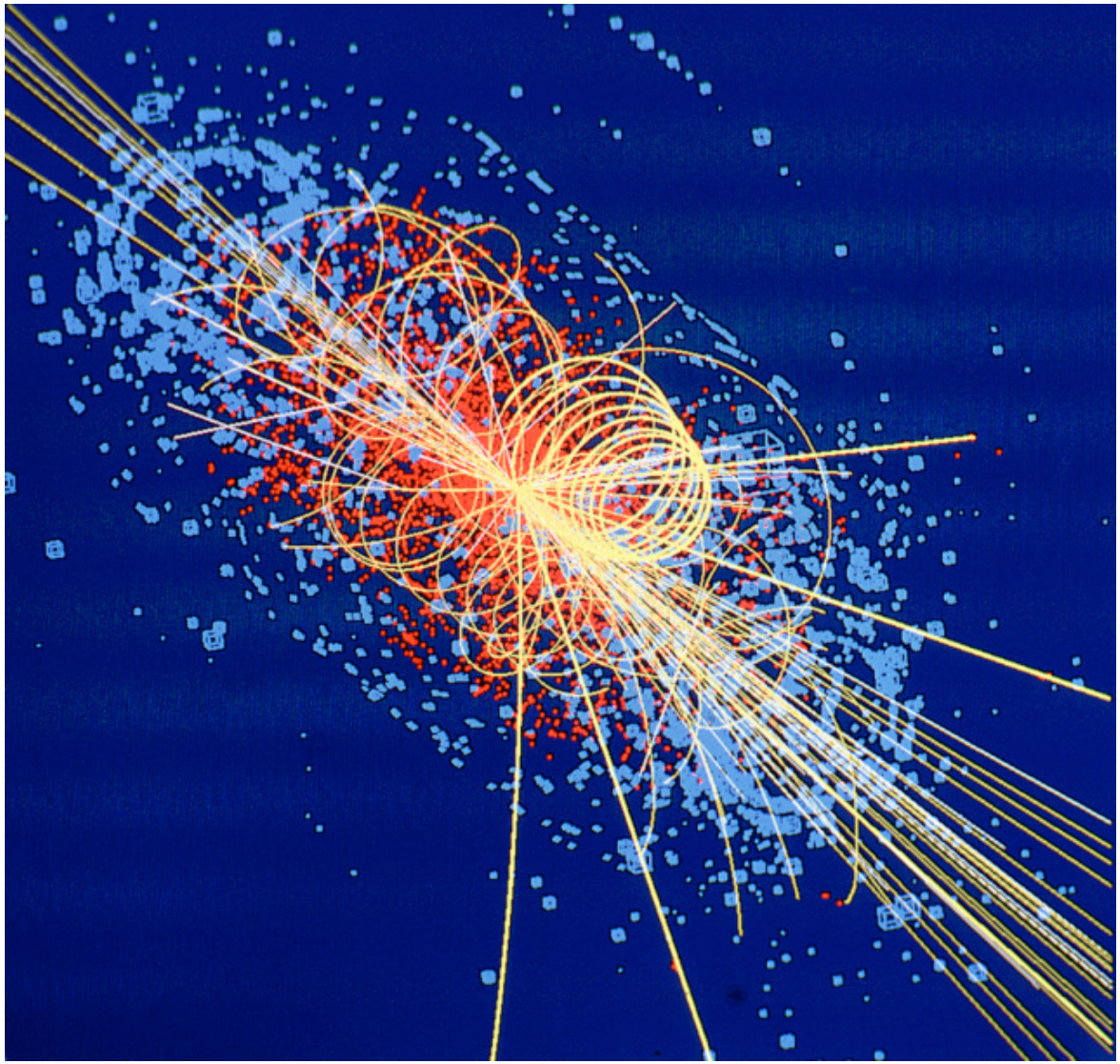
gasdev.c

indexing and ranking

Voor een analyse van LHC data hebben wij een lijst met 100000 events. Wij willen deze lijst sorteren.

Voor elke event hebben wij een set van parameters.

$p_1, p_2, .. p_n$



een Higgs boson in de CMS detector aan de LHC

indexing and ranking

Numerical Recipes

8.4 Indexing and Ranking

The concept of *keys* plays a prominent role in the management of data files. A data *record* in such a file may contain several items, or *fields*. For example, a record in a file of weather observations may have fields recording time, temperature, and wind velocity. When we sort the records, we must decide which of these fields we want to be brought into sorted order. The other fields in a record just come along for the ride, and will not, in general, end up in any particular order. The field on which the sort is performed is called the *key* field.

For a data file with many records and many fields, the actual movement of N records into the sorted order of their keys $K_i,\ i = 1, \ldots, N$, can be a daunting task. Instead, one can construct an *index table* $I_j,\ j = 1, \ldots, N$, such that the smallest K_i has $i = I_1$, the second smallest has $i = I_2$, and so on up to the largest K_i with $i = I_N$. In other words, the array

$$K_{I_j} \quad j = 1, 2, \ldots, N \tag{8.4.1}$$

is in sorted order when indexed by j. When an index table is available, one need not move records from their original order. Further, different index tables can be made from the same set of records, indexing them to different keys.

The algorithm for constructing an index table is straightforward: Initialize the index array with the integers from 1 to N, then perform the Quicksort algorithm, moving the elements around *as if* one were sorting the keys. The integer that initially numbered the smallest key thus ends up in the number one position, and so on.

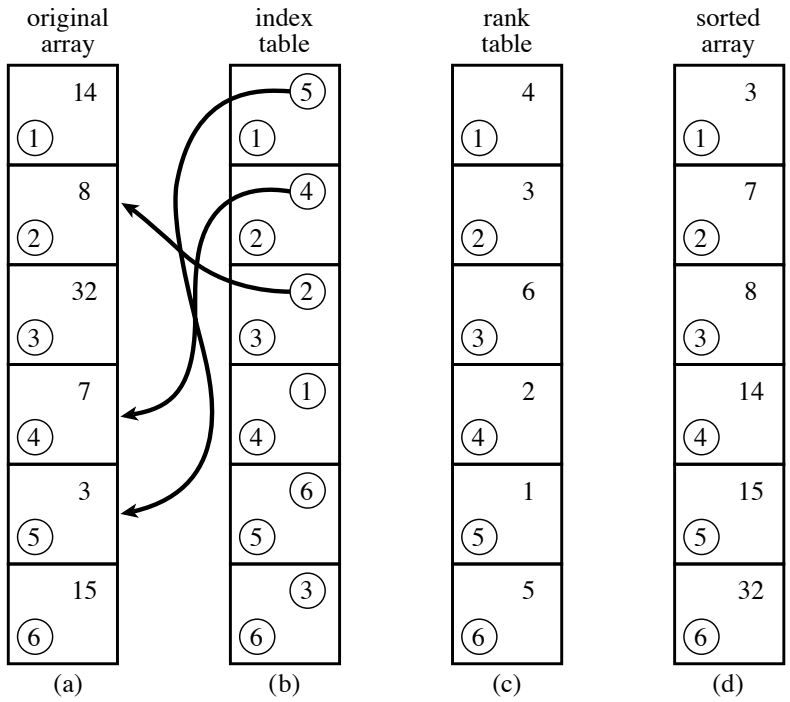


Figure 8.4.1. (a) An unsorted array of six numbers. (b) Index table, whose entries are pointers to the elements of (a) in ascending order. (c) Rank table, whose entries are the ranks of the corresponding elements of (a). (d) Sorted array of the elements in (a).

indexing and ranking

```
#define NRANSI
#include "nrutil.h"
#define SWAP(a,b) itemp=(a);(a)=(b);(b)=itemp;
#define M 7
#define NSTACK 50

void indexx(unsigned long n, float arr[], unsigned long indx[])
{
	unsigned long i,indxt,ir=n,itemp,j,k,l=1;
	int jstack=0,*istack;
	float a;

	istack=ivector(1,NSTACK);
	for (j=1;j<=n;j++) indx[j]=j;
	for (;;) {
		if (ir-l < M) {
			for (j=l+1;j<=ir;j++) {
				indxt=indx[j];
				a=arr[indxt];
				for (i=j-1;i>=1;i--) {
					if (arr[indx[i]] <= a) break;
					indx[i+1]=indx[i];
				}
				indx[i+1]=indxt;
			}
			if (jstack == 0) break;
			ir=istack[jstack--];
			l=istack[jstack--];
		} else {
			k=(l+ir) >> 1;
			SWAP(indx[k],indx[l+1]);
			if (arr[indx[l+1]] > arr[indx[ir]]) {
				SWAP(indx[l+1],indx[ir])
			}
			if (arr[indx[l]] > arr[indx[ir]]) {
				SWAP(indx[l],indx[ir])
			}
			if (arr[indx[l+1]] > arr[indx[l]]) {
				SWAP(indx[l+1],indx[l])
			}
			i=l+1;
			j=ir;
			indxt=indx[l];
			a=arr[indxt];
			for (;;) {
				do i++; while (arr[indx[i]] < a);
				do j--; while (arr[indx[j]] > a);
				if (j < i) break;
				SWAP(indx[i],indx[j])
			}
			indx[l]=indx[j];
			indx[j]=indxt;
			jstack += 2;
			if (jstack > NSTACK) nrerror("NSTACK too small in indexx.");
			if (ir-i+1 >= j-l) {
				istack[jstack]=ir;
				istack[jstack-1]=i;
				ir=j-1;
			} else {
				istack[jstack]=j-1;
				istack[jstack-1]=l;
				l=i;
			}
		}
	}
	free_ivector(istack,1,NSTACK);
}
#undef M
#undef NSTACK
#undef SWAP
#undef NRANSI
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

indexx.c

`void indexx(unsigned long n, float arr[], unsigned long indx[])`
Indexes an array `arr[1..n]`, i.e., outputs the array `indx[1..n]` such that `arr[indx[j]]` is in ascending order for $j = 1, 2, \ldots, N$. The input quantities `n` and `arr` are not changed.

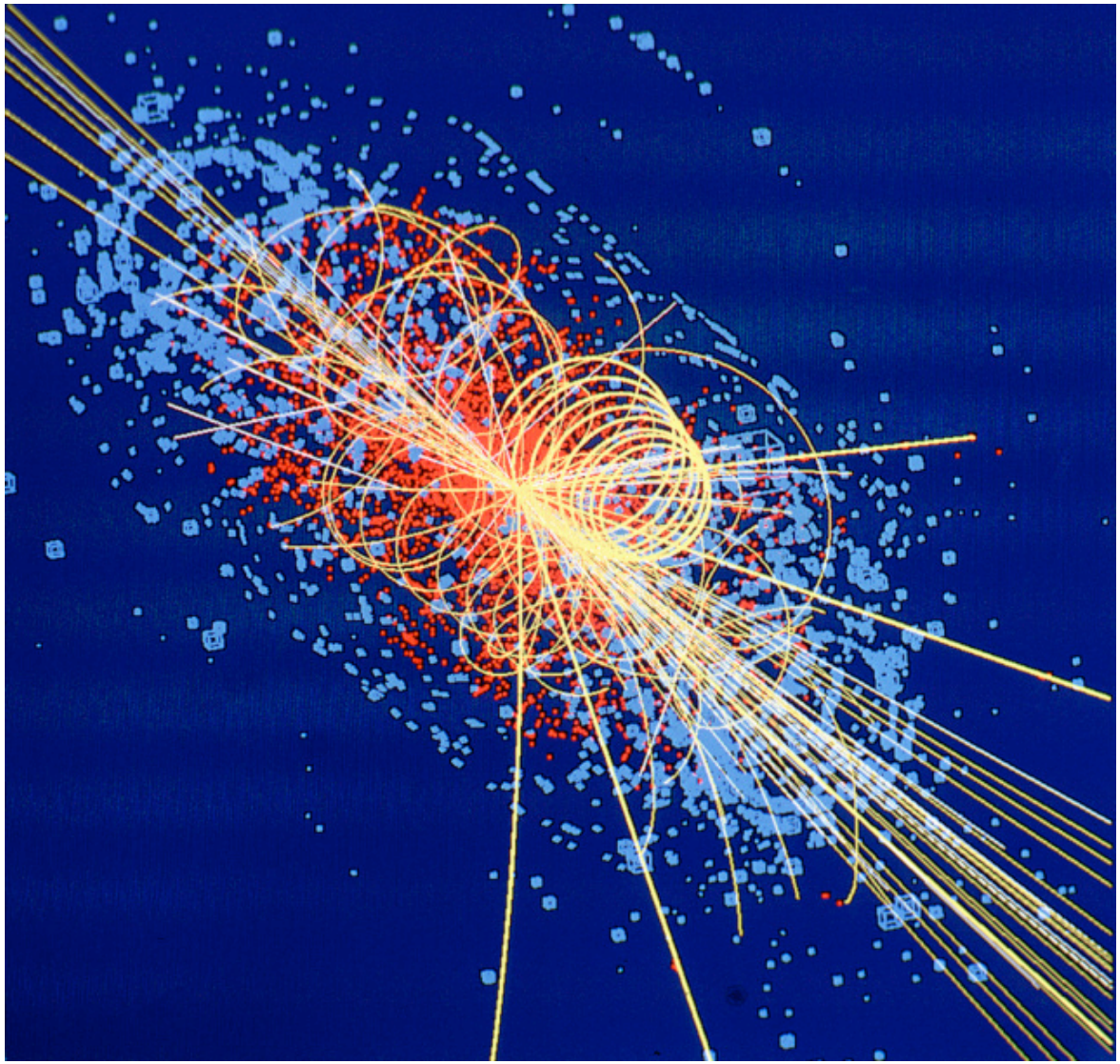
```
#include <stdio.h>

//#define NEVENT 100000
#define NEVENT 6

#include "nrutil.c"
#include "indexx.c"

void generate_list()
 {
  int i;
  FILE *fp;

  fp=fopen("higgs.dat","w");
  for(i=0; i<NEVENT; i++)
   fprintf(fp,"%f \n",(double)rand() );
  fclose(fp);
 }
```

```
int main()
 {
  int n=1;
  float array[NEVENT+1];
  unsigned long index[NEVENT+1];
  FILE *fp;

  //generate_list();

  // read data from file
  fp=fopen("higgs.dat","r");
  while(fscanf(fp,"%f",&array[n])!=EOF && n<=NEVENT) n+
+;
  fclose(fp);

  // print unsorted list
  for(n=1; n<=NEVENT; n++)
   printf("array[%5i] = %f\n", n, array[n]);

  // sort
  indexx(NEVENT, array, index);

  // print index table
  for(n=1; n<=NEVENT; n++)
   printf("index[%i] = %li\n", n, index[n]);

  // print sorted list
  for(n=1; n<=NEVENT; n++)
   printf("array[%5li] = %f\n", index[n], array[index[n]]);
 }
```

higgs.c

heapsort

Nog een keer sorteren: heapsort



een Higgs boson in de CMS detector aan de LHC

8.3 Heapsort

While usually not quite as fast as Quicksort, Heapsort is one of our favorite sorting routines. It is a true "in-place" sort, requiring no auxiliary storage. It is an $N \log_2 N$ process, not only on average, but also for the worst-case order of input data. In fact, its worst case is only 20 percent or so worse than its average running time.

It is beyond our scope to give a complete exposition on the theory of Heapsort. We will mention the general principles, then let you refer to the references [1,2], or analyze the program yourself, if you want to understand the details.

A set of N numbers a_i, $i = 1, \ldots, N$, is said to form a "heap" if it satisfies the relation

$$a_{j/2} \geq a_j \quad \text{for} \quad 1 \leq j/2 < j \leq N \tag{8.3.1}$$

Here the division in $j/2$ means "integer divide," i.e., is an exact integer or else is rounded down to the closest integer. Definition (8.3.1) will make sense if you think of the numbers a_i as being arranged in a binary tree, with the top, "boss," node being a_1, the two "underling" nodes being a_2 and a_3, *their* four underling nodes being a_4 through a_7, etc. (See Figure 8.3.1.) In this form, a heap has every "supervisor" greater than or equal to its two "supervisees," down through the levels of the hierarchy.

If you have managed to rearrange your array into an order that forms a heap, then sorting it is very easy: You pull off the "top of the heap," which will be the largest element yet unsorted. Then you "promote" to the top of the heap its largest underling. Then you promote *its* largest underling, and so on. The process is like what happens (or is supposed to happen) in a large corporation when the chairman of the board retires. You then repeat the whole process by retiring the new chairman of the board. Evidently the whole thing is an $N \log_2 N$ process, since each retiring chairman leads to $\log_2 N$ promotions of underlings.

Well, how do you arrange the array into a heap in the first place? The answer is again a "sift-up" process like corporate promotion. Imagine that the corporation starts out with $N/2$ employees on the production line, but with no supervisors. Now a supervisor is hired to supervise two workers. If he is less capable than one of his workers, that one is promoted in his place, and he joins the production line. After supervisors are hired, then supervisors of supervisors are hired, and so on up the corporate ladder. Each employee is brought in at the top of the tree, but then immediately sifted down, with more capable workers promoted until their proper corporate level has been reached.

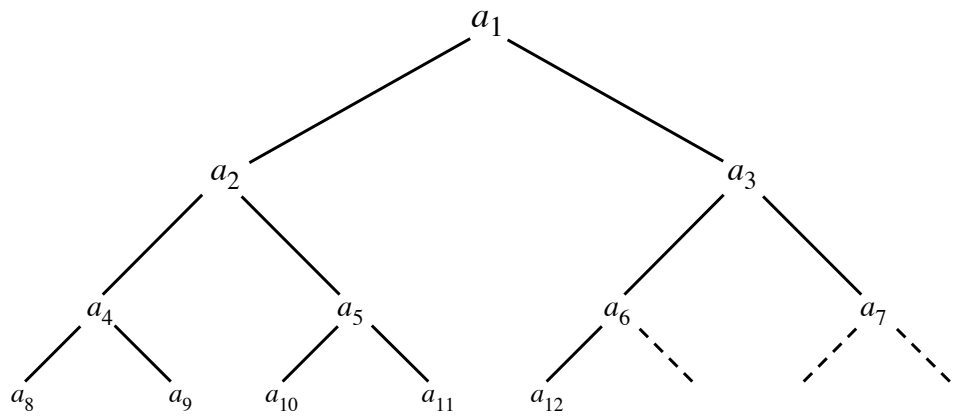


Figure 8.3.1. Ordering implied by a "heap," here of 12 elements. Elements connected by an upward path are sorted with respect to one another, but there is not necessarily any ordering among elements related only "laterally."

In the Heapsort implementation, the same "sift-up" code can be used for the initial creation of the heap and for the subsequent retirement-and-promotion phase. One execution of the Heapsort function represents the entire life-cycle of a giant corporation: $N/2$ workers are hired; $N/2$ potential supervisors are hired; there is a sifting up in the ranks, a sort of super Peter Principle: in due course, each of the original employees gets promoted to chairman of the board.

heapsort

```
void hpsort(unsigned long n, float ra[])
{
        unsigned long i,ir,j,l;
        float rra;

        if (n < 2) return;
        l=(n >> 1)+1;
        ir=n;
        for (;;) {
                if (l > 1) {
                        rra=ra[--l];
                } else {
                        rra=ra[ir];
                        ra[ir]=ra[1];
                        if (--ir == 1) {
                                ra[1]=rra;
                                break;
                        }
                }
                i=l;
                j=l+l;
                while (j <= ir) {
                        if (j < ir && ra[j] < ra[j+1]) j++;
                        if (rra < ra[j]) {
                                ra[i]=ra[j];
                                i=j;
                                j <<= 1;
                        } else j=ir+1;
                }
                ra[i]=rra;
        }
}
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

hpsort.c

```
#include <stdio.h>

#define MAXDATA 6

#include "hpsort.c"

int main()
 {
  int n;
  float array[MAXDATA+1];

  // generate list
  for(n=1; n<=MAXDATA; n++)
   array[n]=rand();
  printf("unsorted data\n");
  for(n=1; n<=MAXDATA; n++)
   printf("%20f \n",array[n]);

  // sort
  hpsort(MAXDATA, array);

  // print sorted list
  printf("sorted data\n");
  for(n=1; n<=MAXDATA; n++)
   printf("%20f \n",array[n]);
 }
```

heapsort.c

Lineaire algebra

Numerical Recipes

Chapter 2. Solution of Linear Algebraic Equations

2.0 Introduction

A set of linear algebraic equations looks like this:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3N}x_N &= b_3 \\ \cdots \qquad\qquad & \cdots \\ a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \cdots + a_{MN}x_N &= b_M \end{aligned} \tag{2.0.1}$$

Here the N unknowns x_j, $j = 1, 2, \ldots, N$ are related by M equations. The coefficients a_{ij} with $i = 1, 2, \ldots, M$ and $j = 1, 2, \ldots, N$ are known numbers, as are the *right-hand side* quantities b_i, $i = 1, 2, \ldots, M$.

Matrices

Equation (2.0.1) can be written in matrix form as

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.0.2}$$

Here the raised dot denotes matrix multiplication, $\mathbf{A}$ is the matrix of coefficients, and $\mathbf{b}$ is the right-hand side written as a column vector,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1N} \\ a_{21} & a_{22} & \ldots & a_{2N} \\ & \cdots & & \\ a_{M1} & a_{M2} & \ldots & a_{MN} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_M \end{bmatrix} \tag{2.0.3}$$

2.3 LU Decomposition and Its Applications

Suppose we are able to write the matrix $\mathbf{A}$ as a product of two matrices,

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \tag{2.3.1}$$

where $\mathbf{L}$ is *lower triangular* (has elements only on the diagonal and below) and $\mathbf{U}$ is *upper triangular* (has elements only on the diagonal and above). For the case of a 4×4 matrix $\mathbf{A}$, for example, equation (2.3.1) would look like this:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{2.3.2}$$

We can use a decomposition such as (2.3.1) to solve the linear set

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \tag{2.3.3}$$

by first solving for the vector $\mathbf{y}$ such that

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{2.3.4}$$

and then solving

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{2.3.5}$$

What is the advantage of breaking up one linear set into two successive ones? The advantage is that the solution of a triangular set of equations is quite trivial, as we have already seen in §2.2 (equation 2.2.4). Thus, equation (2.3.4) can be solved by *forward substitution* as follows,

$$\begin{aligned} y_1 &= \frac{b_1}{\alpha_{11}} \\ y_i &= \frac{1}{\alpha_{ii}} \left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right] \qquad i = 2, 3, \ldots, N \end{aligned} \tag{2.3.6}$$

while (2.3.5) can then be solved by *backsubstitution* exactly as in equations (2.2.2)–(2.2.4),

$$\begin{aligned} x_N &= \frac{y_N}{\beta_{NN}} \\ x_i &= \frac{1}{\beta_{ii}} \left[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j \right] \qquad i = N-1, N-2, \ldots, 1 \end{aligned} \tag{2.3.7}$$

Equations (2.3.6) and (2.3.7) total (for each right-hand side $\mathbf{b}$) N^2 executions of an inner loop containing one multiply and one add. If we have N right-hand sides which are the unit column vectors (which is the case when we are inverting a matrix), then taking into account the leading zeros reduces the total execution count of (2.3.6) from $\frac{1}{2}N^3$ to $\frac{1}{6}N^3$, while (2.3.7) is unchanged at $\frac{1}{2}N^3$.

Notice that, once we have the LU decomposition of $\mathbf{A}$, we can solve with as many right-hand sides as we then care to, one at a time. This is a distinct advantage over the methods of §2.1 and §2.2.

Performing the LU Decomposition

How then can we solve for $\mathbf{L}$ and $\mathbf{U}$, given $\mathbf{A}$? First, we write out the i, jth component of equation (2.3.1) or (2.3.2). That component always is a sum beginning with

$$\alpha_{i1}\beta_{1j} + \cdots = a_{ij}$$

The number of terms in the sum depends, however, on whether i or j is the smaller number. We have, in fact, the three cases,

$$i < j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{ij} = a_{ij} \tag{2.3.8}$$

$$i = j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{jj} = a_{ij} \tag{2.3.9}$$

$$i > j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ij}\beta_{jj} = a_{ij} \tag{2.3.10}$$

Equations (2.3.8)–(2.3.10) total N^2 equations for the $N^2 + N$ unknown α's and β's (the diagonal being represented twice). Since the number of unknowns is greater than the number of equations, we are invited to specify N of the unknowns arbitrarily and then try to solve for the others. In fact, as we shall see, it is always possible to take

$$\alpha_{ii} \equiv 1 \qquad i = 1, \ldots, N \tag{2.3.11}$$

A surprising procedure, now, is *Crout's algorithm*, which quite trivially solves the set of $N^2 + N$ equations (2.3.8)–(2.3.11) for all the α's and β's by just arranging the equations in a certain order! That order is as follows:

- Set $\alpha_{ii} = 1$, $i = 1, \ldots, N$ (equation 2.3.11).
- For each $j = 1, 2, 3, \ldots, N$ do these two procedures: First, for $i = 1, 2, \ldots, j$, use (2.3.8), (2.3.9), and (2.3.11) to solve for β_{ij}, namely

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}. \tag{2.3.12}$$

(When $i = 1$ in 2.3.12 the summation term is taken to mean zero.) Second, for $i = j+1, j+2, \ldots, N$ use (2.3.10) to solve for α_{ij}, namely

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right). \tag{2.3.13}$$

Be sure to do both procedures before going on to the next j.

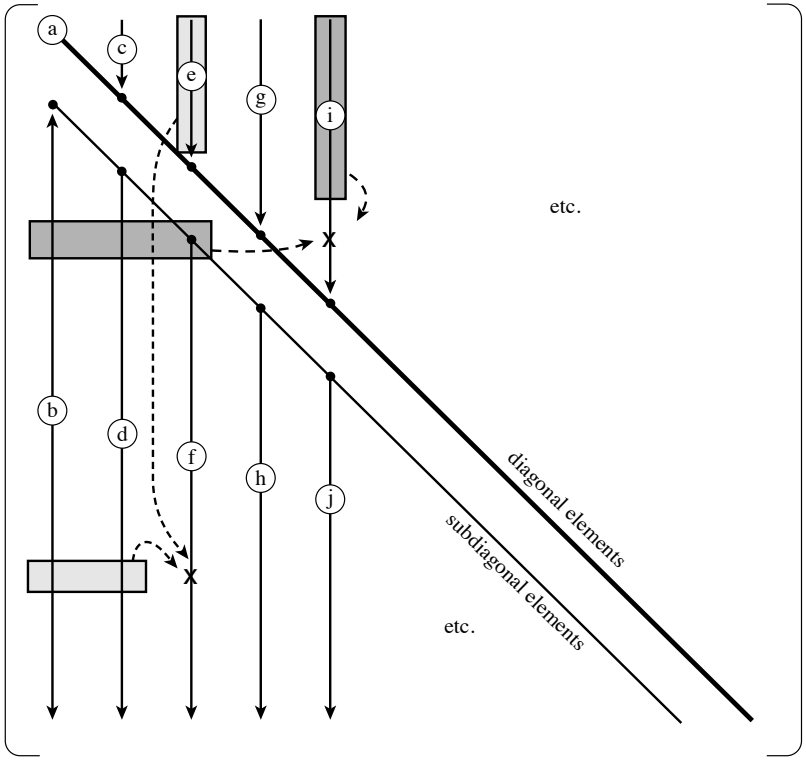


Figure 2.3.1. Crout's algorithm for *LU* decomposition of a matrix. Elements of the original matrix are modified in the order indicated by lower case letters: a, b, c, etc. Shaded boxes show the previously modified elements that are used in modifying two typical elements, each indicated by an "x".

If you work through a few iterations of the above procedure, you will see that the α's and β's that occur on the right-hand side of equations (2.3.12) and (2.3.13) are already determined by the time they are needed. You will also see that every a_{ij} is used only once and never again. This means that the corresponding α_{ij} or β_{ij} can be stored in the location that the a used to occupy: the decomposition is "in place." [The diagonal unity elements α_{ii} (equation 2.3.11) are not stored at all.] In brief, Crout's method fills in the combined matrix of α's and β's,

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix} \tag{2.3.14}$$

by columns from left to right, and within each column from top to bottom (see Figure 2.3.1).

What about pivoting? Pivoting (i.e., selection of a salubrious pivot element for the division in equation 2.3.13) is absolutely essential for the stability of Crout's

method. Only partial pivoting (interchange of rows) can be implemented efficiently. However this is enough to make the method stable. This means, incidentally, that we don't actually decompose the matrix **A** into LU form, but rather we decompose a rowwise permutation of **A**. (If we keep track of what that permutation is, this decomposition is just as useful as the original one would have been.)

Pivoting is slightly subtle in Crout's algorithm. The key point to notice is that equation (2.3.12) in the case of $i = j$ (its final application) is *exactly the same* as equation (2.3.13) except for the division in the latter equation; in both cases the upper limit of the sum is $k = j - 1\ (= i - 1)$. This means that we don't have to commit ourselves as to whether the diagonal element β_{jj} is the one that happens to fall on the diagonal in the first instance, or whether one of the (undivided) α_{ij}'s below it in the column, $i = j+1, \ldots, N$, is to be "promoted" to become the diagonal β. This can be decided after all the candidates in the column are in hand. As you should be able to guess by now, we will choose the largest one as the diagonal β (pivot element), then do all the divisions by that element *en masse*. This is *Crout's method with partial pivoting*. Our implementation has one additional wrinkle: It initially finds the largest element in each row, and subsequently (when it is looking for the maximal pivot element) scales the comparison *as if* we had initially scaled all the equations to make their maximum coefficient equal to unity; this is the *implicit pivoting* mentioned in §2.1.

`void ludcmp(float **a, int n, int *indx, float *d)`
Given a matrix `a[1..n][1..n]`, this routine replaces it by the LU decomposition of a rowwise permutation of itself. `a` and `n` are input. `a` is output, arranged as in equation (2.3.14) above; `indx[1..n]` is an output vector that records the row permutation effected by the partial pivoting; `d` is output as ± 1 depending on whether the number of row interchanges was even or odd, respectively. This routine is used in combination with `lubksb` to solve linear equations or invert a matrix.

LU decomposition

```
#include <math.h>
#define NRANSI
#include "nrutil.h"
#define TINY 1.0e-20;

void ludcmp(float **a, int n, int *indx, float *d)
{
        int i,imax,j,k;
        float big,dum,sum,temp;
        float *vv;

        vv=vector(1,n);
        *d=1.0;
        for (i=1;i<=n;i++) {
                big=0.0;
                for (j=1;j<=n;j++)
                        if ((temp=fabs(a[i][j])) > big) big=temp;
                if (big == 0.0) nrerror("Singular matrix in routine
ludcmp");
                vv[i]=1.0/big;
        }
        for (j=1;j<=n;j++) {
                for (i=1;i<j;i++) {
                        sum=a[i][j];
                        for (k=1;k<i;k++) sum -= a[i][k]*a[k][j];
                        a[i][j]=sum;
                }
                big=0.0;
```

```
                for (i=j;i<=n;i++) {
                        sum=a[i][j];
                        for (k=1;k<j;k++)
                                sum -= a[i][k]*a[k][j];
                        a[i][j]=sum;
                        if ( (dum=vv[i]*fabs(sum)) >= big) {
                                big=dum;
                                imax=i;
                        }
                }
                if (j != imax) {
                        for (k=1;k<=n;k++) {
                                dum=a[imax][k];
                                a[imax][k]=a[j][k];
                                a[j][k]=dum;
                        }
                        *d = -(*d);
                        vv[imax]=vv[j];
                }
                indx[j]=imax;
                if (a[j][j] == 0.0) a[j][j]=TINY;
                if (j != n) {
                        dum=1.0/(a[j][j]);
                        for (i=j+1;i<=n;i++) a[i][j] *= dum;
                }
        }
        free_vector(vv,1,n);
}
#undef TINY
#undef NRANSI
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

ludcmp.c

LU decomposition

Here is the routine for forward substitution and backsubstitution, implementing equations (2.3.6) and (2.3.7).

```
void lubksb(float **a, int n, int *indx, float b[])
{
        int i,ii=0,ip,j;
        float sum;

        for (i=1;i<=n;i++) {
                ip=indx[i];
                sum=b[ip];
                b[ip]=b[i];
                if (ii)
                        for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
                else if (sum) ii=i;
                b[i]=sum;
        }
        for (i=n;i>=1;i--) {
                sum=b[i];
                for (j=i+1;j<=n;j++) sum -= a[i][j]*b[j];
                b[i]=sum/a[i][i];
        }
}
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

What is the advantage of breaking up one linear set into two successive ones? The advantage is that the solution of a triangular set of equations is quite trivial, as we have already seen in §2.2 (equation 2.2.4). Thus, equation (2.3.4) can be solved by *forward substitution* as follows,

$$y_1 = \frac{b_1}{\alpha_{11}}$$
$$y_i = \frac{1}{\alpha_{ii}}\left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j\right] \qquad i = 2, 3, \ldots, N \tag{2.3.6}$$

while (2.3.5) can then be solved by *backsubstitution* exactly as in equations (2.2.2)–(2.2.4),

$$x_N = \frac{y_N}{\beta_{NN}}$$
$$x_i = \frac{1}{\beta_{ii}}\left[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j\right] \qquad i = N-1, N-2, \ldots, 1 \tag{2.3.7}$$

`void lubksb(float **a, int n, int *indx, float b[])`
Solves the set of `n` linear equations $A \cdot X = B$. Here `a[1..n][1..n]` is input, not as the matrix A but rather as its LU decomposition, determined by the routine `ludcmp`. `indx[1..n]` is input as the permutation vector returned by `ludcmp`. `b[1..n]` is input as the right-hand side vector B, and returns with the solution vector X. `a`, `n`, and `indx` are not modified by this routine and can be left in place for successive calls with different right-hand sides `b`. This routine takes into account the possibility that `b` will begin with many zero elements, so it is efficient for use in matrix inversion.

lubksb.c

The LU decomposition in `ludcmp` requires about $\frac{1}{3}N^3$ executions of the inner loops (each with one multiply and one add). This is thus the operation count for solving one (or a few) right-hand sides, and is a factor of 3 better than the Gauss-Jordan routine `gaussj` which was given in §2.1, and a factor of 1.5 better than a Gauss-Jordan routine (not given) that does not compute the inverse matrix. For inverting a matrix, the total count (including the forward and backsubstitution as discussed following equation 2.3.7 above) is $(\frac{1}{3}+\frac{1}{6}+\frac{1}{2})N^3 = N^3$, the same as `gaussj`.

To summarize, this is the preferred way to solve the linear set of equations $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$:

```
float **a,*b,d;
int n,*indx;
...
ludcmp(a,n,indx,&d);
lubksb(a,n,indx,b);
```

The answer **x** will be given back in `b`. Your original matrix **A** will have been destroyed.

If you subsequently want to solve a set of equations with the same **A** but a different right-hand side **b**, you repeat *only*

```
lubksb(a,n,indx,b);
```

not, of course, with the original matrix **A**, but with `a` and `indx` as were already set by `ludcmp`.

Inverse of a Matrix

Using the above LU decomposition and backsubstitution routines, it is completely straightforward to find the inverse of a matrix column by column.

```
#define N ...
float **a,**y,d,*col;
int i,j,*indx;
...
ludcmp(a,N,indx,&d);                    Decompose the matrix just once.
for(j=1;j<=N;j++) {                     Find inverse by columns.
    for(i=1;i<=N;i++) col[i]=0.0;
    col[j]=1.0;
    lubksb(a,N,indx,col);
    for(i=1;i<=N;i++) y[i][j]=col[i];
}
```

The matrix `y` will now contain the inverse of the original matrix `a`, which will have been destroyed. Alternatively, there is nothing wrong with using a Gauss-Jordan routine like `gaussj` (§2.1) to invert a matrix in place, again destroying the original. Both methods have practically the same operations count.

Incidentally, if you ever have the need to compute $\mathbf{A}^{-1}\cdot\mathbf{B}$ from matrices **A** and **B**, you should LU decompose **A** and then backsubstitute with the columns of **B** instead of with the unit vectors that would give **A**'s inverse. This saves a whole matrix multiplication, and is also more accurate.

Determinant of a Matrix

The determinant of an LU decomposed matrix is just the product of the diagonal elements,

$$\det = \prod_{j=1}^{N} \beta_{jj} \tag{2.3.15}$$

We don't, recall, compute the decomposition of the original matrix, but rather a decomposition of a rowwise permutation of it. Luckily, we have kept track of whether the number of row interchanges was even or odd, so we just preface the product by the corresponding sign. (You now finally know the purpose of setting `d` in the routine `ludcmp`.)

Calculation of a determinant thus requires one call to `ludcmp`, with *no* subsequent backsubstitutions by `lubksb`.

```
#define N ...
float **a,d;
int j,*indx;
...
ludcmp(a,N,indx,&d);          This returns d as ±1.
for(j=1;j<=N;j++) d *= a[j][j];
```

The variable `d` now contains the determinant of the original matrix `a`, which will have been destroyed.

For a matrix of any substantial size, it is quite likely that the determinant will overflow or underflow your computer's floating-point dynamic range. In this case you can modify the loop of the above fragment and (e.g.) divide by powers of ten, to keep track of the scale separately, or (e.g.) accumulate the sum of logarithms of the absolute values of the factors and the sign separately.

Solution of linear algebraic equations

```
#include <stdio.h>

#include "nrutil.c"
#include "ludcmp.c"
#include "lubksb.c"

int main()
 {
  int i,j, n=3, *indx;
  float d, *b, **a;

  // initialize matrix
  a=matrix(1,3,1,3);
  a[1][1]=1; a[1][2]=2; a[1][3]=3;
  a[2][1]=4; a[2][2]=5; a[2][3]=6;
  a[3][1]=7; a[3][2]=8; a[3][3]=9;

  // initialize vector
  b=vector(1,3);
  b[1]=1; b[2]=2; b[3]=3;

  indx=ivector(1,3);

  printf("original\n");
  for(i=1; i<=n; i++)
   {
    for(j=1; j<=n; j++)
      printf("%f ",a[i][j]);
    printf(" %5s  %f   \n",(i==2 ? "* X =" : "      "), b[i]);
   }

  ludcmp(a,n,indx,&d);
  printf("after ludcmp\n");
  for(i=1; i<=n; i++)
   {
    for(j=1; j<=n; j++)
      printf("%f ",a[i][j]);
    printf(" %5s  %f   \n",(i==2 ? "* X =" : "      "), b[i]);
   }

  lubksb(a,n,indx,b);
  printf("after lubksb\n");
  for(i=1; i<=n; i++)
   {
    for(j=1; j<=n; j++)
      printf("%f ",a[i][j]);
    printf(" %5s  %f   \n",(i==2 ? "* X =" : "      "), b[i]);
   }

  // calculate determinant
  for(j=1; j<=n; j++) d *= a[j][j];
  printf("det(A) = %f \n",d);

}
```

matrix.c

Interpolation and extrapolation

Chapter 3. Interpolation and Extrapolation

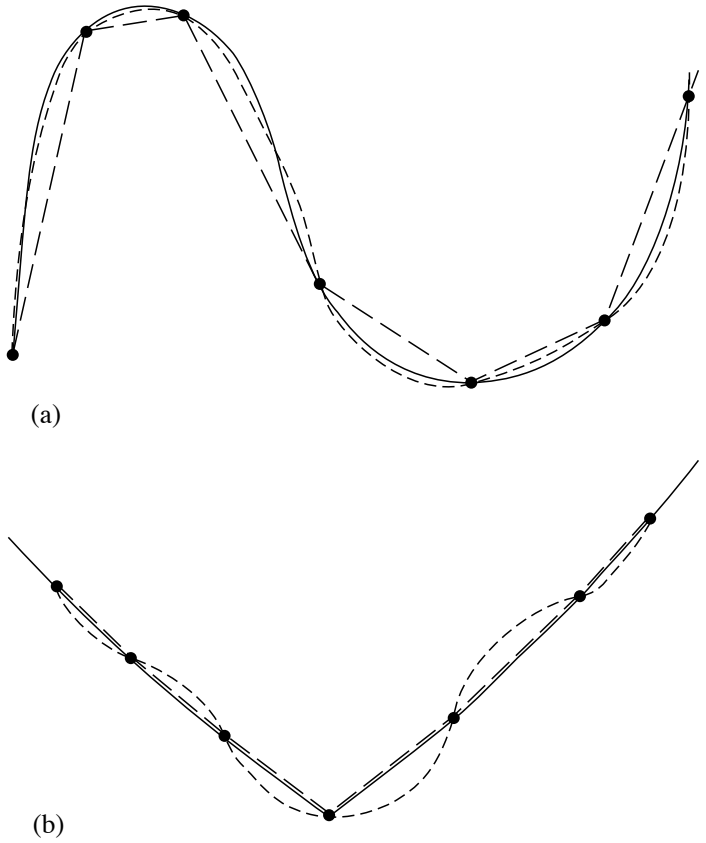


Figure 3.0.1. (a) A smooth function (solid line) is more accurately interpolated by a high-order polynomial (shown schematically as dotted line) than by a low-order polynomial (shown as a piecewise linear dashed line). (b) A function with sharp corners or rapidly changing higher derivatives is *less* accurately approximated by a high-order polynomial (dotted line), which is too "stiff," than by a low-order polynomial (dashed lines). Even some smooth functions, such as exponentials or rational functions, can be badly approximated by high-order polynomials.

3.1 Polynomial Interpolation and Extrapolation

Through any two points there is a unique line. Through any three points, a unique quadratic. Et cetera. The interpolating polynomial of degree $N-1$ through the N points $y_1 = f(x_1), y_2 = f(x_2), \ldots, y_N = f(x_N)$ is given explicitly by Lagrange's classical formula,

$$P(x) = \frac{(x-x_2)(x-x_3)...(x-x_N)}{(x_1-x_2)(x_1-x_3)...(x_1-x_N)}y_1 + \frac{(x-x_1)(x-x_3)...(x-x_N)}{(x_2-x_1)(x_2-x_3)...(x_2-x_N)}y_2 + \cdots + \frac{(x-x_1)(x-x_2)...(x-x_{N-1})}{(x_N-x_1)(x_N-x_2)...(x_N-x_{N-1})}y_N \tag{3.1.1}$$

There are N terms, each a polynomial of degree $N-1$ and each constructed to be zero at all of the x_i except one, at which it is constructed to be y_i.

It is not terribly wrong to implement the Lagrange formula straightforwardly, but it is not terribly right either. The resulting algorithm gives no error estimate, and it is also somewhat awkward to program. A much better algorithm (for constructing the same, unique, interpolating polynomial) is *Neville's algorithm*, closely related to and sometimes confused with *Aitken's algorithm*, the latter now considered obsolete.

Let P_1 be the value at x of the unique polynomial of degree zero (i.e., a constant) passing through the point (x_1, y_1); so $P_1 = y_1$. Likewise define $P_2, P_3, \ldots, P_N$. Now let P_{12} be the value at x of the unique polynomial of degree one passing through both (x_1, y_1) and (x_2, y_2). Likewise $P_{23}, P_{34}, \ldots$, $P_{(N-1)N}$. Similarly, for higher-order polynomials, up to $P_{123\ldots N}$, which is the value of the unique interpolating polynomial through all N points, i.e., the desired answer.

The various P's form a "tableau" with "ancestors" on the left leading to a single "descendant" at the extreme right. For example, with $N = 4$,

$$\begin{array}{lllll} x_1: & y_1 = P_1 & & & \\ & & P_{12} & & \\ x_2: & y_2 = P_2 & & P_{123} & \\ & & P_{23} & & P_{1234} \\ x_3: & y_3 = P_3 & & P_{234} & \\ & & P_{34} & & \\ x_4: & y_4 = P_4 & & & \end{array} \tag{3.1.2}$$

Neville's algorithm is a recursive way of filling in the numbers in the tableau a column at a time, from left to right. It is based on the relationship between a "daughter" P and its two "parents,"

$$P_{i(i+1)\ldots(i+m)} = \frac{(x-x_{i+m})P_{i(i+1)\ldots(i+m-1)} + (x_i - x)P_{(i+1)(i+2)\ldots(i+m)}}{x_i - x_{i+m}} \tag{3.1.3}$$

This recurrence works because the two parents already agree at points $x_{i+1} \ldots x_{i+m-1}$.

An improvement on the recurrence (3.1.3) is to keep track of the small *differences* between parents and daughters, namely to define (for $m = 1, 2, \ldots, N-1$),

$$\begin{aligned} C_{m,i} &\equiv P_{i\ldots(i+m)} - P_{i\ldots(i+m-1)} \\ D_{m,i} &\equiv P_{i\ldots(i+m)} - P_{(i+1)\ldots(i+m)}. \end{aligned} \tag{3.1.4}$$

Then one can easily derive from (3.1.3) the relations

$$\begin{aligned} D_{m+1,i} &= \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \\ C_{m+1,i} &= \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \end{aligned} \tag{3.1.5}$$

At each level m, the C's and D's are the corrections that make the interpolation one order higher. The final answer $P_{1\ldots N}$ is equal to the sum of *any* y_i plus a set of C's and/or D's that form a path through the family tree to the rightmost daughter.

Here is a routine for polynomial interpolation or extrapolation from N input points. Note that the input arrays are assumed to be unit-offset. If you have zero-offset arrays, remember to subtract 1 (see §1.2):

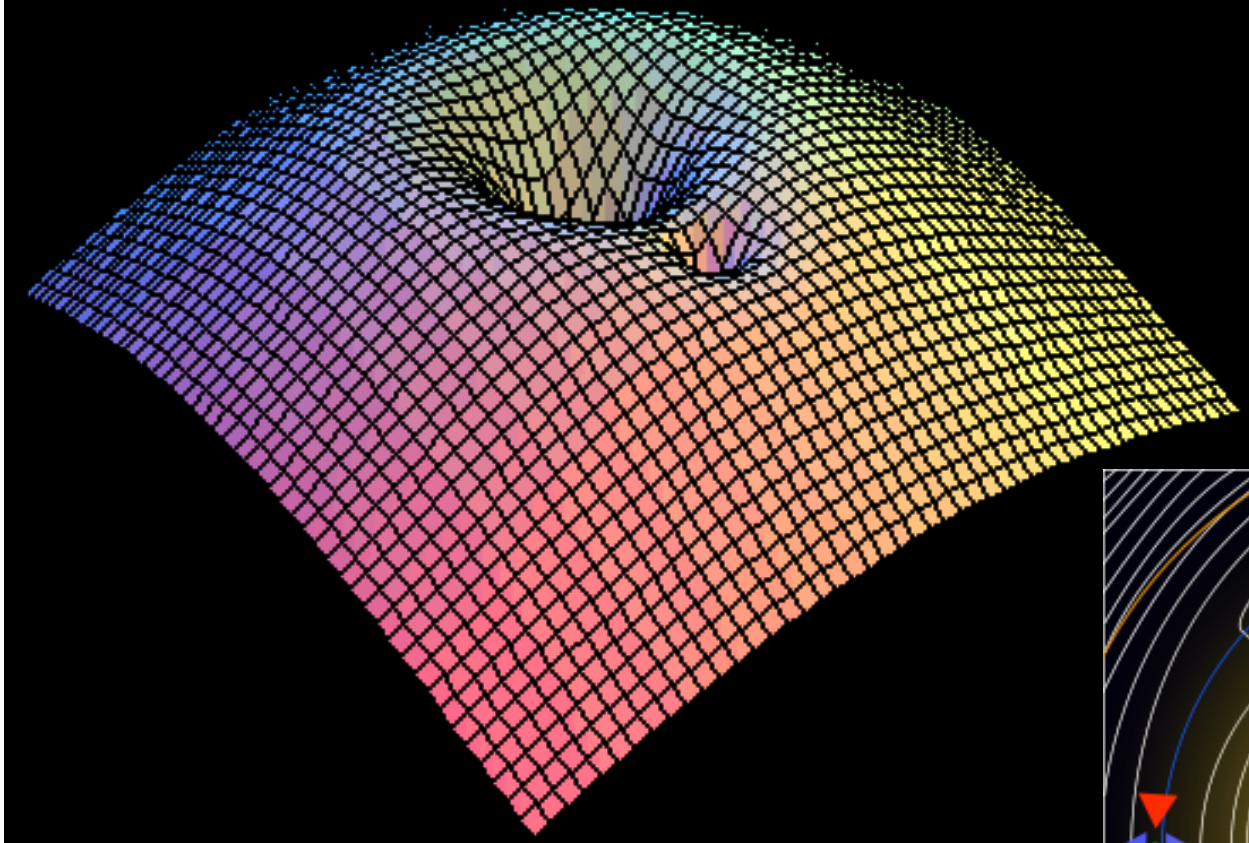
Polynomial interpolation

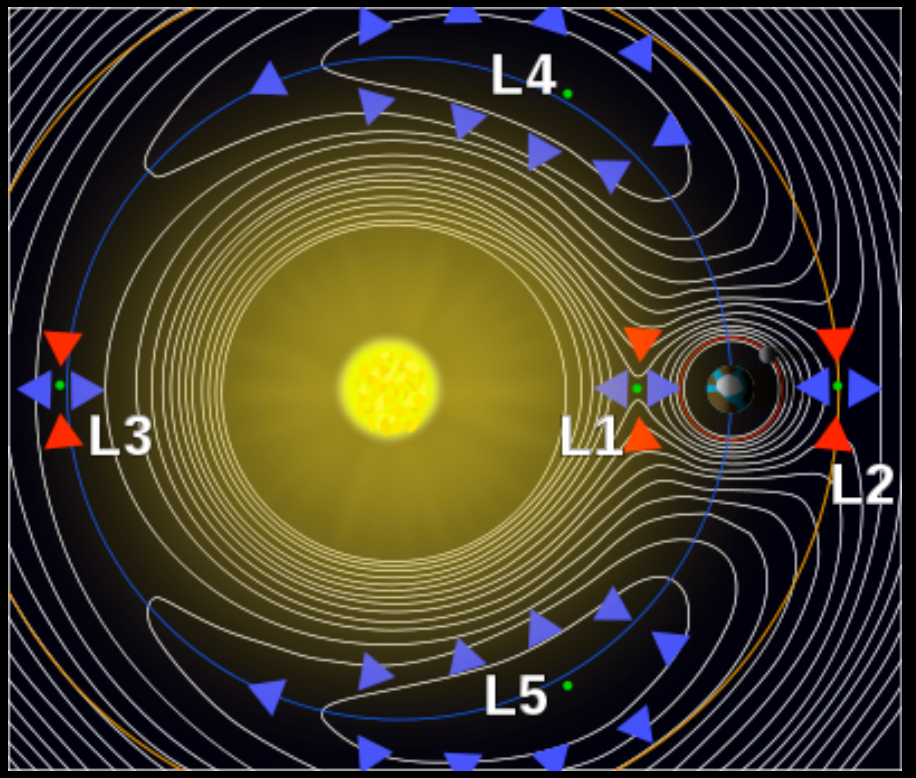
```
#include <math.h>
#define NRANSI
#include "nrutil.h"

void polint(float xa[], float ya[], int n, float x, float *y, float *dy)
{
        int i,m,ns=1;
        float den,dif,dift,ho,hp,w;
        float *c,*d;

        dif=fabs(x-xa[1]);
        c=vector(1,n);
        d=vector(1,n);
        for (i=1;i<=n;i++) {
                if ( (dift=fabs(x-xa[i])) < dif) {
                        ns=i;
                        dif=dift;
                }
                c[i]=ya[i];
                d[i]=ya[i];
        }
        *y=ya[ns--];
        for (m=1;m<n;m++) {
                for (i=1;i<=n-m;i++) {
                        ho=xa[i]-x;
                        hp=xa[i+m]-x;
                        w=c[i+1]-d[i];
                        if ( (den=ho-hp) == 0.0) nrerror("Error in
routine polint");
                        den=w/den;
                        d[i]=hp*den;
                        c[i]=ho*den;
                }
                *y += (*dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
        }
        free_vector(d,1,n);
        free_vector(c,1,n);
}
#undef NRANSI
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

`void polint(float xa[], float ya[], int n, float x, float *y, float *dy)`
Given arrays `xa[1..n]` and `ya[1..n]`, and given a value `x`, this routine returns a value `y`, and an error estimate `dy`. If $P(x)$ is the polynomial of degree $N-1$ such that $P(\mathtt{xa}_i) = \mathtt{ya}_i, i = 1,\ldots,\mathtt{n}$, then the returned value $\mathtt{y} = P(\mathtt{x})$.

polint.c

Minimization or maximization of functions





zwaartekracht potentiaal van de zon en de aarde

Minimization of functions

Numerical Recipes

Chapter 10. Minimization or Maximization of Functions

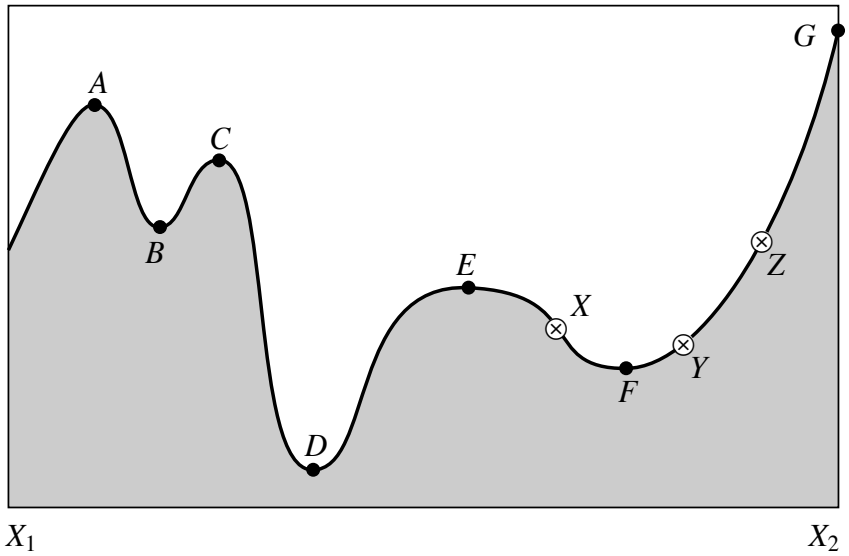


Figure 10.0.1. Extrema of a function in an interval. Points A, C, and E are local, but not global maxima. Points B and F are local, but not global minima. The global maximum occurs at G, which is on the boundary of the interval so that the derivative of the function need not vanish there. The global minimum is at D. At point E, derivatives higher than the first vanish, a situation which can cause difficulty for some algorithms. The points X, Y, and Z are said to "bracket" the minimum F, since Y is less than both X and Z.

Minimization of functions

Numerical Recipes

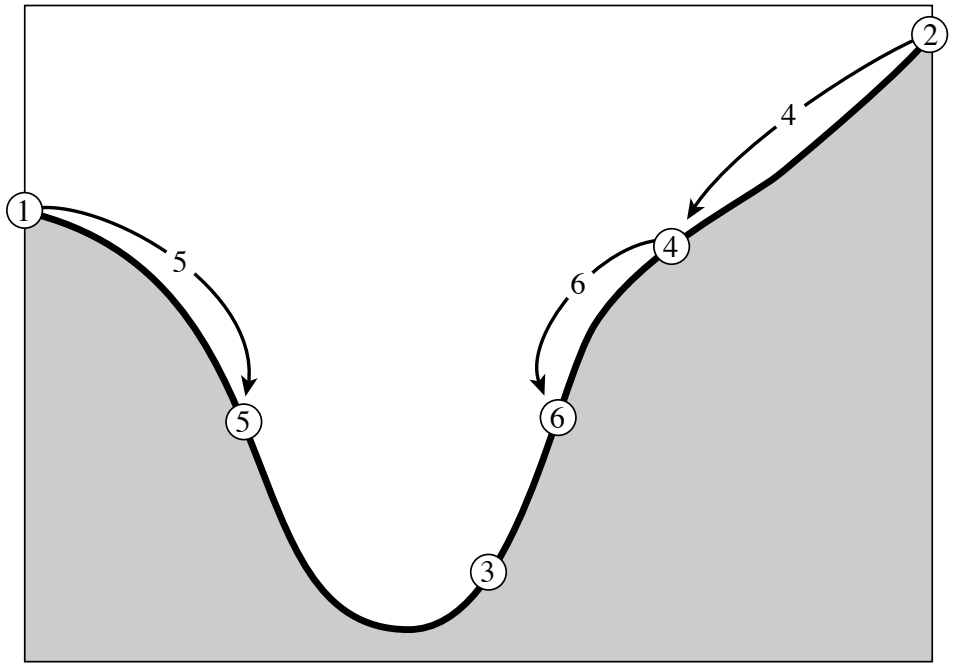


Figure 10.1.1. Successive bracketing of a minimum. The minimum is originally bracketed by points 1,3,2. The function is evaluated at 4, which replaces 2; then at 5, which replaces 1; then at 6, which replaces 4. The rule at each stage is to keep a center point that is lower than the two outside points. After the steps shown, the minimum is bracketed by points 5,3,6.

10.1 Golden Section Search in One Dimension

Recall how the bisection method finds roots of functions in one dimension (§9.1): The root is supposed to have been bracketed in an interval (a, b). One then evaluates the function at an intermediate point x and obtains a new, smaller bracketing interval, either (a, x) or (x, b). The process continues until the bracketing interval is acceptably small. It is optimal to choose x to be the midpoint of (a, b) so that the decrease in the interval length is maximized when the function is as uncooperative as it can be, i.e., when the luck of the draw forces you to take the bigger bisected segment.

There is a precise, though slightly subtle, translation of these considerations to the minimization problem: What does it mean to *bracket* a minimum? A root of a function is known to be bracketed by a pair of points, a and b, when the function has opposite sign at those two points. A minimum, by contrast, is known to be bracketed only when there is a *triplet* of points, $a < b < c$ (or $c < b < a$), such that $f(b)$ is less than both $f(a)$ and $f(c)$. In this case we know that the function (if it is nonsingular) has a minimum in the interval (a, c).

The analog of bisection is to choose a new point x, either between a and b or between b and c. Suppose, to be specific, that we make the latter choice. Then we evaluate $f(x)$. If $f(b) < f(x)$, then the new bracketing triplet of points is (a, b, x); contrariwise, if $f(b) > f(x)$, then the new bracketing triplet is (b, x, c). In all cases the middle point of the new triplet is the abscissa whose ordinate is the best minimum achieved so far; see Figure 10.1.1. We continue the process of bracketing until the distance between the two outer points of the triplet is tolerably small.

How small is "tolerably" small? For a minimum located at a value b, you might naively think that you will be able to bracket it in as small a range as $(1 - \epsilon)b < b < (1 + \epsilon)b$, where ϵ is your computer's floating-point precision, a number like 3×10^{-8} (for `float`) or 10^{-15} (for `double`). Not so! In general, the shape of your function $f(x)$ near b will be given by Taylor's theorem

$$f(x) \approx f(b) + \frac{1}{2} f''(b)(x - b)^2 \tag{10.1.1}$$

The second term will be negligible compared to the first (that is, will be a factor ϵ smaller and will act just like zero when added to it) whenever

$$|x - b| < \sqrt{\epsilon}|b| \sqrt{\frac{2\,|f(b)|}{b^2 f''(b)}} \tag{10.1.2}$$

The reason for writing the right-hand side in this way is that, for most functions, the final square root is a number of order unity. Therefore, as a rule of thumb, it is hopeless to ask for a bracketing interval of width less than $\sqrt{\epsilon}$ times its central value, a fractional width of only about 10^{-4} (single precision) or 3×10^{-8} (double precision). Knowing this inescapable fact will save you a lot of useless bisections!

The minimum-finding routines of this chapter will often call for a user-supplied argument `tol`, and return with an abscissa whose fractional precision is about $\pm$`tol` (bracketing interval of fractional size about 2×`tol`). Unless you have a better

estimate for the right-hand side of equation (10.1.2), you should set `tol` equal to (not much less than) the square root of your machine's floating-point precision, since smaller values will gain you nothing.

It remains to decide on a strategy for choosing the new point x, given (a, b, c). Suppose that b is a fraction w of the way between a and c, i.e.

$$\frac{b-a}{c-a} = w \qquad \frac{c-b}{c-a} = 1-w \tag{10.1.3}$$

Also suppose that our next trial point x is an additional fraction z beyond b,

$$\frac{x-b}{c-a} = z \tag{10.1.4}$$

Then the next bracketing segment will either be of length $w+z$ relative to the current one, or else of length $1-w$. If we want to minimize the worst case possibility, then we will choose z to make these equal, namely

$$z = 1-2w \tag{10.1.5}$$

We see at once that the new point is the symmetric point to b in the original interval, namely with $|b-a|$ equal to $|x-c|$. This implies that the point x lies in the larger of the two segments (z is positive only if $w < 1/2$).

But where in the larger segment? Where did the value of w itself come from? Presumably from the previous stage of applying our same strategy. Therefore, if z is chosen to be optimal, then so was w before it. This *scale similarity* implies that x should be the same fraction of the way from b to c (if that is the bigger segment) as was b from a to c, in other words,

$$\frac{z}{1-w} = w \tag{10.1.6}$$

Equations (10.1.5) and (10.1.6) give the quadratic equation

$$w^2 - 3w + 1 = 0 \qquad \text{yielding} \qquad w = \frac{3-\sqrt{5}}{2} \approx 0.38197 \tag{10.1.7}$$

In other words, the optimal bracketing interval (a, b, c) has its middle point b a fractional distance 0.38197 from one end (say, a), and 0.61803 from the other end (say, b). These fractions are those of the so-called *golden mean* or *golden section*, whose supposedly aesthetic properties hark back to the ancient Pythagoreans. This optimal method of function minimization, the analog of the bisection method for finding zeros, is thus called the *golden section search*, summarized as follows:

Given, at each stage, a bracketing triplet of points, the next point to be tried is that which is a fraction 0.38197 into the larger of the two intervals (measuring from the central point of the triplet). If you start out with a bracketing triplet whose segments are not in the golden ratios, the procedure of choosing successive points at the golden mean point of the larger segment will quickly converge you to the proper, self-replicating ratios.

The golden section search guarantees that each new function evaluation will (after self-replicating ratios have been achieved) bracket the minimum to an interval

just 0.61803 times the size of the preceding interval. This is comparable to, but not quite as good as, the 0.50000 that holds when finding roots by bisection. Note that the convergence is *linear* (in the language of Chapter 9), meaning that successive significant figures are won linearly with additional function evaluations. In the next section we will give a superlinear method, where the rate at which successive significant figures are liberated increases with each successive function evaluation.

```
#include <math.h>
#define R 0.61803399
#define C (1.0-R)
#define SHFT2(a,b,c) (a)=(b);(b)=(c);
#define SHFT3(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

float golden(float ax, float bx, float cx, float (*f)(float), float tol,
	float *xmin)
{
	float f1,f2,x0,x1,x2,x3;

	x0=ax;
	x3=cx;
	if (fabs(cx-bx) > fabs(bx-ax)) {
		x1=bx;
		x2=bx+C*(cx-bx);
	} else {
		x2=bx;
		x1=bx-C*(bx-ax);
	}
	f1=(*f)(x1);
	f2=(*f)(x2);
	while (fabs(x3-x0) > tol*(fabs(x1)+fabs(x2))) {
		if (f2 < f1) {
			SHFT3(x0,x1,x2,R*x1+C*x3)
			SHFT2(f1,f2,(*f)(x2))
		} else {
			SHFT3(x3,x2,x1,R*x2+C*x0)
			SHFT2(f2,f1,(*f)(x1))
		}
	}
	if (f1 < f2) {
		*xmin=x1;
		return f1;
	} else {
		*xmin=x2;
		return f2;
	}
}
#undef C
#undef R
#undef SHFT2
#undef SHFT3
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

`float golden(float ax, float bx, float cx, float (*f)(float), float tol, float *xmin)`

Given a function `f`, and given a bracketing triplet of abscissas `ax`, `bx`, `cx` (such that `bx` is between `ax` and `cx`, and `f(bx)` is less than both `f(ax)` and `f(cx)`), this routine performs a golden section search for the minimum, isolating it to a fractional precision of about `tol`. The abscissa of the minimum is returned as `xmin`, and the minimum function value is returned as `golden`, the returned function value.

golden.c

```
#include <stdio.h>

#include "golden.c"

// the function we want to minimize
float funct(float x)
 {
  return((x-2)*(x-2)+5);
 }

// another function
float funct1(float x)
 {
  return( (x+4) * cos((x-5)*x+17) *exp(x+7));
 }

int main()
 {
  float min,ax,bx,cx,tol,xmin;

// initialize values
  ax=-10;
  bx=0;
  cx=10;
  tol=0.001;

// minimize function
  min=golden(ax,bx,cx, (float(*)(float))funct, tol, &xmin);

  printf("the minimum value is %f at x=%f\n",min,xmin);
 }
```

minimum.c

10.2 Parabolic Interpolation and Brent's Method in One Dimension

We already tipped our hand about the desirability of parabolic interpolation in the previous section's `mnbrak` routine, but it is now time to be more explicit. A golden section search is designed to handle, in effect, the worst possible case of function minimization, with the uncooperative minimum hunted down and cornered like a scared rabbit. But why assume the worst? If the function is nicely parabolic near to the minimum — surely the generic case for sufficiently smooth functions — then the parabola fitted through any three points ought to take us in a single leap to the minimum, or at least very near to it (see Figure 10.2.1). Since we want to find an abscissa rather than an ordinate, the procedure is technically called *inverse parabolic interpolation*.

The formula for the abscissa x that is the minimum of a parabola through three points $f(a)$, $f(b)$, and $f(c)$ is

$$x = b - \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)] - (b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)] - (b-c)[f(b)-f(a)]} \tag{10.2.1}$$

as you can easily derive. This formula fails only if the three points are collinear, in which case the denominator is zero (minimum of the parabola is infinitely far away). Note, however, that (10.2.1) is as happy jumping to a parabolic maximum as to a minimum. No minimization scheme that depends solely on (10.2.1) is likely to succeed in practice.

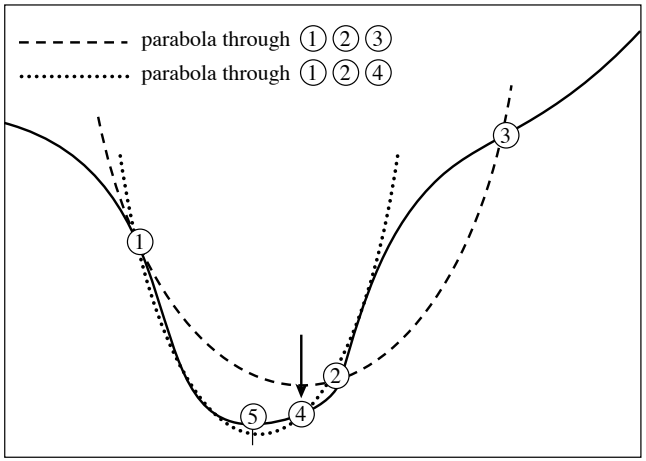


Figure 10.2.1. Convergence to a minimum by inverse parabolic interpolation. A parabola (dashed line) is drawn through the three original points 1,2,3 on the given function (solid line). The function is evaluated at the parabola's minimum, 4, which replaces point 3. A new parabola (dotted line) is drawn through points 1,4,2. The minimum of this parabola is at 5, which is close to the minimum of the function.

The exacting task is to invent a scheme that relies on a sure-but-slow technique, like golden section search, when the function is not cooperative, but that switches over to (10.2.1) when the function allows. The task is nontrivial for several reasons, including these: (i) The housekeeping needed to avoid unnecessary function evaluations in switching between the two methods can be complicated. (ii) Careful attention must be given to the "endgame," where the function is being evaluated very near to the roundoff limit of equation (10.1.2). (iii) The scheme for detecting a cooperative versus noncooperative function must be very robust.

Brent's method [1] is up to the task in all particulars. At any particular stage, it is keeping track of six function points (not necessarily all distinct), a, b, u, v, w and x, defined as follows: the minimum is bracketed between a and b; x is the point with the very least function value found so far (or the most recent one in case of a tie); w is the point with the second least function value; v is the previous value of w; u is the point at which the function was evaluated most recently. Also appearing in the algorithm is the point x_m, the midpoint between a and b; however, the function is not evaluated there.

You can read the code below to understand the method's logical organization. Mention of a few general principles here may, however, be helpful: Parabolic interpolation is attempted, fitting through the points x, v, and w. To be acceptable, the parabolic step must (i) fall within the bounding interval (a, b), and (ii) imply a movement from the best current value x that is *less* than half the movement of the *step before last*. This second criterion insures that the parabolic steps are actually converging to something, rather than, say, bouncing around in some nonconvergent limit cycle. In the worst possible case, where the parabolic steps are acceptable but

useless, the method will approximately alternate between parabolic steps and golden sections, converging in due course by virtue of the latter. The reason for comparing to the step *before* last seems essentially heuristic: Experience shows that it is better not to "punish" the algorithm for a single bad step if it can make it up on the next one.

Another principle exemplified in the code is never to evaluate the function less than a distance `tol` from a point already evaluated (or from a known bracketing point). The reason is that, as we saw in equation (10.1.2), there is simply no information content in doing so: the function will differ from the value already evaluated only by an amount of order the roundoff error. Therefore in the code below you will find several tests and modifications of a potential new point, imposing this restriction. This restriction also interacts subtly with the test for "doneness," which the method takes into account.

A typical ending configuration for Brent's method is that a and b are $2 \times x \times \texttt{tol}$ apart, with x (the best abscissa) at the midpoint of a and b, and therefore fractionally accurate to $\pm\texttt{tol}$.

Indulge us a final reminder that `tol` should generally be no smaller than the square root of your machine's floating-point precision.

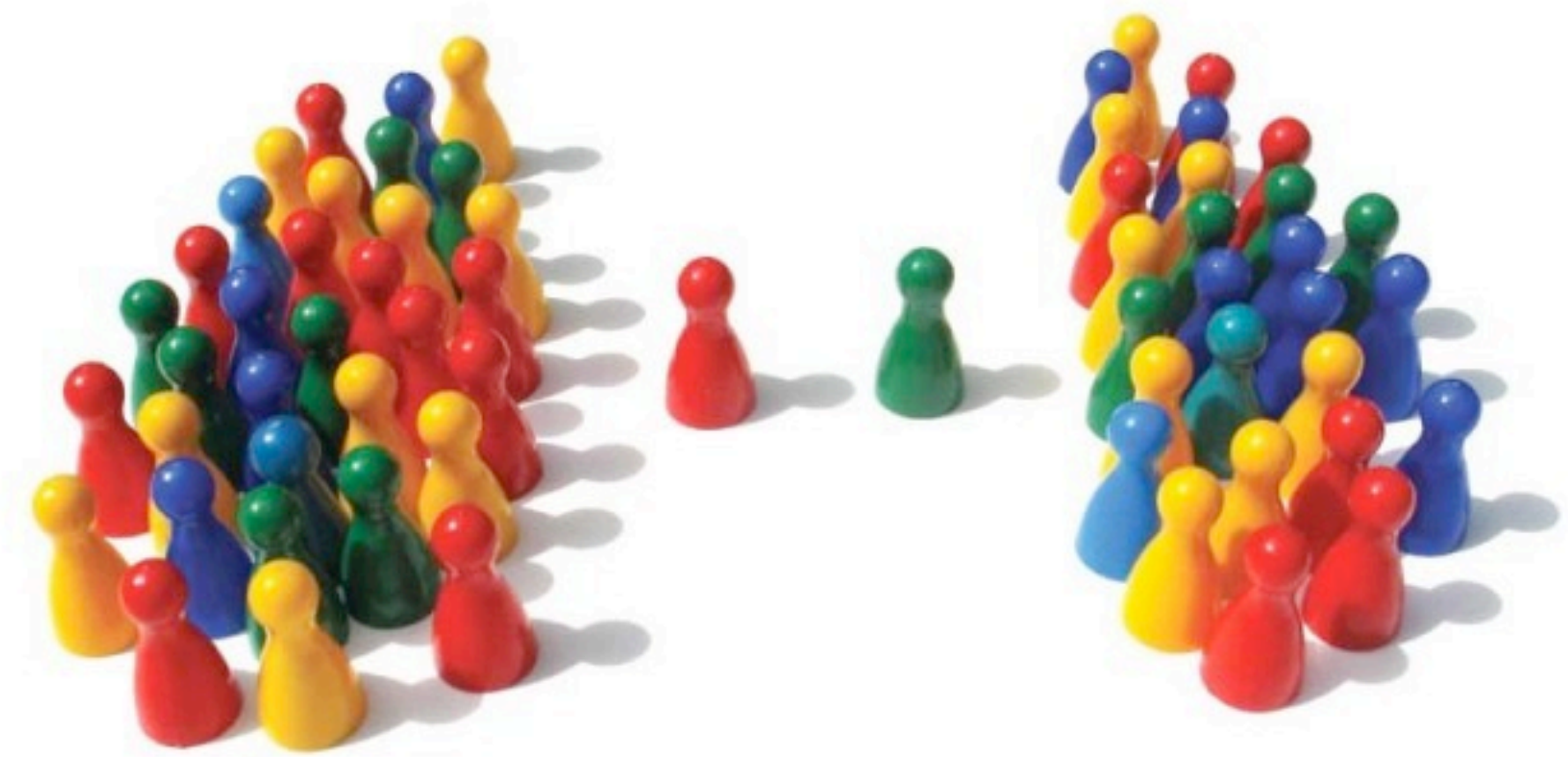
```
float brent(float ax, float bx, float cx, float (*f)(float), float tol,
    float *xmin)
```

Given a function `f`, and given a bracketing triplet of abscissas `ax`, `bx`, `cx` (such that `bx` is between `ax` and `cx`, and `f(bx)` is less than both `f(ax)` and `f(cx)`), this routine isolates the minimum to a fractional precision of about `tol` using Brent's method. The abscissa of the minimum is returned as `xmin`, and the minimum function value is returned as `brent`, the returned function value.

```
#include <math.h>
#define NRANSI
#include "nrutil.h"
#define ITMAX 100
#define CGOLD 0.3819660
#define ZEPS 1.0e-10
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

float brent(float ax, float bx, float cx, float (*f)(float), float tol,
	float *xmin)
{
	int iter;	float a,b,d,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
	float e=0.0;

	a=(ax < cx ? ax : cx);
	b=(ax > cx ? ax : cx);
	x=w=v=bx;
	fw=fv=fx=(*f)(x);
	for (iter=1;iter<=ITMAX;iter++) {
		xm=0.5*(a+b);
		tol2=2.0*(tol1=tol*fabs(x)+ZEPS);
		if (fabs(x-xm) <= (tol2-0.5*(b-a))) {
			*xmin=x;
			return fx;
		}
		if (fabs(e) > tol1) {
			r=(x-w)*(fx-fv);
			q=(x-v)*(fx-fw);
			p=(x-v)*q-(x-w)*r;
			q=2.0*(q-r);
			if (q > 0.0) p = -p;
			q=fabs(q);
			etemp=e;
			e=d;
			if (fabs(p) >= fabs(0.5*q*etemp) || p <= q*(a-x) || p >= q*(b-x))
				d=CGOLD*(e=(x >= xm ? a-x : b-x));
			else {
				d=p/q;
				u=x+d;
				if (u-a < tol2 || b-u < tol2)
					d=SIGN(tol1,xm-x);
			}
		} else {
			d=CGOLD*(e=(x >= xm ? a-x : b-x));
		}
		u=(fabs(d) >= tol1 ? x+d : x+SIGN(tol1,d));
		fu=(*f)(u);
		if (fu <= fx) {
			if (u >= x) a=x; else b=x;
			SHFT(v,w,x,u)
			SHFT(fv,fw,fx,fu)
		} else {
			if (u < x) a=u; else b=u;
			if (fu <= fw || w == x) {
				v=w;
				w=u;
				fv=fw;
				fw=fu;
			} else if (fu <= fv || v == x || v == w) {
				v=u;
				fv=fu;
			}
		}
	}
	nrerror("Too many iterations in brent");
	*xmin=x;
	return fx;
}
#undef ITMAX
#undef CGOLD
#undef ZEPS
#undef SHFT
#undef NRANSI
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

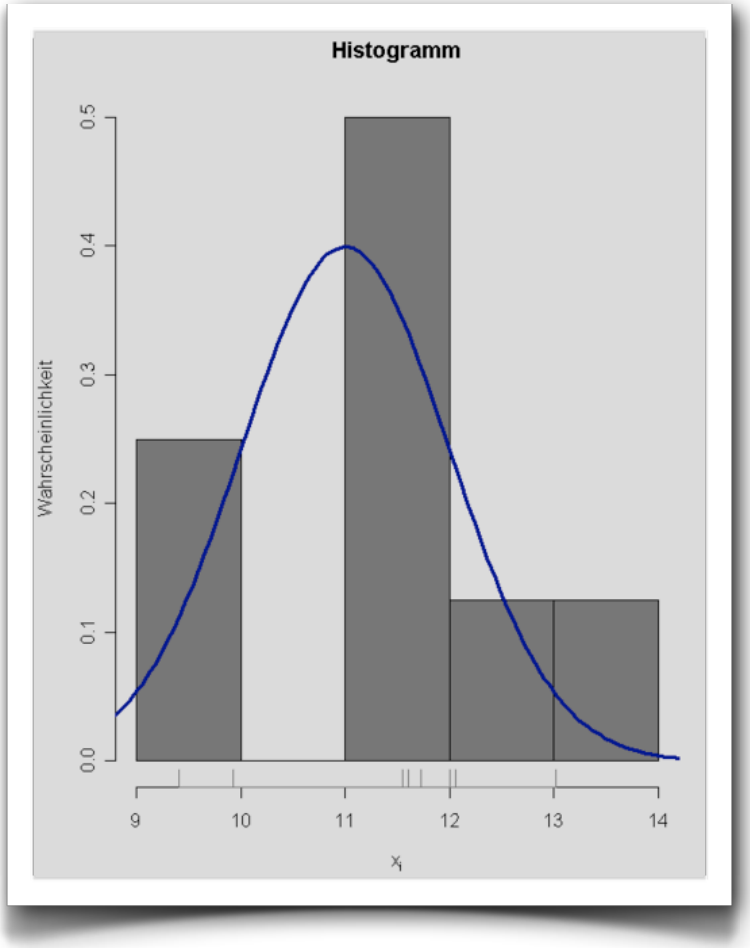
brent.c

Zijn twee populaties verschillend?



Chi-Square Test

Suppose that N_i is the number of events observed in the ith bin, and that n_i is the number expected according to some known distribution. Note that the N_i's are



integers, while the n_i's may not be. Then the chi-square statistic is

$$\chi^2 = \sum_i \frac{(N_i - n_i)^2}{n_i} \tag{14.3.1}$$

where the sum is over all bins. A large value of χ^2 indicates that the null hypothesis (that the N_i's are drawn from the population represented by the n_i's) is rather unlikely.

Any term j in (14.3.1) with $0 = n_j = N_j$ should be omitted from the sum. A term with $n_j = 0,\ N_j \neq 0$ gives an infinite χ^2, as it should, since in this case the N_i's cannot possibly be drawn from the n_i's!

The *chi-square probability function* $Q(\chi^2|\nu)$ is an incomplete gamma function, and was already discussed in §6.2 (see equation 6.2.18). Strictly speaking $Q(\chi^2|\nu)$ is the probability that the sum of the squares of ν random *normal* variables of unit variance (and zero mean) will be greater than χ^2. The terms in the sum (14.3.1) are not individually normal. However, if either the number of bins is large ($\gg 1$), or the number of events in each bin is large ($\gg 1$), then the chi-square probability function is a good approximation to the distribution of (14.3.1) in the case of the null hypothesis. Its use to estimate the significance of the chi-square test is standard.

The appropriate value of ν, the number of degrees of freedom, bears some additional discussion. If the data are collected with the model n_i's fixed — that is, not later renormalized to fit the total observed number of events ΣN_i — then ν equals the number of bins N_B. (Note that this is *not* the total number of *events*!) Much more commonly, the n_i's are normalized after the fact so that their sum equals the sum of the N_i's. In this case the correct value for ν is $N_B - 1$, and the model is said to have one constraint (`knstrn=1` in the program below). If the model that gives the n_i's has additional free parameters that were adjusted after the fact to agree with the data, then each of these additional "fitted" parameters decreases ν (and increases `knstrn`) by one additional unit.

Zijn twee populaties verschillend?

```
void chsone(float bins[], float ebins[], int nbins, int knstrn, float *df,
        float *chsq, float *prob)
{
        float gammq(float a, float x);
        void nrerror(char error_text[]);
        int j;
        float temp;

        *df=nbins-knstrn;
        *chsq=0.0;
        for (j=1;j<=nbins;j++) {
                if (ebins[j] <= 0.0) nrerror("Bad expected number in chsone");
                temp=bins[j]-ebins[j];
                *chsq += temp*temp/ebins[j];
        }
        *prob=gammq(0.5*(*df),0.5*(*chsq));
}
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

```
void chsone(float bins[], float ebins[], int nbins, int knstrn, float *df,
    float *chsq, float *prob)
```

Given the array `bins[1..nbins]` containing the observed numbers of events, and an array `ebins[1..nbins]` containing the expected numbers of events, and given the number of constraints `knstrn` (normally one), this routine returns (trivially) the number of degrees of freedom `df`, and (nontrivially) the chi-square `chsq` and the significance `prob`. A small value of `prob` indicates a significant difference between the distributions `bins` and `ebins`. Note that `bins` and `ebins` are both `float` arrays, although `bins` will normally contain integer values.

chsone.c

Zijn twee populaties verschillend?

Chi-Square Test

Numerical Recipes

Next we consider the case of comparing *two* binned data sets. Let R_i be the number of events in bin i for the first data set, S_i the number of events in the same bin i for the second data set. Then the chi-square statistic is

$$\chi^2 = \sum_i \frac{(R_i - S_i)^2}{R_i + S_i} \tag{14.3.2}$$

Comparing (14.3.2) to (14.3.1), you should note that the denominator of (14.3.2) is *not* just the average of R_i and S_i (which would be an estimator of n_i in 14.3.1). Rather, it is twice the average, the sum. The reason is that each term in a chi-square sum is supposed to approximate the square of a normally distributed quantity with unit variance. The variance of the difference of two normal quantities is the sum of their individual variances, not the average.

If the data were collected in such a way that the sum of the R_i's is necessarily equal to the sum of S_i's, then the number of degrees of freedom is equal to one less than the number of bins, $N_B - 1$ (that is, `knstrn` $= 1$), the usual case. If this requirement were absent, then the number of degrees of freedom would be N_B. Example: A birdwatcher wants to know whether the distribution of sighted birds as a function of species is the same this year as last. Each bin corresponds to one species. If the birdwatcher takes his data to be the first 1000 birds that he saw in each year, then the number of degrees of freedom is $N_B - 1$. If he takes his data to be all the birds he saw on a random sample of days, the same days in each year, then the number of degrees of freedom is N_B (`knstrn` $= 0$). In this latter case, note that he is also testing whether the birds were more numerous overall in one year or the other: That is the extra degree of freedom. Of course, any additional constraints on the data set lower the number of degrees of freedom (i.e., increase `knstrn` to *more positive* values) in accordance with their number.

Zijn twee populaties verschillend?

```
void chstwo(float bins1[], float bins2[], int nbins, int knstrn, float *df, float *chsq, float *prob)
{
        float gammq(float a, float x);
        int j;
        float temp;

        *df=nbins-knstrn;
        *chsq=0.0;
        for (j=1;j<=nbins;j++)
                if (bins1[j] == 0.0 && bins2[j] == 0.0)
                        --(*df);
                else {
                        temp=bins1[j]-bins2[j];
                        *chsq += temp*temp/(bins1[j]+bins2[j]);
                }
        *prob=gammq(0.5*(*df),0.5*(*chsq));
}
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

```
void chstwo(float bins1[], float bins2[], int nbins, int knstrn, float *df,
    float *chsq, float *prob)
```

Given the arrays `bins1[1..nbins]` and `bins2[1..nbins]`, containing two sets of binned data, and given the number of constraints `knstrn` (normally 1 or 0), this routine returns the number of degrees of freedom `df`, the chi-square `chsq`, and the significance `prob`. A small value of `prob` indicates a significant difference between the distributions `bins1` and `bins2`. Note that `bins1` and `bins2` are both `float` arrays, although they will normally contain integer values.

{

chstwo.c

Equation (14.3.2) and the routine `chstwo` both apply to the case where the total number of data points is the same in the two binned sets. For unequal numbers of data points, the formula analogous to (14.3.2) is

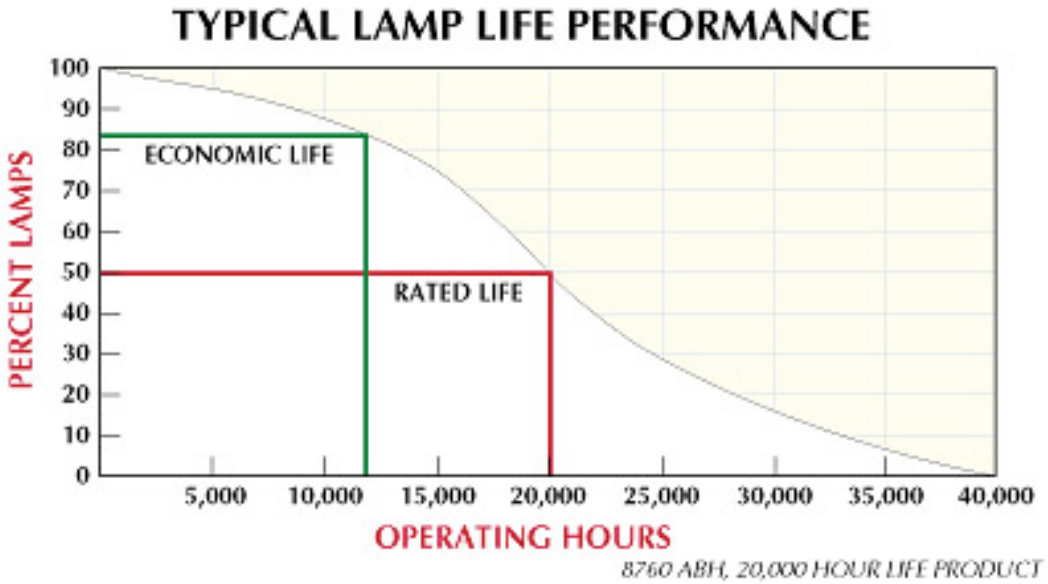
$$\chi^2 = \sum_i \frac{(\sqrt{S/R}R_i - \sqrt{R/S}S_i)^2}{R_i + S_i} \tag{14.3.3}$$

where

$$R \equiv \sum_i R_i \qquad S \equiv \sum_i S_i \tag{14.3.4}$$

are the respective numbers of data points. It is straightforward to make the corresponding change in `chstwo`.

Kolmogorov-Smirnov Test



Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov (or *K–S*) test is applicable to unbinned distributions that are functions of a single independent variable, that is, to data sets where each data point can be associated with a single number (lifetime of each lightbulb when it burns out, or declination of each star). In such cases, the list of data points can be easily converted to an unbiased estimator $S_N(x)$ of the *cumulative* distribution function of the probability distribution from which it was drawn: If the N events are located at values $x_i,\ i = 1, \ldots, N$, then $S_N(x)$ is the function giving the fraction of data points to the left of a given value x. This function is obviously constant between consecutive (i.e., sorted into ascending order) x_i's, and jumps by the same constant $1/N$ at each x_i. (See Figure 14.3.1.)

Different distribution functions, or sets of data, give different cumulative distribution function estimates by the above procedure. However, all cumulative distribution functions agree at the smallest allowable value of x (where they are zero), and at the largest allowable value of x (where they are unity). (The smallest and largest values might of course be $\pm\infty$.) So it is the behavior between the largest and smallest values that distinguishes distributions.

One can think of any number of statistics to measure the overall difference between two cumulative distribution functions: the absolute value of the area between them, for example. Or their integrated mean square difference. The Kolmogorov-Smirnov D is a particularly simple measure: It is defined as the *maximum value* of the absolute difference between two cumulative distribution functions. Thus, for comparing one data set's $S_N(x)$ to a known cumulative distribution function $P(x)$, the K–S statistic is

$$D = \max_{-\infty<x<\infty} |S_N(x) - P(x)| \tag{14.3.5}$$

while for comparing two different cumulative distribution functions $S_{N_1}(x)$ and $S_{N_2}(x)$, the K–S statistic is

$$D = \max_{-\infty<x<\infty} |S_{N_1}(x) - S_{N_2}(x)| \tag{14.3.6}$$

What makes the K–S statistic useful is that *its* distribution in the case of the null hypothesis (data sets drawn from the same distribution) can be calculated, at least to useful approximation, thus giving the significance of any observed nonzero value of D. A central feature of the K–S test is that it is invariant under reparametrization of x; in other words, you can locally slide or stretch the x axis in Figure 14.3.1, and the maximum distance D remains unchanged. For example, you will get the same significance using x as using $\log x$.

The function that enters into the calculation of the significance can be written as the following sum:

$$Q_{KS}(\lambda) = 2\sum_{j=1}^{\infty}(-1)^{j-1}\, e^{-2j^2\lambda^2} \tag{14.3.7}$$

which is a monotonic function with the limiting values

$$Q_{KS}(0) = 1 \qquad Q_{KS}(\infty) = 0 \tag{14.3.8}$$

In terms of this function, the significance level of an observed value of D (as a disproof of the null hypothesis that the distributions are the same) is given approximately [1] by the formula

$$\text{Probability}\ (D > \text{observed}\ ) = Q_{KS}\left(\left[\sqrt{N_e} + 0.12 + 0.11/\sqrt{N_e}\right]\ D\right) \tag{14.3.9}$$

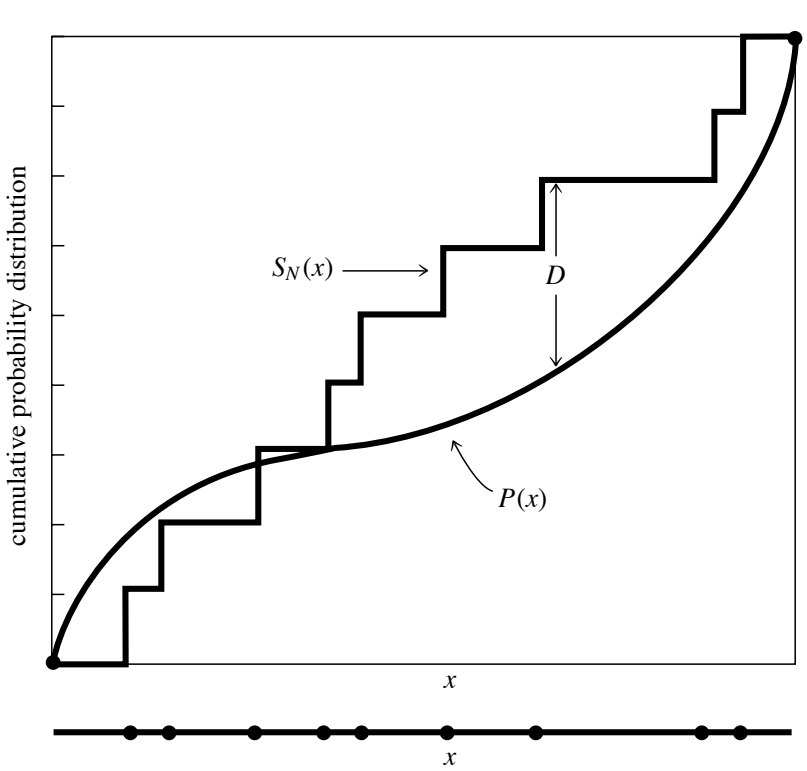


Figure 14.3.1. Kolmogorov-Smirnov statistic D. A measured distribution of values in x (shown as N dots on the lower abscissa) is to be compared with a theoretical distribution whose cumulative probability distribution is plotted as $P(x)$. A step-function cumulative probability distribution $S_N(x)$ is constructed, one that rises an equal amount at each measured point. D is the greatest distance between the two cumulative distributions.

where N_e is the effective number of data points, $N_e = N$ for the case (14.3.5) of one distribution, and

$$N_e = \frac{N_1 N_2}{N_1 + N_2} \tag{14.3.10}$$

for the case (14.3.6) of two distributions, where N_1 is the number of data points in the first distribution, N_2 the number in the second.

The nature of the approximation involved in (14.3.9) is that it becomes asymptotically accurate as the N_e becomes large, but is already quite good for $N_e \geq 4$, as small a number as one might ever actually use. (See [1].)

```
#include <math.h>
#define NRANSI
#include "nrutil.h"

void ksone(float data[], unsigned long n, float (*func)(float), float *d, float *prob)
{
        float probks(float alam);
        void sort(unsigned long n, float arr[]);
        unsigned long j;
        float dt,en,ff,fn,fo=0.0;

        sort(n,data);
        en=n;
        *d=0.0;
        for (j=1;j<=n;j++) {
                fn=j/en;
                ff=(*func)(data[j]);
                dt=FMAX(fabs(fo-ff),fabs(fn-ff));
                if (dt > *d) *d=dt;
                fo=fn;
        }
        en=sqrt(en);
        *prob=probks((en+0.12+0.11/en)*(*d));
}
#undef NRANSI
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

```
void ksone(float data[], unsigned long n, float (*func)(float), float *d,
    float *prob)
```

Given an array `data[1..n]`, and given a user-supplied function of a single variable `func` which is a cumulative distribution function ranging from 0 (for smallest values of its argument) to 1 (for largest values of its argument), this routine returns the K–S statistic `d`, and the significance level `prob`. Small values of `prob` show that the cumulative distribution function of `data` is significantly different from `func`. The array `data` is modified by being sorted into ascending order.

ksone.c

```
#include <math.h>
#define EPS1 0.001
#define EPS2 1.0e-8

float probks(float alam)
{
        int j;
        float a2,fac=2.0,sum=0.0,term,termbf=0.0;

        a2 = -2.0*alam*alam;
        for (j=1;j<=100;j++) {
                term=fac*exp(a2*j*j);
                sum += term;
                if (fabs(term) <= EPS1*termbf || fabs(term) <= EPS2*sum) return sum;
                fac = -fac;
                termbf=fabs(term);
        }
        return 1.0;
}
#undef EPS1
#undef EPS2
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

The function that enters into the calculation of the significance can be written as the following sum:

$$Q_{KS}(\lambda) = 2\sum_{j=1}^{\infty}(-1)^{j-1}\, e^{-2j^2\lambda^2} \qquad (14.3.7)$$

which is a monotonic function with the limiting values

$$Q_{KS}(0) = 1 \qquad Q_{KS}(\infty) = 0 \qquad (14.3.8)$$

`float probks(float alam)`
Kolmogorov-Smirnov probability function.

probks.c