Wetenschappelijk Programmeren (Voortzetting)

7.6 Simple Monte Carlo Integration

Inspirations for numerical methods can spring from unlikely sources. "Splines" first were flexible strips of wood used by draftsmen. "Simulated annealing" (we shall see in §10.9) is rooted in a thermodynamic analogy. And who does not feel at least a faint echo of glamor in the name "Monte Carlo method"?

Suppose that we pick N random points, uniformly distributed in a multidimensional volume V. Call them $x_1, \ldots, x_N$. Then the basic theorem of Monte Carlo integration estimates the integral of a function f over the multidimensional volume,

$$\int f \, dV \approx V \langle f \rangle \pm V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \tag{7.6.1}$$

Here the angle brackets denote taking the arithmetic mean over the N sample points,

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f(x_i) \qquad \langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f^2(x_i) \tag{7.6.2}$$

The "plus-or-minus" term in (7.6.1) is a one standard deviation error estimate for the integral, not a rigorous bound; further, there is no guarantee that the error is distributed as a Gaussian, so the error term should be taken only as a rough indication of probable error.

Suppose that you want to integrate a function g over a region W that is not easy to sample randomly. For example, W might have a very complicated shape. No problem. Just find a region V that *includes* W and that *can* easily be sampled (Figure 7.6.1), and then define f to be equal to g for points in W and equal to zero for points outside of W (but still inside the sampled V). You want to try to make V enclose W as closely as possible, because the zero values of f will increase the error estimate term of (7.6.1). And well they should: points chosen outside of W have no information content, so the effective value of N, the number of points, is reduced. The error estimate in (7.6.1) takes this into account.

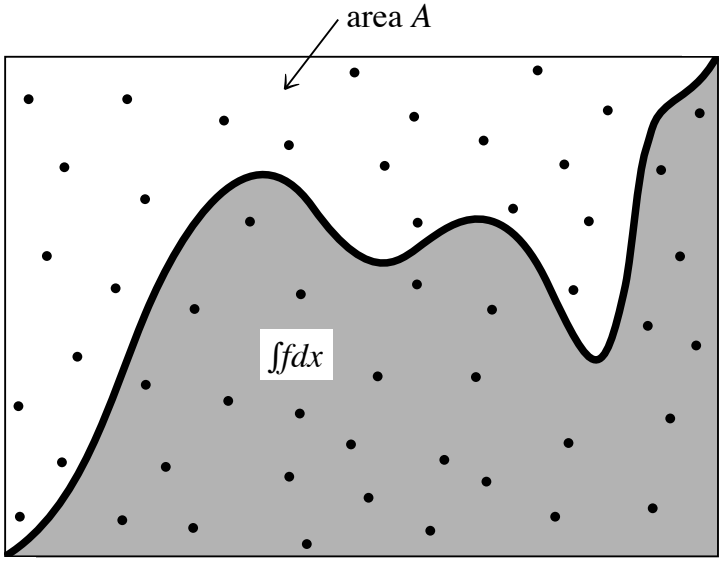


Figure 7.6.1. Monte Carlo integration. Random points are chosen within the area A. The integral of the function f is estimated as the area of A multiplied by the fraction of random points that fall below the curve f. Refinements on this procedure can improve the accuracy of the method; see text.

General purpose routines for Monte Carlo integration are quite complicated (see §7.8), but a worked example will show the underlying simplicity of the method. Suppose that we want to find the weight and the position of the center of mass of an object of complicated shape, namely the intersection of a torus with the edge of a large box. In particular let the object be defined by the three simultaneous conditions

$$z^2 + \left(\sqrt{x^2+y^2} - 3\right)^2 \le 1 \tag{7.6.3}$$

(torus centered on the origin with major radius $= 4$, minor radius $= 2$)

$$x \ge 1 \qquad y \ge -3 \tag{7.6.4}$$

(two faces of the box, see Figure 7.6.2). Suppose for the moment that the object has a constant density ρ.

We want to estimate the following integrals over the interior of the complicated object:

$$\int \rho \, dx\, dy\, dz \qquad \int x\rho \, dx\, dy\, dz \qquad \int y\rho \, dx\, dy\, dz \qquad \int z\rho \, dx\, dy\, dz \tag{7.6.5}$$

The coordinates of the center of mass will be the ratio of the latter three integrals (linear moments) to the first one (the weight).

In the following fragment, the region V, enclosing the piece-of-torus W, is the rectangular box extending from 1 to 4 in x, -3 to 4 in y, and -1 to 1 in z.

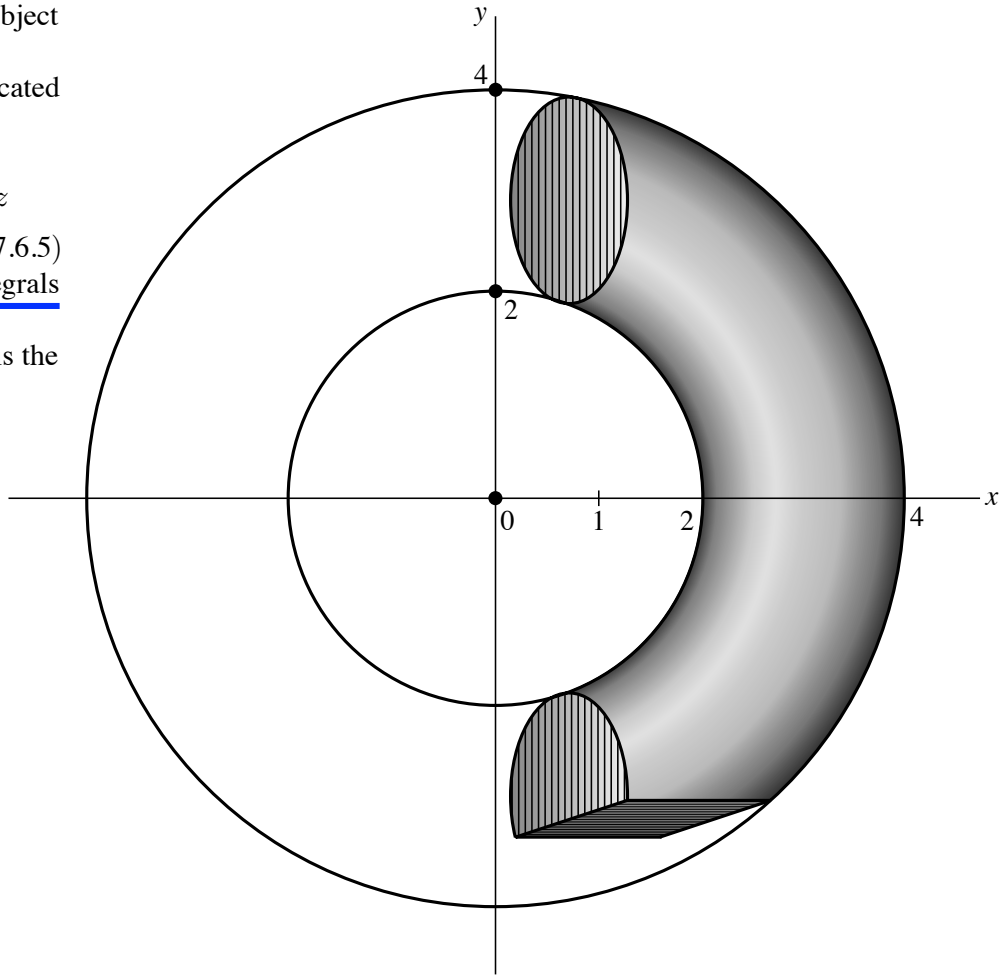


Figure 7.6.2. Example of Monte Carlo integration (see text). The region of interest is a piece of a torus, bounded by the intersection of two planes. The limits of integration of the region cannot easily be written in analytically closed form, so Monte Carlo is a useful technique.

```
#include<stdio.h>
#include<stdlib.h>#include<math.h>#include "nrutil.h"

int main()
 {  int j,idum,n=500000;
  float den=0.1;
  float sw,swx,swy,swz;
  float varw,varx,vary,varz;
  float vol,w;  float x,y,z,dw,dx,dy,dz;
sw=swx=swy=swz=0.0;

  varw=varx=vary=varz=0.0;
  vol=3.0*7.0*2.0;    // volume of the sample region

  for(j=1;j<=n;j++)
   {
     x=1.0+3.0   *(float)rand()/RAND_MAX;
     y=(-3.0)+7.0*(float)rand()/RAND_MAX;
     z=(-1.0)+2.0*(float)rand()/RAND_MAX;
     if (z*z+SQR(sqrt(x*x+y*y)-3.0) < 1.0)      // Is it in the
torus?
      {
        // If so, add to the various cumulants.
        sw += den;
        swx += x*den;
        swy += y*den;
        swz += z*den;
        varw += SQR(den);
        varx += SQR(x*den);
        vary += SQR(y*den);
        varz += SQR(z*den);
      }
    }
```

```
    // The values of the integrals (7.6.5),
    w=vol*sw/n;
    x=vol*swx/n;
    y=vol*swy/n;
    z=vol*swz/n;
    // and their corresponding error estimates.
    dw=vol*sqrt((varw/n-SQR(sw/n))/n);
    dx=vol*sqrt((varx/n-SQR(swx/n))/n);
    dy=vol*sqrt((vary/n-SQR(swy/n))/n);
    dz=vol*sqrt((varz/n-SQR(swz/n))/n);

   printf("center of gravity\n x=%12.4f +/-%12.4f\n y=%12.4f +/-%12.4f\n z=%12.4f +/-
%12.4f\n",
          x/w, y/w, z/w, dx/w, dy/w, dz/w);
 }
```

torus.c

7.7 Quasi- (that is, Sub-) Random Sequences

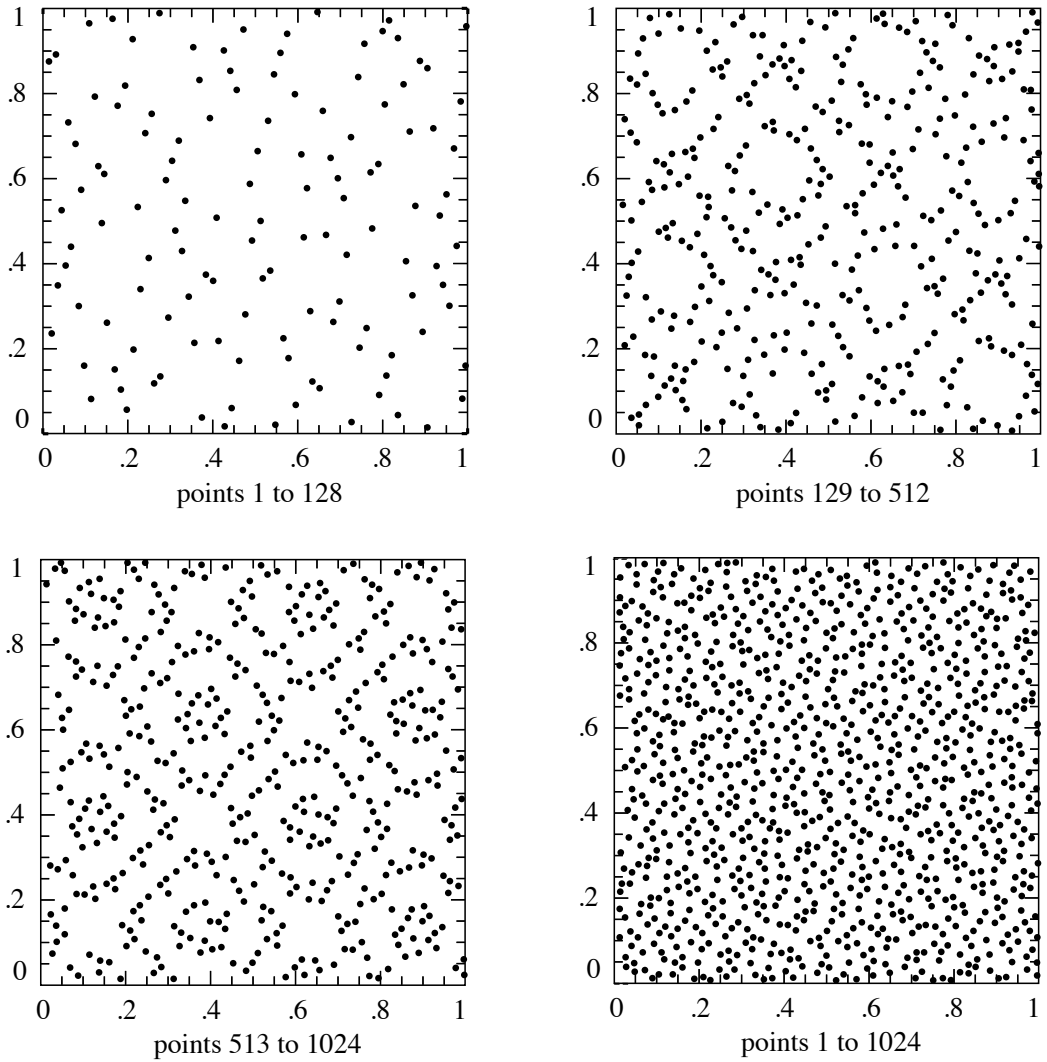


Figure 7.7.1. First 1024 points of a two-dimensional Sobol' sequence. The sequence is generated number-theoretically, rather than randomly, so successive points at any stage "know" how to fill in the gaps in the previously generated distribution.

```
#define NRANSI
#include "nrutil.h"
#define MAXBIT 30
#define MAXDIM 6

void sobseq(int *n, float x[])
{
	int j,k,l;
	unsigned long i,im,ipp;
	static float fac;
	static unsigned long in,ix[MAXDIM+1],*iu[MAXBIT+1];
	static unsigned long mdeg[MAXDIM+1]={0,1,2,3,3,4,4};
	static unsigned long ip[MAXDIM+1]={0,0,1,1,2,1,4};
	static unsigned long iv[MAXDIM*MAXBIT+1]={
		0,1,1,1,1,1,1,3,1,3,3,1,1,5,7,7,3,3,5,15,11,5,15,13,9};

	if (*n < 0) {
		for (j=1,k=0;j<=MAXBIT;j++,k+=MAXDIM) iu[j] = &iv[k];
		for (k=1;k<=MAXDIM;k++) {
			for (j=1;j<=mdeg[k];j++) iu[j][k] <<= (MAXBIT-j);
			for (j=mdeg[k]+1;j<=MAXBIT;j++) {
				ipp=ip[k];
				i=iu[j-mdeg[k]][k];
				i ^= (i >> mdeg[k]);
				for (l=mdeg[k]-1;l>=1;l--) {
					if (ipp & 1) i ^= iu[j-l][k];
					ipp >>= 1;
				}
				iu[j][k]=i;
			}
		}
		fac=1.0/(1L << MAXBIT);
		in=0;
	}
	else {
		im=in;
		for (j=1;j<=MAXBIT;j++) {
			if (!(im & 1)) break;
			im >>= 1;
		}
		if (j > MAXBIT) nrerror("MAXBIT too small
in sobseq");
		im=(j-1)*MAXDIM;
		for (k=1;k<=IMIN(*n,MAXDIM);k++) {
			ix[k] ^= iv[im+k];
			x[k]=ix[k]*fac;
		}
		in++;
	}
}
#undef MAXBIT
#undef MAXDIM
#undef NRANSI
/* (C) Copr. 1986-92 Numerical Recipes Software ?
421.1-9. */
```

`void sobseq(int *n, float x[])`
When `n` is negative, internally initializes a set of `MAXBIT` direction numbers for each of `MAXDIM` different Sobol' sequences. When `n` is positive (but $\leq$`MAXDIM`), returns as the vector `x[1..n]` the next values from `n` of these sequences. (`n` must not be changed between initializations.)

sobseq.c

How good is a Sobol' sequence, anyway? For Monte Carlo integration of a smooth function in n dimensions, the answer is that the fractional error will decrease with N, the number of samples, as $(\ln N)^n/N$, i.e., almost as fast as $1/N$. As an example, let us integrate a function that is nonzero inside a torus (doughnut) in three-dimensional space. If the major radius of the torus is R_0, the minor radial coordinate r is defined by

$$r = \left([(x^2 + y^2)^{1/2} - R_0]^2 + z^2 \right)^{1/2} \tag{7.7.4}$$

Let us try the function

$$f(x,y,z) = \begin{cases} 1 + \cos\left(\dfrac{\pi r^2}{a^2}\right) & r < r_0 \\ 0 & r \geq r_0 \end{cases} \tag{7.7.5}$$

which can be integrated analytically in cylindrical coordinates, giving

$$\int\int\int dx\, dy\, dz\, f(x,y,z) = 2\pi^2 a^2 R_0 \tag{7.7.6}$$

With parameters $R_0 = 0.6$, $r_0 = 0.3$, we did 100 successive Monte Carlo integrations of equation (7.7.4), sampling uniformly in the region $-1 < x, y, z < 1$, for the two cases of uncorrelated random points and the Sobol' sequence generated by the routine `sobseq`. Figure 7.7.2 shows the results, plotting the r.m.s. average error of the 100 integrations as a function of the number of points sampled. (For any *single* integration, the error of course wanders from positive to negative, or vice versa, so a logarithmic plot of fractional error is not very informative.) The thin, dashed curve corresponds to uncorrelated random points and shows the familiar $N^{-1/2}$ asymptotics. The thin, solid gray curve shows the result for the Sobol' sequence. The logarithmic term in the expected $(\ln N)^3/N$ is readily apparent as curvature in the curve, but the asymptotic N^{-1} is unmistakable.

To understand the importance of Figure 7.7.2, suppose that a Monte Carlo integration of f with 1% accuracy is desired. The Sobol' sequence achieves this accuracy in a few thousand samples, while pseudorandom sampling requires nearly 100,000 samples. The ratio would be even greater for higher desired accuracies.

A different, not quite so favorable, case occurs when the function being integrated has hard (discontinuous) boundaries inside the sampling region, for example the function that is one inside the torus, zero outside,

$$f(x,y,z) = \begin{cases} 1 & r < r_0 \\ 0 & r \geq r_0 \end{cases} \tag{7.7.7}$$

where r is defined in equation (7.7.4). Not by coincidence, this function has the same analytic integral as the function of equation (7.7.5), namely $2\pi^2 a^2 R_0$.

The carefully hierarchical Sobol' sequence is based on a set of Cartesian grids, but the boundary of the torus has no particular relation to those grids. The result is that it is essentially random whether sampled points in a thin layer at the surface of the torus, containing on the order of $N^{2/3}$ points, come out to be inside, or outside, the torus. The square root law, applied to this thin layer, gives $N^{1/3}$ fluctuations in the sum, or $N^{-2/3}$ fractional error in the Monte Carlo integral. One sees this behavior verified in Figure 7.7.2 by the thicker gray curve. The thicker dashed curve in Figure 7.7.2 is the result of integrating the function of equation (7.7.7) using independent random points. While the advantage of the Sobol' sequence is not quite so dramatic as in the case of a smooth function, it can nonetheless be a significant factor ($\sim$5) even at modest accuracies like 1%, and greater at higher accuracies.

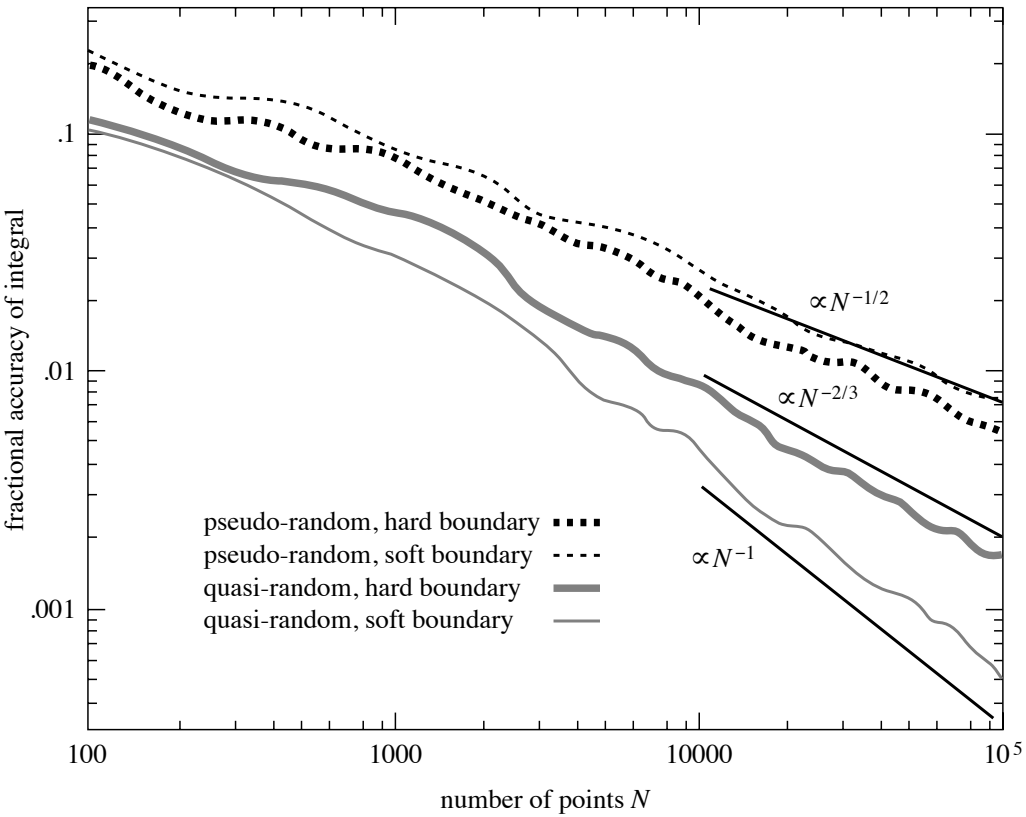


Figure 7.7.2. Fractional accuracy of Monte Carlo integrations as a function of number of points sampled, for two different integrands and two different methods of choosing random points. The quasi-random Sobol' sequence converges much more rapidly than a conventional pseudo-random sequence. Quasi-random sampling does better when the integrand is smooth ("soft boundary") than when it has step discontinuities ("hard boundary"). The curves shown are the r.m.s. average of 100 trials.

root finding

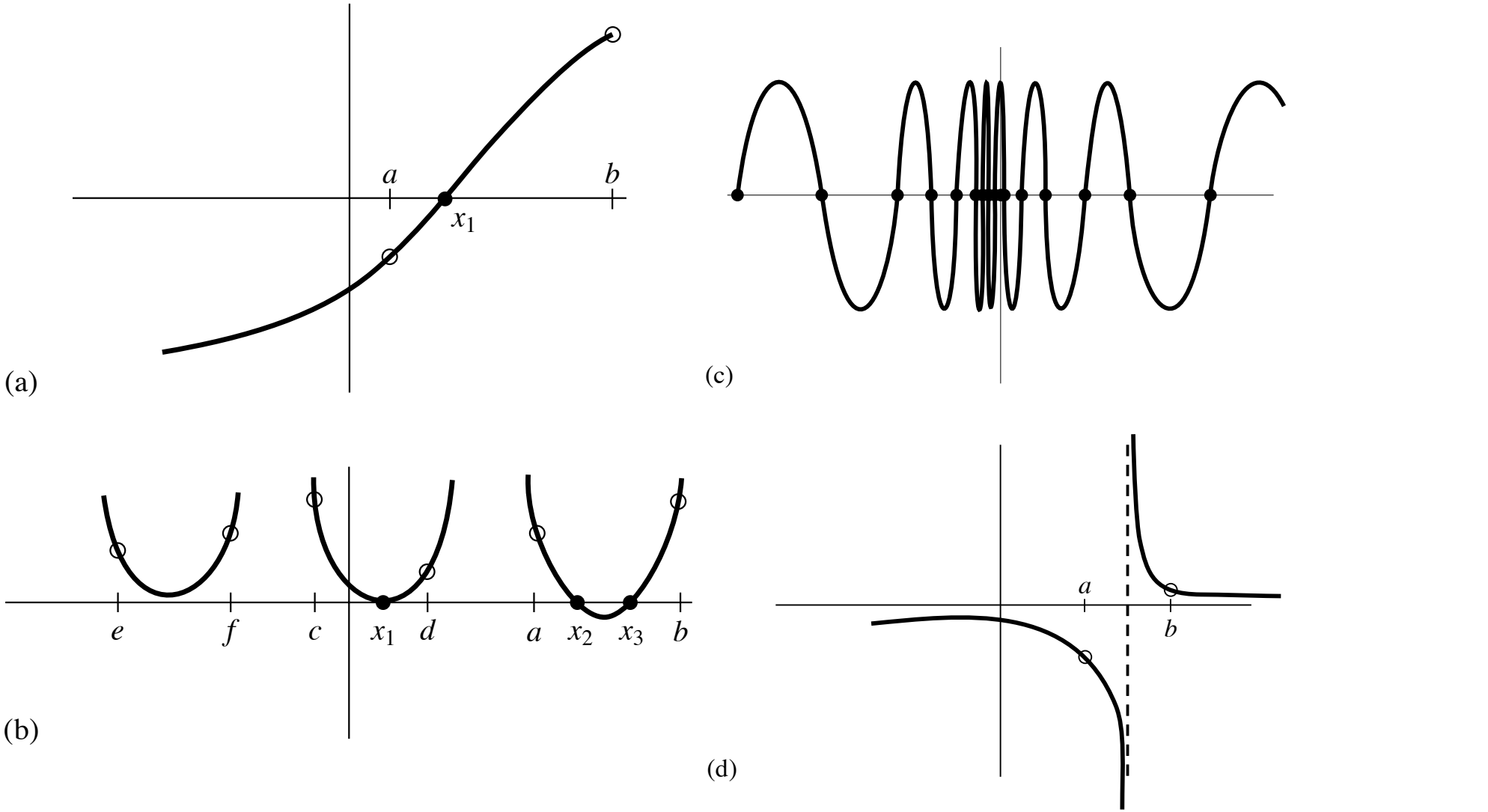


Figure 9.1.1. Some situations encountered while root finding: (a) shows an isolated root x_1 bracketed by two points a and b at which the function has opposite signs; (b) illustrates that there is not necessarily a sign change in the function near a double root (in fact, there is not necessarily a root!); (c) is a pathological function with many roots; in (d) the function has opposite signs at points a and b, but the points bracket a singularity, not a root.

```
#include <math.h>
#define FACTOR 1.6
#define NTRY 50

int zbrac(float (*func)(float), float *x1, float *x2)
{
        void nrerror(char error_text[]);
        int j;
        float f1,f2;

        if (*x1 == *x2) nrerror("Bad initial range in zbrac");
        f1=(*func)(*x1);
        f2=(*func)(*x2);
        for (j=1;j<=NTRY;j++) {
                if (f1*f2 < 0.0) return 1;
                if (fabs(f1) < fabs(f2))
                        f1=(*func)(*x1 += FACTOR*(*x1-*x2));
                else
                        f2=(*func)(*x2 += FACTOR*(*x2-*x1));
        }
        return 0;
}
#undef FACTOR
#undef NTRY
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

`int zbrac(float (*func)(float), float *x1, float *x2)`
Given a function `func` and an initial guessed range `x1` to `x2`, the routine expands the range geometrically until a root is bracketed by the returned values `x1` and `x2` (in which case `zbrac` returns 1) or until the range becomes unacceptably large (in which case `zbrac` returns 0).
{

zbrac.c

```
void zbrak(float (*fx)(float), float x1, float x2, int n, float xb1[],
        float xb2[], int *nb)
{
        int nbb,i;
        float x,fp,fc,dx;

        nbb=0;
        dx=(x2-x1)/n;
        fp=(*fx)(x=x1);
        for (i=1;i<=n;i++) {
                fc=(*fx)(x += dx);
                if (fc*fp < 0.0) {
                        xb1[++nbb]=x-dx;
                        xb2[nbb]=x;
                        if(*nb == nbb) return;

                }
                fp=fc;
        }
        *nb = nbb;
}
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

```
void zbrak(float (*fx)(float), float x1, float x2, int n, float xb1[],
    float xb2[], int *nb)
```

Given a function `fx` defined on the interval from `x1`–`x2` subdivide the interval into `n` equally spaced segments, and search for zero crossings of the function. `nb` is input as the maximum number of roots sought, and is reset to the number of bracketing pairs `xb1[1..nb]`, `xb2[1..nb]` that are found.

zbrak.c

bisection method

```
#include <math.h>
#define JMAX 40

float rtbis(float (*func)(float), float x1, float x2, float xacc)
{
        void nrerror(char error_text[]);
        int j;
        float dx,f,fmid,xmid,rtb;

        f=(*func)(x1);
        fmid=(*func)(x2);
        if (f*fmid >= 0.0) nrerror("Root must be bracketed for bisection in rtbis");
        rtb = f < 0.0 ? (dx=x2-x1,x1) : (dx=x1-x2,x2);
        for (j=1;j<=JMAX;j++) {
                fmid=(*func)(xmid=rtb+(dx *= 0.5));
                if (fmid <= 0.0) rtb=xmid;
                if (fabs(dx) < xacc || fmid == 0.0) return rtb;
        }
        nrerror("Too many bisections in rtbis");
        return 0.0;
}
#undef JMAX
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

`float rtbis(float (*func)(float), float x1, float x2, float xacc)`
Using bisection, find the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `rtbis`, will be refined until its accuracy is $\pm$`xacc`.

rtbis.c

```
#include<stdio.h>
#define NN 20

#include "nrutil.c"
#include "zbrac.c"
#include "zbrak.c"
#include "rtbis.c"

float function(float x)
 {
  return(sin(x));
 }

int main()
 {
  int n=10,nb=NN;
  float x1=-1., x2=1.,res;
  float xb1[NN+1], xb2[NN+1];
  float xacc=0.000001;

  res= zbrac( (float(*)(float))function,&x1,&x2);
  if(!res) printf("zbrac: range unacceptably large\n");
   else    printf("zbrac: zero crossing between %f and %f \n\n",x1,x2);

  x1=-10.; x2=11;
  zbrak( (float(*)(float))function, x1, x2, n, xb1, xb2, &nb);
  for(n=1; n<=nb; n++)
   {
    printf("zbrak: zero crossing found [%f,%f]\n",xb1[n],xb2[n]);
    printf("       rtbis: zero found at %f\n",
      rtbis( (float(*)(float))function, xb1[n],xb2[n],xacc));
   }
 }
```

zero.c

9.4 Newton-Raphson Method Using Derivative

Perhaps the most celebrated of all one-dimensional root-finding routines is *Newton's method*, also called the *Newton-Raphson method*. This method is distinguished from the methods of previous sections by the fact that it requires the evaluation of both the function $f(x)$, *and* the derivative $f'(x)$, at arbitrary points x. The Newton-Raphson formula consists geometrically of extending the tangent line at a current point x_i until it crosses zero, then setting the next guess x_{i+1} to the abscissa of that zero-crossing (see Figure 9.4.1). Algebraically, the method derives from the familiar Taylor series expansion of a function in the neighborhood of a point,

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \ldots . \tag{9.4.1}$$

For small enough values of δ, and for well-behaved functions, the terms beyond linear are unimportant, hence $f(x+\delta) = 0$ implies

$$\delta = -\frac{f(x)}{f'(x)}. \tag{9.4.2}$$

Newton-Raphson is not restricted to one dimension. The method readily generalizes to multiple dimensions, as we shall see in §9.6 and §9.7, below.

Far from a root, where the higher-order terms in the series *are* important, the Newton-Raphson formula can give grossly inaccurate, meaningless corrections. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. This can be death to the method (see Figure 9.4.2). If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson sends its solution off to limbo, with vanishingly small hope of recovery.

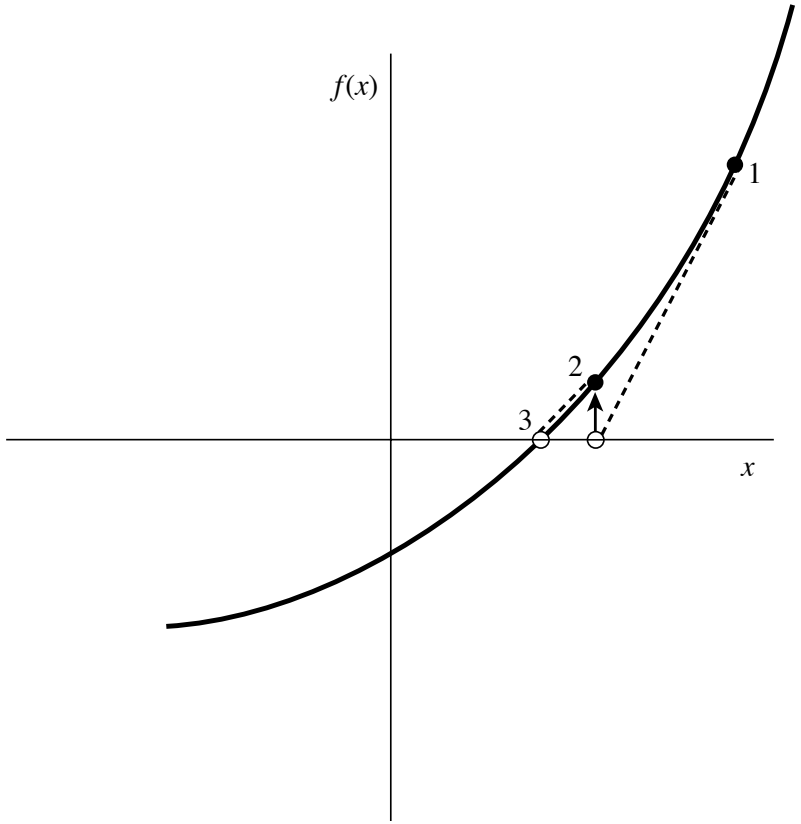


Figure 9.4.1. Newton's method extrapolates the local derivative to find the next estimate of the root. In this example it works well and converges quadratically.

Like most powerful tools, Newton-Raphson can be destructive used in inappropriate circumstances. Figure 9.4.3 demonstrates another possible pathology.

Why do we call Newton-Raphson powerful? The answer lies in its rate of convergence: Within a small distance ϵ of x the function and its derivative are approximately:

$$\begin{aligned} f(x+\epsilon) &= f(x) + \epsilon f'(x) + \epsilon^2 \frac{f''(x)}{2} + \cdots, \\ f'(x+\epsilon) &= f'(x) + \epsilon f''(x) + \cdots \end{aligned} \tag{9.4.3}$$

By the Newton-Raphson formula,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \tag{9.4.4}$$

so that

$$\epsilon_{i+1} = \epsilon_i - \frac{f(x_i)}{f'(x_i)}. \tag{9.4.5}$$

When a trial solution x_i differs from the true root by ϵ_i, we can use (9.4.3) to express $f(x_i), f'(x_i)$ in (9.4.4) in terms of ϵ_i and derivatives at the root itself. The result is a recurrence relation for the deviations of the trial solutions

$$\epsilon_{i+1} = -\epsilon_i^2 \frac{f''(x)}{2f'(x)}. \tag{9.4.6}$$

Equation (9.4.6) says that Newton-Raphson converges *quadratically* (cf. equation 9.2.3). Near a root, the number of significant digits approximately *doubles* with each step. This very strong convergence property makes Newton-Raphson the method of choice for any function whose derivative can be evaluated efficiently, and whose derivative is continuous and nonzero in the neighborhood of a root.

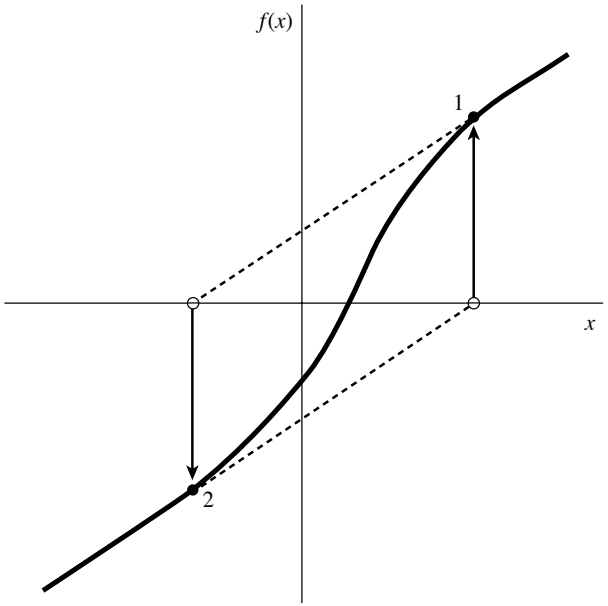


Figure 9.4.3. Unfortunate case where Newton's method enters a nonconvergent cycle. This behavior is often encountered when the function f is obtained, in whole or in part, by table interpolation. With a better initial guess, the method would have succeeded.

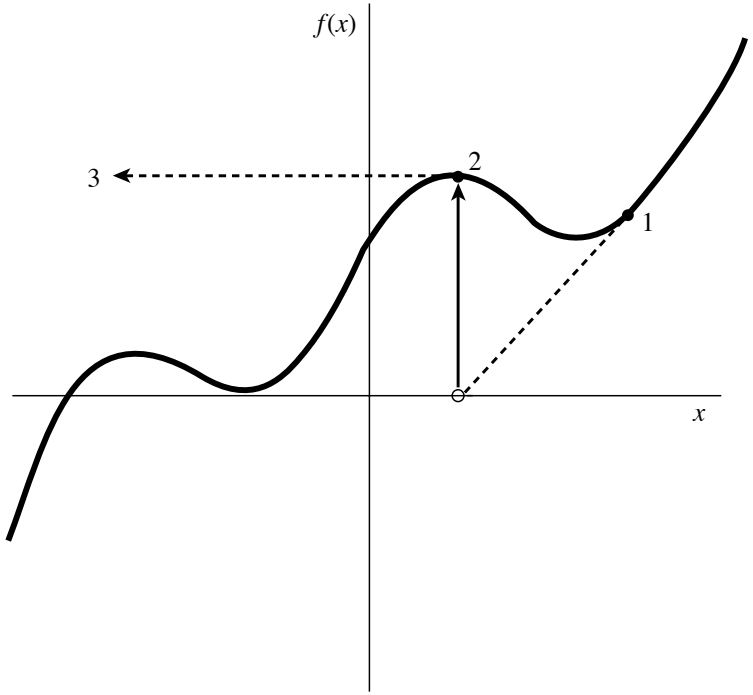


Figure 9.4.2. Unfortunate case where Newton's method encounters a local extremum and shoots off to outer space. Here bracketing bounds, as in `rtsafe`, would save the day.

Newton method

```
#include <math.h>
#define JMAX 20

float rtnewt(void (*funcd)(float, float *, float *), float x1, float x2,
        float xacc)
{
        void nrerror(char error_text[]);
        int j;
        float df,dx,f,rtn;

        rtn=0.5*(x1+x2);
        for (j=1;j<=JMAX;j++) {
                (*funcd)(rtn,&f,&df);
                dx=f/df;
                rtn -= dx;
                if ((x1-rtn)*(rtn-x2) < 0.0)
                        nrerror("Jumped out of brackets in rtnewt");
                if (fabs(dx) < xacc) return rtn;
        }
        nrerror("Maximum number of iterations exceeded in rtnewt");
        return 0.0;
}
#undef JMAX
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

rtnewt.c

```
float rtnewt(void (*funcd)(float, float *, float *), float x1, float x2,
    float xacc)
```

Using the Newton-Raphson method, find the root of a function known to lie in the interval [x1, x2]. The root `rtnewt` will be refined until its accuracy is known within ±`xacc`. `funcd` is a user-supplied routine that returns both the function value and the first derivative of the function at the point `x`.

```
#include<stdio.h>
#include<math.h>

#include "nrutil.c"
#include "rtnewt.c"

void function(float x, float *y, float *dy)
 {
  *y=sin(x);
  *dy=cos(x);
 }

int main()
 {
  float x1=1;
  float x2=4;
  float xacc=0.001;

  printf("zero found at x=%f\n",
    rtnewt( (void(*)(float,float*,float*))function,x1,x2,xacc));
 }
```

zeronewton.c

```
#include <math.h>
#define MAXIT 100

float rtsafe(void (*funcd)(float, float *, float *), float x1, float x2,
        float xacc)
{
        void nrerror(char error_text[]);
        int j;
        float df,dx,dxold,f,fh,fl;
        float temp,xh,xl,rts;

        (*funcd)(x1,&fl,&df);
        (*funcd)(x2,&fh,&df);
        if ((fl > 0.0 && fh > 0.0) || (fl < 0.0 && fh < 0.0))
                nrerror("Root must be bracketed in rtsafe");
        if (fl == 0.0) return x1;
        if (fh == 0.0) return x2;
        if (fl < 0.0) {
                xl=x1;
                xh=x2;
        } else {
                xh=x1;
                xl=x2;
        }
        rts=0.5*(x1+x2);
        dxold=fabs(x2-x1);
        dx=dxold;
        (*funcd)(rts,&f,&df);
        for (j=1;j<=MAXIT;j++) {
                if ((((rts-xh)*df-f)*((rts-xl)*df-f) >= 0.0)
                        || (fabs(2.0*f) > fabs(dxold*df))) {
                        dxold=dx;
                        dx=0.5*(xh-xl);
                        rts=xl+dx;
                        if (xl == rts) return rts;
                } else {
                        dxold=dx;
                        dx=f/df;
                        temp=rts;
                        rts -= dx;
                        if (temp == rts) return rts;
                }
                if (fabs(dx) < xacc) return rts;
                (*funcd)(rts,&f,&df);
                if (f < 0.0)
                        xl=rts;
                else
                        xh=rts;
        }
        nrerror("Maximum number of iterations exceeded in rtsafe");
        return 0.0;
}
#undef MAXIT
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

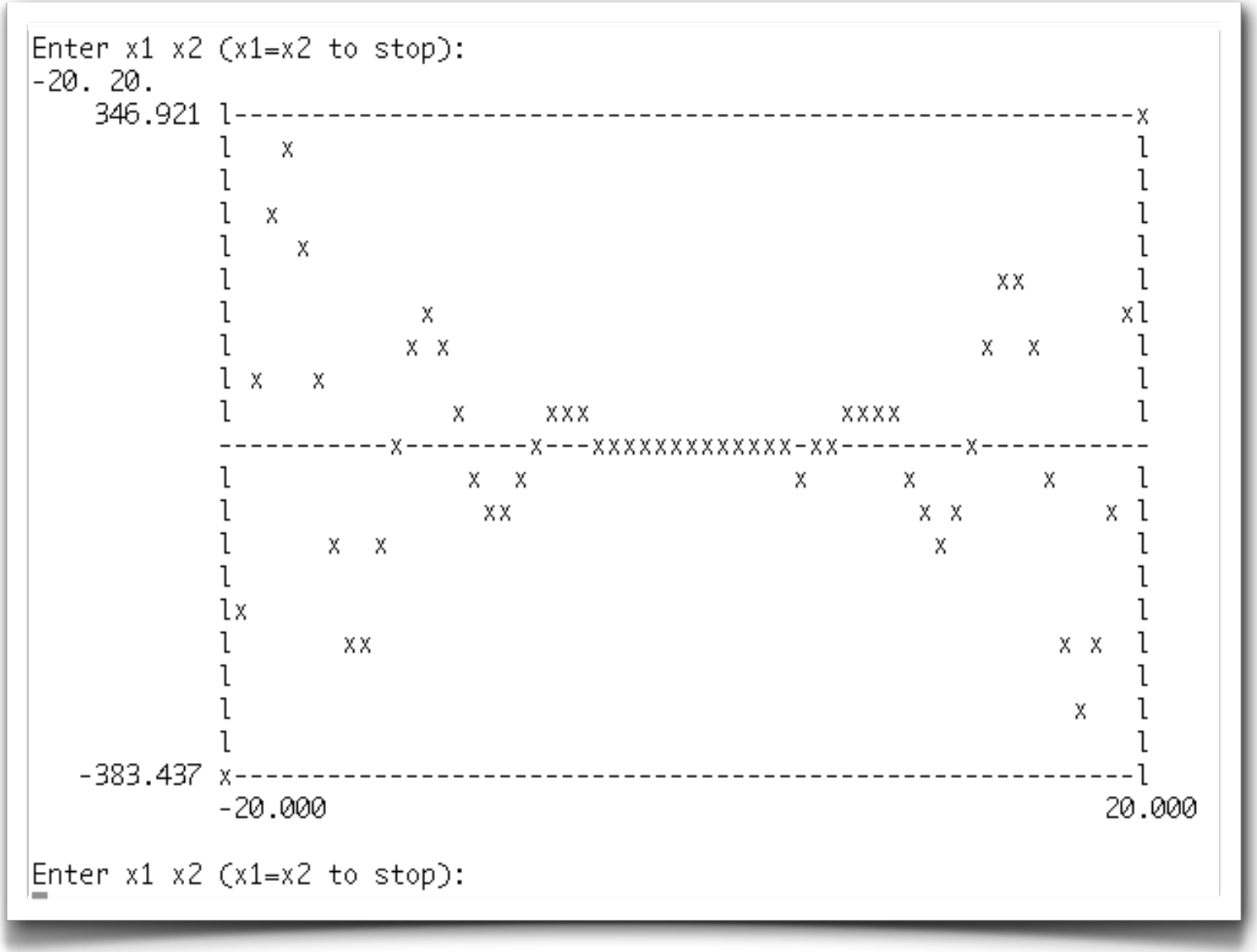
```
float rtsafe(void (*funcd)(float, float *, float *), float x1, float x2,
    float xacc)
```

Using a combination of Newton-Raphson and bisection, find the root of a function bracketed between `x1` and `x2`. The root, returned as the function value `rtsafe`, will be refined until its accuracy is known within $\pm$`xacc`. `funcd` is a user-supplied routine that returns both the function value and the first derivative of the function.

rtsafe.c

Grafische representatie van data

Een heel simpel functie om een ASCII plot van een functie te doen.



```
#include <stdio.h>
#define ISCR 60
#define JSCR 21
#define BLANK ' '
#define ZERO '-'
#define YY 'l'
#define XX '-'
#define FF 'x'

void scrsho(float (*fx)(float))
{
        int jz,j,i;
        float ysml,ybig,x2,x1,x,dyj,dx,y[ISCR+1];
        char scr[ISCR+1][JSCR+1];

        for (;;) {
                printf("\nEnter x1 x2 (x1=x2 to stop):\n");
                scanf("%f %f",&x1,&x2);
                if (x1 == x2) break;
                for (j=1;j<=JSCR;j++)
                        scr[1][j]=scr[ISCR][j]=YY;
                for (i=2;i<=(ISCR-1);i++) {
                        scr[i][1]=scr[i][JSCR]=XX;
                        for (j=2;j<=(JSCR-1);j++)
                                scr[i][j]=BLANK;
                }
                dx=(x2-x1)/(ISCR-1);
                x=x1;
                ysml=ybig=0.0;
                for (i=1;i<=ISCR;i++) {
                        y[i]=(*fx)(x);
                        if (y[i] < ysml) ysml=y[i];
                        if (y[i] > ybig) ybig=y[i];
                        x += dx;
                }
                if (ybig == ysml) ybig=ysml+1.0;
                dyj=(JSCR-1)/(ybig-ysml);
                jz=1-(int) (ysml*dyj);
                for (i=1;i<=ISCR;i++) {
                        scr[i][jz]=ZERO;
                        j=1+(int) ((y[i]-ysml)*dyj);
                        scr[i][j]=FF;
                }
                printf(" %10.3f ",ybig);
                for (i=1;i<=ISCR;i++) printf("%c",scr[i][JSCR]);
                printf("\n");
                for (j=(JSCR-1);j>=2;j--) {
                        printf("%12s"," ");
                        for (i=1;i<=ISCR;i++) printf("%c",scr[i][j]);
                        printf("\n");
                }
                printf(" %10.3f ",ysml);
                for (i=1;i<=ISCR;i++) printf("%c",scr[i][1]);
                printf("\n");
                printf("%8s %10.3f %44s %10.3f\n"," ",x1," ",x2);
        }
}
#undef ISCR
#undef JSCR
```

scrsho.c

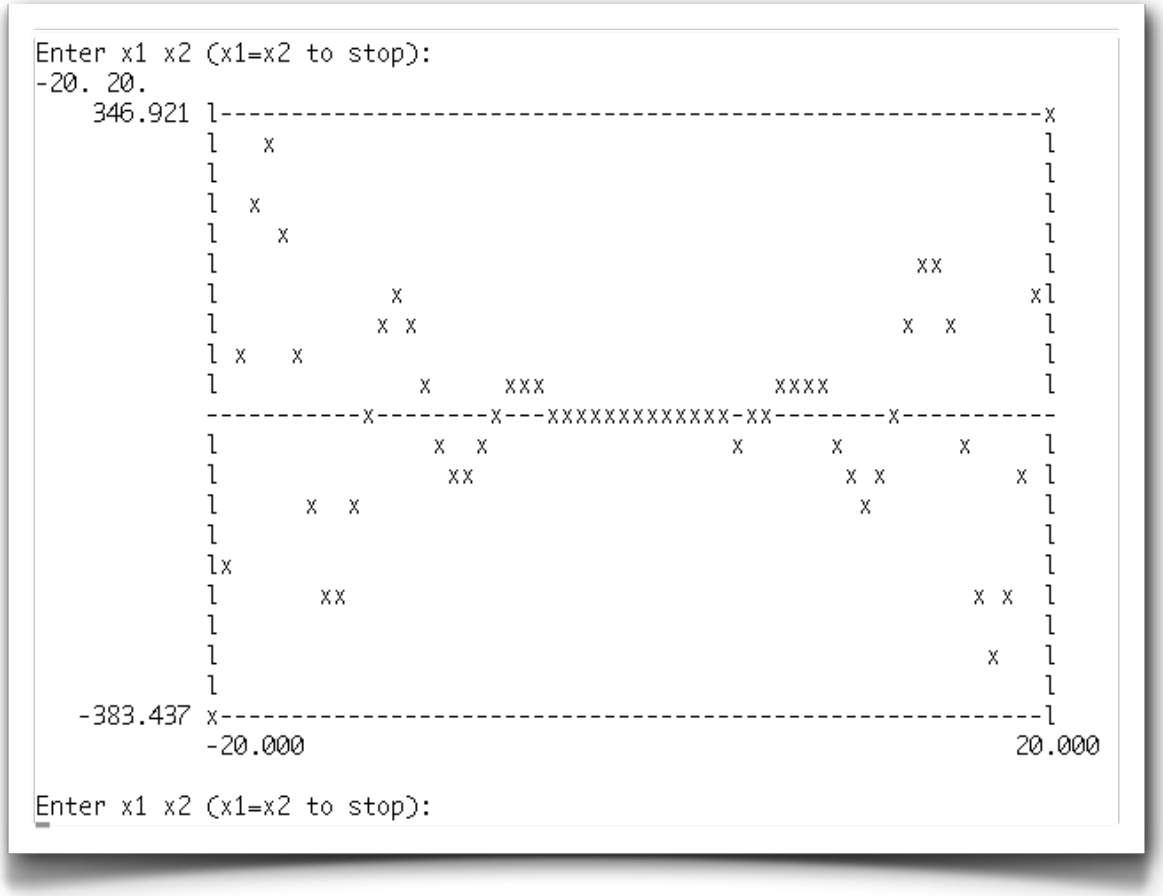
`void scrsho(float (*fx)(float))`
For interactive CRT terminal use. Produce a crude graph of the function `fx` over the prompted-for interval `x1,x2`. Query for another plot until the user signals satisfaction.

```c
#include<stdio.h>
#include<math.h>

#include "scrsho.c"

float function(float x)
 {
  return(x*(x-1)*sin(x));
 }

int main()
 {
  scrsho( (float(*)(float))function );
 }
```



plotfunction.c

Grafische representatie van data

Bijvoorbeeld hebben wij vaak in het natuurkunde practicum de opdracht een aantal metingen te verwerken.

voorbeeld: Wij hebben een tabel met metingen:
10 1.34
20 2.58
30 5.25
40 6.87
Deze schrijven wij in de bestand `data.in`**.**

In een C programma lezen wij dit bestand en wij doen de nodige berekeningen. Daarnaar roepen wij gnuplot om de data grafisch weer te geven.

Grafische representatie van data

GNU plot (http://gnuplot.info)

gnuplot is een standaard programma op onze Linux systeem.

```
> ssh -X username@lilo.science.ru.nl
> gnuplot
```

De data zijn in de bestand “data.out“.

```
> plot “data.out“ with lines
```

De data worden in een raam uitgegeven.

```
> gnuplot commands
```

gnuplot wordt uitgevoerd. “*commands*“ is een bestand met commando‘s voor gnuplot:

```
plot [0:50] "data.dat" with lines
pause 10
exit
```

Grafische representatie van data

```
#include<stdio.h>
#include<stdlib.h>

/* graphical output with gnuplot */

/* this functions reads the data and performs necessary
calculations etc. */
int read_data(FILE *fp)
 {
   /* here we have to do the necessary things */
  return(0);
 }

/* this functions writes the data to disk */
int write_data(FILE *fp)
 {
   /* here we have to do the necessary things */
  return(0);
 }
```

```
int main()
 {
  char input[]="data.in";
  char output[]="data.out";
  FILE *fp;

  printf("opening file ...\n");
  if( (fp=fopen(input,"r"))==NULL )
    {
     printf("file ERROR!\n");
     exit(1);
    }
  printf("reading data ...\n");
  read_data(fp);
  close(fp);

  if( (fp=fopen(output,"w"))==NULL )
    {
     printf("file ERROR!\n");
     exit(1);
    }
  printf("writing data ...\n");
  write_data(fp);
  close(fp);

  /* call gnuplot to display data */
  printf("call gnuplot ...\n");
  system("gnuplot commands");
 }
```

plot.c

Grafische representatie van data

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NBIN 20
#define NN 200000

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

        if  (iset == 0) {
                do {
                        v1=2.0*(float)rand()/RAND_MAX-1.0;
                        v2=2.0*(float)rand()/RAND_MAX-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
```

```
void printhisto(int *h)
 {
 int i,j,max=0;
 float scale;
 FILE *fp;

 // search maximum
 for(i=0; i<NBIN; i++)
   if(max<h[i]) max=h[i];
 // scale for printing
 scale=(float)max/50;

 for(i=0; i<NBIN; i++)
  {
   printf("%5i ",h[i]);
   for(j=0; j<h[i]/scale; j++) printf("#");
   printf("\n");
  }

 // write data to file
 fp=fopen("gnuplotexample.dat","w");
 for(i=0; i<NBIN; i++)
   fprintf(fp,"%f %f\n",-5+0.5*i ,(double)h[i]);
 close(fp);
 }
```

```
int main()
 {
 int i,index;
 int histo[NBIN];
 long int n=1;
 float x;

 // clear histogram
 for(i=0; i<NBIN; i++)
  histo[i]=0;

 // fill histogram
 for(i=0; i<NN; i++)
  {
// x=(float)rand()/RAND_MAX;
// index=x*(NBIN);
   x=gasdev(&n);
   index=(x+5)*(NBIN/10);
   if(index>=0 && index<NBIN) histo[index]++;
  }

 // print histogram
 printhisto(histo);
 system("gnuplot gnuplotexample.commands");
 }
```

gnuplotexample.c

DEBUGGER

Hoe kunnen wij in een programma fouten finden?
De debugger gdb is een tool om fouten in programmas te finden.

debugger - gdb

Wij compileren het programma met
gcc -g3 -o programma programma.c

De debugger wordt met
gdb ./programma
gestart.

Enkele debugger commands:
run: start de programma
where: waar in de code gebuurt de fault
print: drukt de inhoud van een variabele af
quit: beëindigt de debugger

```
GNU gdb 6.3.50-20050815 (Apple version gdb-1515) (Sat Jan 15 08:33:48 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin"...Reading symbols for shared
libraries .. done

(gdb) run
Starting program: /Users/jrh/Documents/Folien/prog1415/debug
Reading symbols for shared libraries +. done

Program received signal EXC_ARITHMETIC, Arithmetic exception.
0x0000000100000f2f in main () at debug.c:10
10        k= 8/(i-5);
(gdb) print k
$1 = -8
(gdb) print i
$2 = 5
(gdb)
```

debug.c

Introduction to GDB Commands

GDB contains a large repertoire of commands. The manual *Debugging with GDB* includes extensive documentation on the use of these commands, together with examples of their use. Furthermore, the command *help* invoked from within GDB activates a simple help facility which summarizes the available commands and their options. In this section we summarize a few of the most commonly used commands to give an idea of what GDB is about. You should create a simple program with debugging information and experiment with the use of these GDB commands on the program as you read through the following section.

`set args` *arguments*
The *arguments* list above is a list of arguments to be passed to the program on a subsequent run command, just as though the arguments had been entered on a normal invocation of the program. The `set args` command is not needed if the program does not require arguments.

`run`
The `run` command causes execution of the program to start from the beginning. If the program is already running, that is to say if you are currently positioned at a breakpoint, then a prompt will ask for confirmation that you want to abandon the current execution and restart.

`break` *location*
The break command sets a breakpoint, that is to say a point at which execution will halt and GDB will await further commands. *location* is either a line number within a file, given in the format `file:linenumber`, or it is the name of a subprogram. If you request that a breakpoint be set on a subprogram that is overloaded, a prompt will ask you to specify on which of those subprograms you want to breakpoint. You can also specify that all of them should be breakpointed. If the program is run and execution encounters the breakpoint, then the program stops and GDB signals that the breakpoint was encountered by printing the line of code before which the program is halted.

`breakpoint exception` *name*
A special form of the breakpoint command which breakpoints whenever exception *name* is raised. If *name* is omitted, then a breakpoint will occur when any exception is raised.

Introduction to GDB Commands

`print` *expression*
This will print the value of the given expression. Most simple Ada expression formats are properly handled by `GDB`, so the expression can contain function calls, variables, operators, and attribute references.

`continue`
Continues execution following a breakpoint, until the next breakpoint or the termination of the program.

`step`
Executes a single line after a breakpoint. If the next statement is a subprogram call, execution continues into (the first statement of) the called subprogram.

`next`
Executes a single line. If this line is a subprogram call, executes and returns from the call.

`list`
Lists a few lines around the current source location. In practice, it is usually more convenient to have a separate edit window open with the relevant source file displayed. Successive applications of this command print subsequent lines. The command can be given an argument which is a line number, in which case it displays a few lines around the specified one.

`backtrace`
Displays a backtrace of the call chain. This command is typically used after a breakpoint has occurred, to examine the sequence of calls that leads to the current breakpoint. The display includes one line for each activation record (frame) corresponding to an active subprogram.

Introduction to GDB Commands

`up`
At a breakpoint, GDB can display the values of variables local to the current frame. The command `up` can be used to examine the contents of other active frames, by moving the focus up the stack, that is to say from callee to caller, one frame at a time.
`down`
Moves the focus of GDB down from the frame currently being examined to the frame of its callee (the reverse of the previous command),
`frame` *n*
Inspect the frame with the given number. The value 0 denotes the frame of the current breakpoint, that is to say the top of the call stack.
The above list is a very short introduction to the commands that GDB provides. Important additional capabilities, including conditional breakpoints, the ability to execute command sequences on a breakpoint, the ability to debug at the machine instruction level and many other features are described in detail in *Debugging with GDB*. Note that most commands can be abbreviated (for example, c for continue, bt for backtrace).

(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.

```
(gdb) break 84
Breakpoint 1 at 0x100000c6c: file debug1.c, line 84.
(gdb) run < practicum/taak24.dat
Starting program: /Users/jrh/Documents/Folien/prog1415/debug1 < practicum/taak24.dat
Reading symbols for shared libraries +. done
give the input values for the matrix A[3][3]
the contents of the matrix A[3][3] is
   1.0000    2.0000    3.0000
   4.0000    5.0000    6.0000
   7.0000    8.0000    9.0000
give the input values for the vector b[3]
the contents of the vector b[3] is:
(    4.0000    5.0000    6.0000)

Breakpoint 1, main () at debug1.c:84
84              x[i]=det(AA)/d;
(gdb) print A
$1 = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
(gdb) print AA
$2 = {{4, 5, 6}, {4, 5, 6}, {7, 8, 9}}
(gdb) print i
$3 = 0
(gdb) print d
$4 = -54
(gdb) continue
Continuing.

Breakpoint 1, main () at debug1.c:84
84              x[i]=det(AA)/d;
(gdb) print AA
$5 = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
(gdb) continue
Continuing.

Breakpoint 1, main () at debug1.c:84
84              x[i]=det(AA)/d;
(gdb) print AA
$6 = {{1, 2, 3}, {4, 5, 6}, {4, 5, 6}}
(gdb) continue
Continuing.
and the solution of the system A*x=b is
the contents of the vector x[3] is:
(    2.5000    1.0000   -0.0000)

Program exited with code 012.
(gdb)
```

debugger - gdb

```
(gdb) run < practicum/taak24.dat
Starting program: /Users/jrh/Documents/Folien/prog1415/a.out < practicum/
taak24.dat
Reading symbols for shared libraries +. done
give the input values for the matrix A[3][3]
the contents of the matrix A[3][3] is
   1.0000    2.0000    3.0000
   4.0000    5.0000    6.0000
   7.0000    8.0000    9.0000
give the input values for the vector b[3]
the contents of the vector b[3] is:
(    4.0000    5.0000    6.0000)

Program received signal EXC_BAD_INSTRUCTION, Illegal instruction/
operand.
0x0000000100000c83 in main () at debug1.c:84
84          x[i]=det(AA)/d;
(gdb) print d
$1 = 5
(gdb) print AA
$2 = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
(gdb) print i
$3 = 1082130432
(gdb)
```

debug1.c