Appendix A. NTP Simulator Source Program Listing

Following is a listing of the NTP simulator used to test and evaluate the various algorithms of NTP. The Unix NTP daemon closely follows the construction of this simulator, which was also used as the basis of the code fragments given in the NTP Version 3 Specification RFC-1305. Note that the simulator as presented here is not a complete implementation supporting multiple peers, since its purpose in this report is to test and evaluate the modified local clock algorithms. Also, note that the various tests with simulated errors, spikes and ramps are usually done simply by modifying the source text, rather than a complete, user-friendly interface. Future enhancements may involve reading a script which specifies when test signals are to occur and their magnitude.

```
/*
 * NTP simulator
 *
 * This program reads sample data from a history file and simulates the
 * behavior of the NTP filter, selection and local-clock algorithms.
 *
 * Assertions
 *
 * 1     The clock discipline loop must be stable over a continous update
 *       interval range from 16 s to an indefinite upper limit, but in no
 *       case less than 16384 s.
 *
 * 2     The loop must have a capture range at least 300 ppm for any
 *       configuration of update interval and time error.
 *
 * 3     The minimum of the Allan variance characteristic is assumed at
 *       about 800 s. A phase-lock loop (PLL) is used below that update
 *       nterval, while a modified frequency-lock loop (FLL) is used
 *       above that.
 *
 * 4     The time discipline must be linear over the range +-128 ms) and
 *       use step adjustments outside that range.
 *
 * 5     To allow for erratic behavior of radio clocks in the vicinity of
 *       a leap second, a time step adjustement must not be performed
 *       until after a watchdog interval of 900 s during which all
 *       updates are outside the linear range.
 *
 * 6     While offsets exceed 128 ms or dispersions exceed 128 ms, only
 *       the most recent update in the clock filter is used.
 *
 * 7     An update from a peer with dispersion above 128 ms does not
 *       affect the clock frequency.
 *
 * 8     An update from a peer with synchronization distance greater than
 *       1 s is ignored.
```

```
 *
 * 9    Only those sample aged less than 800 s in the clock filter are
 *      used for time calculations. Dispersion calculations use all 8
 *      samples.
 *
 * 10   If an update increases the dispersion by a factor of 8 or more,
 *      the update is ignored (spike detector).
 */

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

#define DEBUG                   /* debug switch */

/*
 * Sizes and limits of things
 */
#define NMAX 40                 /* max number of clocks */
#define FMAX 8                  /* max clock filter size */
#define MAXCLOCK 10             /* max candidate clocks */
#define MINCLOCK 1              /* min survivor clocks */
#define LIMIT 30                /* poll-adjust threshold */

/*
 * Time offset values (s)
 */
#define MAXSKEW 1.318           /* max phase error per MAXAGE */
#define MAXPHASE .128           /* max phase error */
#define MAXFREQ 350e-6          /* max frequency error */

/*
 * Time interval values (s)
 */
#define DAY 86400.              /* one day of seconds */
#define MAXAGE 86400.           /* max clock age */
#define MAXREACH 86400.         /* reachability timeout */
#define MAXSEC 800.             /* max sample age */
#define MAXWATCH 900.           /* max watchdog interval */
#define MINPOLL 6               /* log2 min poll interval */
#define MAXPOLL 10              /* log2 max poll interval */

/*
 * Local clock oscillator characteristics
 */
#define HZ 1000.                /* clock rate (Hz) */
#define RHO (1. / HZ)           /* clock precision (s) */
#define PHI (MAXSKEW / MAXAGE)      /* clock frequency tolerance */
```

```
/*
 * Noise gates (s)
 */
#define MAXDISP 16.                 /* dispersion gate (s) */
#define MAXDIST 1.                  /* synch distance gate (s) */
#define SGATE 8.                    /* spike gate (ratio) */
#define PGATE 2.                    /* poll-adjust gate (ratio) */

/*
 * Time constants and weight factors
 */
#define FILTER .5                   /* clock filter weight */
#define SELECT .75                  /* clock select weight */

/*
 * Statistics counters
 */
struct stats {
        double d0;                  /* sample count */
        double d1;                  /* sum of samples */
        double d2;                  /* sum of sample squares */
        double d3;                  /* sample maximum */
        double d4;                  /* sample minimum */
} stats;                            /* system statistics counters */

/*
 * Peer state variables
 */
struct peer {
        double filtp[FMAX];         /* filter offset samples */
        double fildp[FMAX];         /* filter delay samples */
        double filep[FMAX];         /* filter dispersion samples */
        double tp;                  /* offset */
        double dp;                  /* delay */
        double ep;                  /* dispersion */
        double utc;                 /* last update time */
        int cd;                     /* peer id */
        int st;                     /* stratum */
        struct stats stats;         /* peer statistics counters */
} peer;                             /* temp peer structure */

/*
 * Function declarations
 */
void localclock(struct peer *);     /* local clock algorithm */
void filter(struct peer *, double, double, double); /* clock filter */
void dts(void);                     /* intersection algorithm */
void select(void);                  /* selection algorithm */
```

```
double combine(void);              /* combining algorithm */
double distance(struct peer *);     /* compute synch distance */
void clear(struct peer *);         /* clear peer */
void clrstat(struct stats *);      /* clear statistics */
void stat(struct stats *, double); /* update statistics */
void display(void);                /* display statistics */
void reset(void);                  /* reset system */
double gauss(double);              /* noise generator */

/*
 * State variables
 */
long day;                          /* modified Julian day */
double sec;                        /* seconds past midnight UTC */
double tstamp;                     /* time at current update */
double error;                      /* simulated error */
double mu;                         /* interval since last update */
double theta;                      /* clock offset */
double delta;                      /* roundtrip delay */
double epsil;                      /* synch distance */
double utc;                        /* time at last update */
double lasttime;                   /* time at last clock update */
double lasttheta;                  /* offset at last clock update */
double xi;                         /* select dispersion */
double frequency;                  /* frequency */
double stability;                  /* stability */
double noise;                      /* phase noise */
int pollx;                         /* log2 poll interval (s) */
double ooffset[3];                 /* simulated noise */

/*
 * Predicted state variables (s)
 */
double tp;                         /* clock offset */
double dp;                         /* roundtrip delay */
double ep;                         /* dispersion */
double rp;                         /* fractional frequency */

/*
 * Local clock tuning parameters. The ratio Kf / Kg^2 must be 4.
 */
double Kf = 65536;                 /* PLL frequency weight (2^16) */
double Kg = 64;                    /* PLL phase weight (2^6) */
double G = .25;                    /* frequency average */
struct peer *peers[NMAX];          /* where the peers are */
struct peer *plist[NMAX];          /* where the truechimes are */
double bot, top;                   /* confidence interval limits */
```

```
int npeers;                         /* number of peer clocks */
int nclock;                         /* number of survivor clocks */
struct peer *source;                /* clock source */
int acts = 0;                       /* switch for telephone update */
int tcnt;                           /* poll interval counter */
int dcnt;                           /* update counter */

/*
 * File format
 *
 *  MJD   UTC sec    Offset   Delay  Dispersion
 * -------------------------------------------
 * 49285 70976.089 -0.000156 0.00233 0.00008
 * 49285 70992.088 -0.000156 0.00233 0.00029
 * 49285 71008.089 -0.000156 0.00233 0.00061
 * 49285 71024.086 -0.000120 0.00241 0.00006
 * 49285 71040.095 -0.000120 0.00241 0.00026
 */
FILE *fopen(), *fp_in, *fp_out;     /* file handles */

/*
 * Main program
 *
 * Usage: ntp input file [ output file ]
 */
main(argc, argv)
        int argc; char *argv[];
{
        struct peer *pp;            /* peer structure pointer */
        double offset;              /* clock offset (ms) */
        double delay;               /* delay */
        double disp;                /* dispersion */
        double lclock;              /* leaping lizards local clock */
        int i;                      /* int temps */
        double dtemp;               /* double temps */

        if (argc < 2) {
                printf("usage: infile [outfile]\n");
                return (1);
        }
        if ((fp_in = fopen (argv[1], "r")) == NULL) {
                printf("Input file %s not found\n", argv[1]);
                return (1);
        }
        fp_out = stdout;
        if (argc > 2)
                fp_out = fopen (argv[2], "w");
```

```
peers[0] = &peer;               /* temp */
npeers = 1;
reset();

/*
 * Read next sample and find peer structure.
 *
 *      npeers          number of peers
 *      nclock          number of survivor clocks
 */
error = 0;
while (fscanf(fp_in, "%li%lf%lf%lf%lf",
        &day, &sec, &offset, &delay, &disp) != EOF) {
        dcnt++;

        /*
         * Provisions are made here to add Gaussian noise to the
         * offset samples for experiments. In the cast of ACTS,
         * this provides a dispersion estimate for the telephone
         * call.
         */
        offset += gauss(0e-6);
        if (acts) {
                ooffset[2] = ooffset[1];
                ooffset[1] = ooffset[0];
                ooffset[0] = offset;
                delay = 0;
                disp += max(max(fabs(ooffset[0] - ooffset[1]),
                    fabs(ooffset[1] - ooffset[2])), fabs(ooffset[0] - ooffset[2]));
        }
        if (dcnt % (1 << (pollx - 4)))
                continue;

        /*
         * Determine local time and offset. Provisions are made
         * to add a constant frequency offset for experiments.
         */
        tstamp = day * DAY + sec;
        if (utc == 0)
                utc = tstamp;
        mu = tstamp - utc;
        utc = tstamp;
        error += mu * 0e-6;
        lclock += tp + rp * mu;
        offset -= lclock + error;

        /*
         * Clear peers that have become unreachable.
```

```
		 */
		for (i = 0; i < npeers; i++)
			if (!acts && pp->utc != 0 && tstamp - pp->utc > MAXREACH)
				clear(peers[i]);

		/*
		 * Process sample and peer statistics.
		 */
		pp = peers[0];
		pp->cd = 1;
		pp->st = 1;
		filter(pp, offset, delay, disp);
		stat(&pp->stats, pp->tp);

		/*
		 * Determine survivors and combine offsets.
		 */
		dts();
		if (nclock < 1)
			continue;
		select();
		if (pp != source)
			continue;

		/*
		 * Update local clock and system statistics.
		 */
		theta = combine();
		delta = pp->dp;
		epsil = pp->ep + fabs(pp->tp) + delta/2 + xi;
		localclock(pp);
		dp = pp->dp;
		ep = pp->ep;
#ifdef DEBUG
		fprintf(fp_out,
		    "%5li%10.3lf %8.3lf %8.3lf %6.0lf %8.3lf %8.3lf %8.3lf%4i\n",
		    day, sec, theta * 1e3, rp * 1e6, mu, noise * 1e3,
		    frequency * 1e6, stability * 1e6, tcnt);
#endif
		stat(&stats, theta);
	}
	fclose(fp_in);
	if (fp_out != stdout)
		fclose(fp_out);
	display();
}
```

```
/*
 * Subroutines
 *
 * NTP local clock algorithm
 *
 *	pollx		log2 poll interval
 *	tp		time prediction
 *	rp		frequency prediction
 */
void
localclock(pp)
	struct peer *pp;		/* peer structure pointer */
{
	double ftmp;			/* double temps */
	int i;				/* int temps */

	/*
	 * If the absolute error exceeds MAXPHASE (128 ms), unlock the
	 * loop and allow it to coast up to MAXWATCH (900 s) with the
	 * poll interval clamped to MINPOLL. If the absolute error on a
	 * subsequent update is less than this, resume normal updates;
	 * if not, step the clock to the indicated time.
	 */
	noise = pp->ep + xi;
	if (fabs(theta) > MAXPHASE) {
		tcnt = 0;
		pollx = MINPOLL;
		if (!lasttime || mu > MAXWATCH) {
			for (i = 0; i < npeers; i++) {
				pp = peers[i];
				clear(pp);
			}
			tp = theta;
		} else {
			tp = 0;
			return;
		}

	/*
	 * If the dispersion exceeds MAXPHASE (128 ms), just set the
	 * clock and wait for the next update.
	 */
	} else if (noise > MAXPHASE) {
		tp = theta;

	/*
	 * If the dispersion has increased substantially over the
```

```
 * previous value, we have a spike which probably should be
 * suppressed.
 */
} else if (noise > SGATE * ep) {
        printf("outlyer %lf %lf %lf\n", ep, pp->ep, pp->tp);
        return;

/*
 * Use a phase-lock loop (PLL) at intervals < MAXSEC (800 s).
 */
} else if (mu < MAXSEC) {
        long tau;               /* PLL time constant */

        tau = 1 << (pollx - 4);
        tp = (1. - pow(1. - 1. / Kg, mu)) * theta / tau;
        rp += theta * mu / (tau * tau * Kf);

/*
 * Otherwise, use a hybrid frequency-lock loop (FLL).
 */
} else {
        tp = theta;
        rp += theta / mu * G;
}

/*
 * Clamp the frequency to the tolerance.
 */
if (rp > MAXFREQ)
        rp = MAXFREQ;
else if (rp <= -MAXFREQ)
        rp = -MAXFREQ;

/*
 * If the absolute error is greater than PGATE (4) times the
 * noise (sum of filter plus select dispersion), reduce the
 * theshold by twice the poll interval; if less than -LIMIT
 * (-30), halve the poll interval and reset the thresold. If
 * not, increase the threshold by the poll interval; if greater
 * than LIMIT (30) double the poll interval and reset the
 * threshold.
 */
if (mu > 1 << (MINPOLL - 1)) {
        if (fabs(theta) > PGATE * noise) {
                tcnt -= pollx << 1;
                if (tcnt < -LIMIT) {
                        tcnt = -LIMIT;
                        if (pollx > MINPOLL) {
```

```
                                        tcnt = 0;
                                        pollx--;
                                }
                        }
                } else {
                        tcnt += pollx;
                        if (tcnt > LIMIT) {
                                tcnt = LIMIT;
                                if (pollx < MAXPOLL) {
                                        tcnt = 0;
                                        pollx++;
                                }
                        }
                }

                /*
                 * Update raw phase and frequency noise estimates.
                 */
                if (lasttime)
                        mu = tstamp - lasttime;
                else
                        mu = 0;
                if (mu > 1 << (MINPOLL - 1)) {
                        ftmp = (theta - lasttheta) / mu;
                        frequency += (ftmp - frequency) / 4;
                        stability += (fabs(ftmp) - stability) / 4;
                }
        }
        lasttime = tstamp;
        lasttheta = theta;
}

/*
 * Clock filter algorithm
 *
 *      tstamp          current time
 */
void
filter(pp, offset, delay, disp)
        struct peer *pp;                /* peer structure pointer */
        double offset;                  /* sample offset */
        double delay;                   /* sample delay */
        double disp;                    /* sample dispersion */
{
        double list[FMAX];              /* synch distance array */
        int indx[FMAX];                 /* index list */
```

```
int i, j, k, n;                 /* int temps */
double x, y, phi;               /* double temps */

/*
 * Update clock filter and insert new sample
 */
if (pp->utc == 0)
        pp->utc = tstamp;
phi = tstamp - pp->utc;
pp->utc = tstamp;
for (i = FMAX - 1; i > 0; i--) {
        pp->filtp[i] = pp->filtp[i - 1];
        pp->fildp[i] = pp->fildp[i - 1];
        pp->filep[i] = pp->filep[i - 1] + PHI * phi;
}
pp->filtp[0] = offset;
pp->fildp[0] = delay;
pp->filep[0] = disp + PHI * delay;

/*
 * Construct temp list sorted by synch distance. This algorithm
 * keeps all samples with dispersion less than MAXDISP (16 s) in
 * fifo order, except those less than MAXSEC old, which are
 * sorted by distance.
 */
y = 0;
for (n = 0; n < FMAX; n++) {
        list[n] = pp->filep[n] + pp->fildp[n] / 2;
        indx[n] = n;
        for (j = 0; j < n && y < MAXSEC; j++) {
                if (list[j] > list[n]) {
                        x = list[j];
                        k = indx[j];
                        list[j] = list[n];
                        indx[j] = indx[n];
                        list[n] = x;
                        indx[n] = k;
                }
        }
        y += phi;
}

/*
 * Calculate filter dispersion.
 */
i = indx[0];
pp->tp = pp->filtp[i];
```

```
        pp->dp = pp->fildp[i];
        pp->ep = pp->filep[i];
        y = 0;
        for (i = n - 1; i >= 0; i--) {
                if (pp->filep[indx[i]] >= MAXDISP)
                        y = FILTER * (y + MAXDISP);
                else {
                        y = FILTER * (y + fabs(pp->filtp[indx[0]] - pp->filtp[indx[i]]));
                }
        }
        pp->ep += y;
}

/*
 * Intersection algorithm
 *
 *      peers           peer pointer array
 *      npeers          number of peers
 *      nclock          number of intersection peers
 *      bot             lowpoint
 *      top             highpoint
 */
void
dts()
{
        struct peer *pp;                /* peer structure pointer */
        double list[3 * FMAX];          /* temporary list */
        int indx[3 * FMAX];             /* index list */
        int f;                          /* intersection ceiling */
        int end;                        /* endpoint counter */
        int clk;                        /* falseticker counter */
        int i, j, k, n;                 /* int temps */
        double x, y;                    /* double temps */

        /*
         * This is a modification of the Marzullo algorithm in which the
         * midpoints of the interval intersections must be in the
         * intervals.
         */
        nclock = 0;
        i = 0;

        /*
         * Construct the endpoint list.
         */
        for (n = 0; n < npeers; n++) {
                pp = peers[n];
```

```
if (pp->ep >= MAXDISP)
        continue;
nclock++;
list[i] = pp->tp - distance(pp);
indx[i] = -1;           /* lowpoint */
for (j = 0; j < i; j++) {
        if ((list[j] > list[i]) || ((list[j] == list[i])
           && (indx[j] > indx[i]))) {
                x = list[j];
                k = indx[j];
                list[j] = list[i];
                indx[j] = indx[i];
                list[i] = x;
                indx[i] = k;
                }
}

i = i + 1;
list[i] = pp->tp;
indx[i] = 0;            /* midpoint */
for (j = 0; j < i; j++) {
        if ((list[j] > list[i]) ||
           ((list[j] == list[i]) && (indx[j] >
           indx[i]))) {
                x = list[j];
                k = indx[j];
                list[j] = list[i];
                indx[j] = indx[i];
                list[i] = x;
                indx[i] = k;
        }
}

i = i + 1;
list[i] = pp->tp + distance(pp);
indx[i] = 1;            /* highpoint */
for (j = 0; j < i; j++) {
        if ((list[j] > list[i]) ||
           ((list[j] == list[i]) && (indx[j] >
           indx[i]))) {
                x = list[j];
                k = indx[j];
                list[j] = list[i];
                indx[j] = indx[i];
                list[i] = x;
                indx[i] = k;
        }
```

```
		}
		i = i + 1;
	}

	/*
	 * Toss out falsetickers that do not lie in a common
	 * intersection.
	 */
	if (nclock < 1)
		return;
	for (f = 0; ; f++) {
		clk = 0;
		end = 0;
		for (j = 0; j < i; j++) {
			end = end - indx[j];
			bot = list[j];
			if (end >= (nclock - f))
				break;
			if (indx[j] == 0)
				clk = clk + 1;
		}
		end = 0;
		for (j = i-1; j >= 0; j--) {
			end = end + indx[j];
			top = list[j];
			if (end >= (nclock - f))
				break;
			if (indx[j] == 0)
				clk = clk + 1;
		}
		if (clk <= f)
			break;
	}
	nclock -= clk;
}

/*
 * Selection algorithm
 *
 *	peers		peer pointer array
 *	npeers		number of peers
 *	plist		survivor peer pointer array
 *	nclock		number of survivor peers
 *	bot		lowpoint
 *	top		highpoint
 *	source		system peer
 */
```

```
void
select()
{
	struct peer *pp;		/* peer structure pointer */
	struct peer *pptemp;		/* another one */
	double eps;			/* min peer dispersion */
	int i, j, k, m, n;		/* int temps */
	double x, y, z;			/* double temps */
	double list[NMAX];		/* synch distance array */

	/*
	 * Sort candidate list by synch distance. Note, all survivors
	 * must have distance less than MAXDISP (1 s), as left by the
	 * intersection algorithm.
	 */
	m = 0;
	for (n = 0; n < npeers; n++) {
		pp = peers[n];
		if (pp->tp >= bot && pp->tp <= top) {
			list[m] = MAXDISP * pp->st + distance(pp);
			plist[m] = peers[n];
			for (j = 0; j < m; j++) {
				if (list[j] > list[m]) {
					x = list[j];
					pptemp = plist[j];
					list[j] = list[m];
					plist[j] = plist[m];
					list[m] = x;
					plist[m] = pptemp;
				}
			}
			m = m + 1;
		}
	}
	nclock = m;
	if (m == 0) {
		source = 0;
		return;
	}

	/*
	 * Cast out outlyers with max select dispersion less than min
	 * filter dispersion.
	 */
	if (m > MAXCLOCK)
		m = MAXCLOCK;
	while (1) {
```

```
		xi = 0; eps = MAXDISP;
		for (j = 0; j < m; j++) {
			x = 0;
			for (k = m - 1; k >= 0; k--)
				x = SELECT * (x + fabs(plist[j]->tp - plist[k]->tp));
			if (x > xi) {	/* max(xi) */
				xi = x;
				i = j;
			}
			x = plist[j]->ep + PHI * (tstamp - plist[j]->utc);
			if (x < eps)
				eps = x; /* min(eps) */
		}

		if ((xi <= eps) || (m <= MINCLOCK))
			break;
		if (plist[i] == source)
			source = 0;
		for (j = i; j < m-1; j++)
			plist[j] = plist[j+1];
		m = m - 1;
	}

	/*
	 * Declare the winner, but avoid clockhopping if the winner is
	 * already one of the good guys.
	 */
	nclock = m;
	pp = plist[0];
	if (source != pp)
		if (source == 0)
			source = pp;
		else if (pp->st < source->st)
			source = pp;}

/*
 * Combining algorithm
 *
 *	plist		survivor peer pointer array
 *	nclock		number of survivor peers
 *	returns		 combined time offset
 */
double
combine()
{
	struct peer *pp;		/* peer structure pointer */
```

```
	int i, j;			/* int temps */
	double x, y, z;			/* double temps */

	y = z = 0;
	for (i = 0; i < nclock; i++) {
		pp = plist[i];
		x = MAXDISP * pp->st + distance(pp);
		y += 1. / x;
		z += pp->tp / x;
	}
	return (z / y);
}

/*
 * Compute synch distance
 *
 *	tstampupdate time
 *	returns synch distance
 */
double
distance(pp)
	struct peer *pp;		/* peer structure pointer */
{
	return (pp->ep + PHI * (tstamp - pp->utc) + fabs(pp->dp) / 2);
}

/*
 * Clear peer
 */
void
clear(pp)
	struct peer *pp;		/* peer structure pointer */
{
	int j;				/* int temps */

	for (j = 0; j < FMAX; j++) {
		pp->filtp[j] = pp->fildp[j] = 0;
		pp->filep[j] = MAXDISP;
	}
	pp->tp = pp->dp = pp->utc = 0;
	pp->ep = MAXDISP;
}

/*
 * Reset all variables
 *
 *	tp	system offset
 *	dp	system delay
```

```
 *      ep      system dispersion
 *      rp      system fractional frequency
 *      utc     last update time
 *      pollx   log2 poll interval
 *      source          synch source peer
 *      stats   system statistics sructure
 */
void
reset()
{
        struct peer *pp;                /* peer structure pointer */
        struct stats *sp;               /* statistics structure pointer */
        int i;                          /* int temps */

        tp = dp = ep = rp = utc = 0;
        pollx = MINPOLL;
        source = 0;
        ooffset[0] = ooffset[1] = ooffset[2] = 0;
        clrstat(&stats);
        for (i = 0; i < npeers; i++) {
                pp = peers[i];
                clear(pp);
                clrstat(&pp->stats);
        }
}

/*
 * Clear statistics
 */
void
clrstat(sp)
        struct stats *sp;               /* statistics structure pointer */
{

        sp->d0 = 0;
        sp->d1 = 0;
        sp->d2 = 0;
        sp->d3 = -1e10;
        sp->d4 = 1e10;
}

/*
 * Update statistics
 */
void
stat(sp, u)
        struct stats *sp;               /* statistics structure pointer */
```

```
        double u;                       /* sample update */
{

        sp->d0++;
        sp->d1 += u;
        sp->d2 += u * u;
        if (u > sp->d3)
                sp->d3 = u;
        if (u < sp->d4)
                sp->d4 = u;
}

/*
 * Display statistics
 */
void
display()
{
        struct peer *pp;                /* peer structure pointer */
        struct stats *sp;               /* statistics structure pointer */
        int i;                          /* int temps */
        double x, y, z;                 /* double temps */

        printf(" ID    Samp      Mean    StdDev       Max\n");
        sp = &stats;
        if (sp->d0 < 2) {
                x = 0;
                y = 0;
        } else {
                x = sp->d1 / sp->d0;
                y = sqrt(sp->d2 / sp->d0 - sp->d1 / sp->d0 * sp->d1 / sp->d0);
        }
        z = (sp->d3 - sp->d4) / 2.;
        printf("%3i%8.0f%11.3f%11.3f%11.3f\n", 0, sp->d0, x * 1e3, y * 1e3, z * 1e3);
        for (i = 0; i < npeers; i++) {
                pp = peers[i];
                sp = &pp->stats;
                if (sp->d0 < 2) {
                        x = 0;
                        y = 0;
                } else {
                        x = sp->d1 / sp->d0;
                        y = sqrt(sp->d2 / sp->d0 - sp->d1 / sp->d0 * sp->d1 / sp->d0);
                }
                z = (sp->d3 - sp->d4) / 2.;
                printf("%3i%8.0f%11.3f%11.3f%11.3f\n",
                    pp->cd, sp->d0, x * 1e3, y * 1e3, z * 1e3);
```

```
        }
}
/*
 * Subroutine gauss()
 *
 * This subroutine generates a random number uniformly distributed
 * over the interval [-1, 1], then transforms the distribution to
 * a Gaussian distribution with zero mean and specified standard
 * deviation.
 *
 * Calling sequence: x = gauss(sigma)
 *       sigma standard deviation of noise
 *       x       noise sample
 *
 * Variables and functions used (math library)
 *       rand() generate uniform random sample over [0, 32767]
 *       sqrt()  square rootsie
 *       log()   log base e
 */
double gauss(sigma)
        double sigma;                   /* standard deviation of noise */
        {
        double x, y;                    /* double temps */

        x = ((double)rand() / 16384 - 1);
        if (x > 0)
                y = sigma / sqrt(2) * log(1 / x);
        else if (x < 0)
                y = -sigma / sqrt(2) * log(1 / -x);
        else
                y = 0;
        return (y);
}
```