HF Simulator:
Channel and Modem Modules

Eric E. Johnson
Johnson Research
Las Cruces, NM
7/8/91

1 INTRODUCTION

This manual describes the design and use of software simulators for narrow-band HF channels, and for the robust modem specified in FED-STD-1045 and MIL-STD-188-141A. This software was developed by Johnson Research under the sponsorship of the Institute for Telecommunication Sciences, National Telecommunications and Information Administration, U.S. Department of Commerce, Boulder, Colorado. The initial work leading to the development of these simulators was sponsored by the U.S. Army Information Systems Engineering Command, Ft. Huachuca, Arizona.

Several approaches are often available to evaluate the potential performance of technical systems, including analytical (mathematical) models, simulations, and measurement of real systems. Of these, a mathematical model is usually the most powerful, in the sense that it is the fastest to evaluate, and therefore allows the widest range of design choices to be compared within a given period of time. The principal drawback of mathematical models is that the simplifying assumptions that must be made to allow solution of the model in closed form often restrict the range of applicability of the model, and may only approximately predict the performance of even the simplest real systems.

The most accurate technique for performance prediction is, of course, to build exactly the system of interest, and to measure its performance over the range of important situations. The difficulties in this approach are also obvious: first, it is usually economically impractical to construct all of the interesting variations on a design, especially when operating on a tight schedule; second, it may be impossible to produce the desired operating conditions on demand (e.g., to control the ionosphere to test HF systems).

Simulation techniques fall between the purely theoretical and the purely empirical approaches: they use mathematical models whenever such models are available, but do not require that closed form solutions exist; this permits the use of models with fewer simplifying assumptions, giving at least the *possibility* of more accurate results. When the system to be simulated is modeled in software, the design parameters of the system are usually easy to change, which permits simulations to evaluate a wider range of choices than the purely empirical approach of building each different design. Thus, simulation often strikes an attractive balance of accuracy, flexibility and speed.

Evaluators of HF modems and protocols have often used hardware "channel simulators" which embody the well-known Watterson model of narrow-band skywave propagation[1]. These simulators allow the evaluator to select three important ionospheric propagation characteristics — SNR, multipath delay, and fading bandwidth — and to maintain the chosen channel characteristics for the duration of a test. These channel simulators also possess the essential attribute of repeatability, which allows results to be reproduced and later tests to be performed under the same experimental conditions.

---

1 Watterson, *et al.*, "Experimental Confirmation of an HF Channel Model," *IEEE Transactions on Communication Technology*, **(COM-18)** 6, pp. 792-803, December 1970.

The essential distinction between these hardware simulators and the software simulators described here is that a software "channel simulator" can be combined with software simulations of modems, protocols, and so on much more easily than can a hardware simulator: combining hardware and software simulators would require A/D and D/A converters and the software to drive them, while a purely software approach connects the various simulators together in the link edit phase of compilation. Another benefit of the software approach is that simulations can be run on nearly any computer without the need to interact with a hardware simulator in real time. Finally, the most obvious benefit of a software simulator is that the costly hardware simulator itself is not needed.

The simulators described here are found at the lowest levels of a stack of simulators organized to correspond with the ISO model, as seen in Figure 1. These simulators are specifically oriented to the standard ALE waveform, but can be modified to accommodate other waveforms in a straightforward manner. The modem simulator converts tri-bits (integers from 0 to 7, inclusive) into vectors of complex numbers (representing the selected FSK tones) to be transmitted through the HF channel by the channel simulator module. The demodulator determines its output tri-bits by comparison of each received vector with a set of templates, one per FSK tone.

Both the modem simulator and the HF channel simulator are provided as independently-compilable modules, with functional interfaces to the main routine in the simulator. This main routine will usually provide the user interface, acquire the parameters needed for each simulation run, call the initialization routines in the modules, and then run the requested simulation(s). As an example of the use of these modules, we present and discuss a BER simulator.

The design and implementation of the simulator modules discussed here were guided by the often conflicting goals of speed, clarity, and portability. Within inner loops, the code is sometimes a bit cryptic in order to save time because these instructions account for significant fractions of the total execution time. Elsewhere the coding style is more transparent, and comments are included throughout.

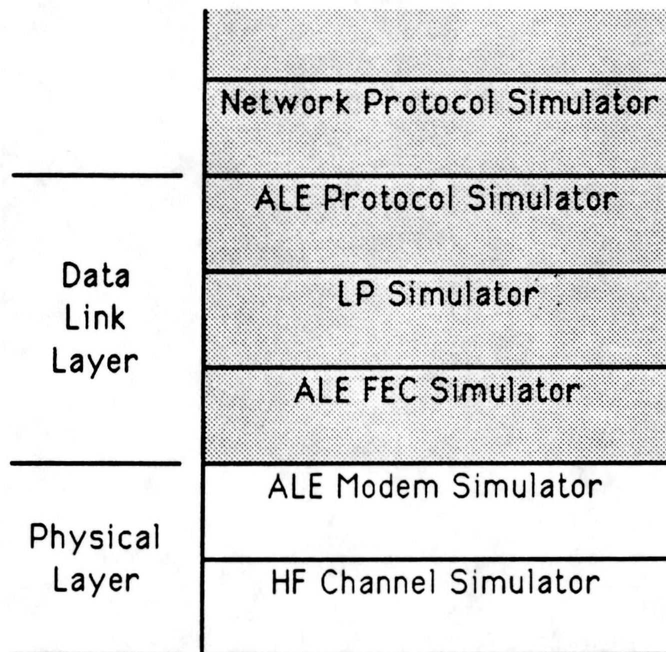


Figure 1: Simulator Stack

2 HF CHANNEL SIMULATOR

The HF channel simulator described here includes two equal-weight fading paths, and implements Watterson's model as follows:

1. Additive noise is generated with a Rayleigh amplitude distribution and a uniform phase distribution.
2. The second path is delayed from the main path by a fixed delay line. The delay may be a non-integral number of taps, in which case the output is obtained by interpolation.
3. Four independent fading gain processes are used for the in-phase and quadrature components of the direct and delayed paths. These processes have Gaussian amplitude distributions and Gaussian power spectra.

Each of these points is discussed in detail in the paragraph on the relevant functions below.

2.2 Data Structures

A few constants and data types and one external variable are defined in a header file `hfdef.h`, which are used by both the HF channel simulator and the ALE modem simulator:

```
#define Pi  3.141592654
#define TPi 6.283185307

#define Ts  0.008
#define NPTS 32

typedef double  SymVect[2][NPTS];

extern double SigLev;  /* declared in hflib.c */
```

`Pi` is simply the constant π. `TPi` is 2π. `Ts` is the symbol period: 8 ms. The modem tones (symbols) are processed in both simulators at a rate of `NPTS` samples per tone. Symbols are therefore stored as 32-sample complex vectors of type `SymVect`, in which row 0 is the in-phase (real) component and row 1 is the quadrature (imaginary) component. Finally, the global variable `SigLev` is declared as external.

2.3 HFchan

The key function in the channel simulator is `HFchan`; it uses the following variables and data structures:

```
double SigLev = 1.0;            /* global so it can be set by main routine */

static SymVect RcvSig;
```

The global variable `SigLev` sets the signal-to-noise ratio of the HF channel. (`SigLev` is global so that the main routine can set the SNR according to user requests, or vary the SNR through several runs.) `RcvSig` is a statically-allocated symbol vector (`SymVect`) used by `HFchan` to store its result.

```
SymVect *HFchan(XmitSig)
     SymVect *XmitSig;
{
    register int i;
    register SymVect *in, *out;
    double   ni, nq, Ri, Rq, RDi, RDq, tmp1, tmp2;
    register double Xi, Xq, Di, Dq;  /* in-phase and quadrature main and  */
                                     /* delayed path samples, respectively */
    in = XmitSig;
    out = RcvSig;
    for (i=0; i<NPTS; i++) {
        Rayleigh(&ni, &nq);  /*                GENERATE NOISE              */
                             /* Rayleigh generates in-phase and quadrature */
                             /* components, jointly normal, with each      */
                             /* component having RMS amplitude of unity;   */
                             /* RMS noise power is also unity.             */

        if (FreqSpread > 0.0) {         /*      GENERATE FADING GAINS       */
            FadeGains(IFade0, QFade0); /* generate direct-path fade gains  */
            FadeGains(IFade1, QFade1); /* generate delayed-path fade gains */
        }
        else {
            Rayleigh(&tmp1, &tmp2);    /* don't care (to draw RNs)         */
            Rayleigh(&tmp1, &tmp2);    /* leave fade gains at initialized  */
        }                              /*   values (phase shift only)      */

        if (DelayLineLen + DlyTF > 1.0) {         /*  multipath?  */
            Di = DelayLine(Xi=SigLev*(*in)[0][i], &iptr, IDLine);/* generate */
            Dq = DelayLine(Xq=SigLev*(*in)[1][i], &qptr, QDLine);/* delayed  */
                                                                 /* signals  */

            CompMult(Xi, Xq, *IFade0, *QFade0, &Ri, &Rq);/* introduce fading */
            CompMult(Di, Dq, *IFade1, *QFade1, &RDi, &RDq);
        }
        else {                      /* all power in single path */
            CompMult((*in)[0][i], (*in)[1][i], *IFade0, *QFade0, &Ri, &Rq);
            RDi = RDq = 0.0;
            Ri *= sqrt2;
            Rq *= sqrt2;
        }
        (*out)[0][i] = Ri + RDi + ni;      /* compute output */
        (*out)[1][i] = Rq + RDq + nq;
    }
    return (out);
}
```

The function calling HFchan passes a pointer to a symbol vector, which is to be processed by the simulated HF channel. One sample at a time is processed in the inner loop (see the code above). First, the samples of noise for each time step are generated; then the fading gains are computed for the direct and delayed paths; next, the delay lines are updated, returning the current delayed path sample, and the fade gains computed above are applied; finally, the noise and direct and delayed path samples are added together and the result is stored in the output symbol vector. When all of the elements of the input vector have been processed, a pointer to the output vector is returned.

The registered pointers in and out are used for fast access to the input and output symbol vectors, respectively. ni and nq store the noise samples, Xi and Xq the input samples, Di and Dq the delayed samples, and Ri, Rq, RDi, and RDq the results for each time step.

Note that the output symbol vector is always stored in RcvSig, and that the external function is given a pointer to this (static) array. If that function needs access to more than one received symbol at a time, it will need to pass a pointer to a symbol vector to (a modified version of) `HFchan` each time.

2.4 DelayLine

The delayed path used to simulate multipath is implemented as a pair of digital delay lines, `IDLine` (in-phase) and `QDLine` (quadrature). These arrays hold circular buffers of `DelayLineLen` samples. When the delay is not an integral multiple of the sample period, `DlyTF` stores the odd fraction of a sample. `iptr` and `qptr` store the current positions in the circular buffers.

```
static int DelayLineLen;          /* if more than one set of delay lines */
static double DlyTF;              /* needed, must pass these as parms    */

static int iptr, qptr;            /* these are passed as parms */
static double IDLine[65], QDLine[65];
```

The `DelayLine` function is given a pointer `LName` to the delay line to use (which is just an array of double precision floating point numbers), the current offset `Ptr` into the circular buffer embedded in that array, and the current direct path sample `InSig` to put into the delay line. The offset is incremented modulo `DelayLineLen`, the length of the circular buffer, and `nt` is set to the offset of the next element past the updated `Ptr`, again modulo `DelayLineLen`. The delay line output for this time step is interpolated from the elements pointed to by `Ptr` and `nt`, and `InSig` is put into the delay line.

However, if the total delay is less than one sample, the circular buffer will be of length 1, and only the first element of the array will be used.

```
double DelayLine(InSig, Ptr, LName)
     double InSig;     /* current sample: put into delay line */
     int *Ptr;         /* current offset into delay line      */
     double *LName;    /* ptr to delay line to use            */
{
    register double *tempptr, temp;
    register int nt;

    if (DelayLineLen > 1) {  /* use the delay line */
        nt = (++(*Ptr)+1) % DelayLineLen;
        tempptr = LName + (*Ptr %= DelayLineLen);
        temp = *tempptr * DlyTF + (*(LName+nt)) * (1.0 - DlyTF);
        *tempptr = InSig;
    }
    else {  /* use only first element of delay line */
        temp = InSig * (1.0 - DlyTF) + *LName * DlyTF;
        *LName = InSig;
    }
    return (temp);
}
```

The delay line function could be optimized for the common case of an integral number of samples in the delay time (`DlyTF` = 0.0). The interpolation statements could also be rewritten to reduce the number of floating point operations. Because this function is called in the inner loop, these could be worthwhile.

2.5 Rayleigh

Noise is generated using a pseudorandom number generator with a uniform (0, 1) distribution. A Rayleigh amplitude is produced using the inverse transform technique, and a uniformly-distributed phase is generated directly. The function returns its results in rectangular coordinates.

```
void Rayleigh (x, y)
     double *x, *y;
{
    register double r, a;

    r = sqrt(-2.0*log(uniform()));
    a = TPi * uniform();
    *x = r * cos(a);
    *y = r * sin(a);
}
```

2.6 FadeGains

The fade gains module contains three functions:

`FadeGains`, the external interface to the module

`Gauss_Filter`, a DSP filter which produces fade gains with the correct statistical properties

`CompMult`, which multiplies complex arguments to apply the complex fade gains

`FreqSpread, g, a0, a1,` and `a2` are the fading bandwidth (Doppler spread) and coefficients used in the DSP Gaussian filter, respectively. The `IFade0, QFade0, IFade1,` and `QFade1` arrays hold the state variables for in-phase and quadrature, direct and delayed paths, respectively, for the filter; in the usual DSP terms, these arrays hold (in order) y[n], y[n-1], y[n-2], x[n], x[n-1], and x[n-2]: the current and past outputs and the current and past inputs, as discussed further in the Gauss_Filter section.

```
static double  FreqSpread, g, a0, a1, a2;
static double  IFade0[6], QFade0[6], /* direct-path fading filter state vars */
               IFade1[6], QFade1[6];/* delayed-path fading filter state vars */
```

The Watterson model specifies that the fading gains (tap gains) be independent zero-mean complex-Gaussian random processes with Rayleigh amplitude and uniform phase density functions, and Gaussian power spectra. Random processes with the requisite characteristics are produced by passing Rayleigh-distributed noise through a filter whose $|H(j\Omega)|^2$ has the required Gaussian shape.

```
void FadeGains(i, q)
     double *i, *q;
{
    /* generate Rayleigh-distributed fade gain perturbations */
    Rayleigh(i+3, q+3); /* assign to current input positions */

    /* Run through gaussian filter */
    Gauss_Filter(i);
    Gauss_Filter(q);
}
```

2.7 Gauss Filter

Watterson *et al.* chose a Gaussian shape for the power spectrum of the tap gain functions in their model:

$$v_{si}(\nu) = \frac{C_{sia}(0)}{(2\pi)^{1/2}\,\sigma_{sia}} \exp\left[-\frac{(\nu-\nu_{sia})^2}{2\sigma_{sia}^2}\right] + \frac{C_{sib}(0)}{(2\pi)^{1/2}\,\sigma_{sib}} \exp\left[-\frac{(\nu-\nu_{sib})^2}{2\sigma_{sib}^2}\right]$$

where the s subscript indicates Watterson's specific model, the i subscript refers to the i-th tap, and the a and b subscripts refer to the two magnetoionic components (where these are distinguishable). $C_{si_}(0)$ is the fraction of total channel input power delivered by each component, $\sigma_{si_}$ is the frequency spread of each component, and $\nu_{si_}$ is the frequency offset of the power spectrum of each component. (All frequencies in the Watterson model are in Hz.) In keeping with usual practice, our simulator does not resolve the magnetoionic components, and therefore requires only a single Gaussian function for the tap-gain spectrum, and no frequency offset ν_i is used.

The spectrum $v_{si}(\nu)$ is the Fourier transform of the tap-gain correlation function. The correlation function of the output of a filter whose input is Rayleigh noise is simply the magnitude squared of its impulse response, so its power spectrum is the magnitude squared of its transfer function. Thus, we may produce a random process with the correct spectrum by passing Rayleigh noise through a filter with a transfer function magnitude squared given by

$$|H_i(j\Omega)|^2 = \frac{C(0)}{\sqrt{2\pi}\,\sigma} \exp\left[-\frac{\Omega^2}{2(2\pi\sigma)^2}\right]$$

In our simulator, the two paths have equal power and the same frequency spread, so the subscripts on C and σ are dropped.

This transfer function can be closely approximated by a two-pole filter of the form shown below. The power spectra of the two filters are compared in Figure 2.

$$H_a(s) = \frac{\sqrt{G}}{\left(\frac{cs}{2\pi\sigma}\right)^2 + \left(\frac{as}{2\pi\sigma}\right) + 1}$$

Clearly, the DC gains of the filters must be the same, so

$$G = \frac{C(0)}{\sqrt{2\pi}\,\sigma}.$$

The parameter **a** must be set to $\sqrt{\pi/2}$ for the correct total power; c was set to 3/4 to obtain the approximation to a Gaussian shape shown in Figure 2.

A discrete-time (i.e., DSP) filter to implement the $H_a(s)$ above was designed using the bilinear transformation

$$s \rightarrow \frac{2}{T}\left(\frac{1-z^{-1}}{1+z^{-1}}\right) = \frac{2}{T}\left(\frac{z-1}{z+1}\right)$$

The sample rate (4000 samples/s) so far exceeds the frequency spreads of interest (a few Hz) that pre-warping of the critical frequencies is not necessary.

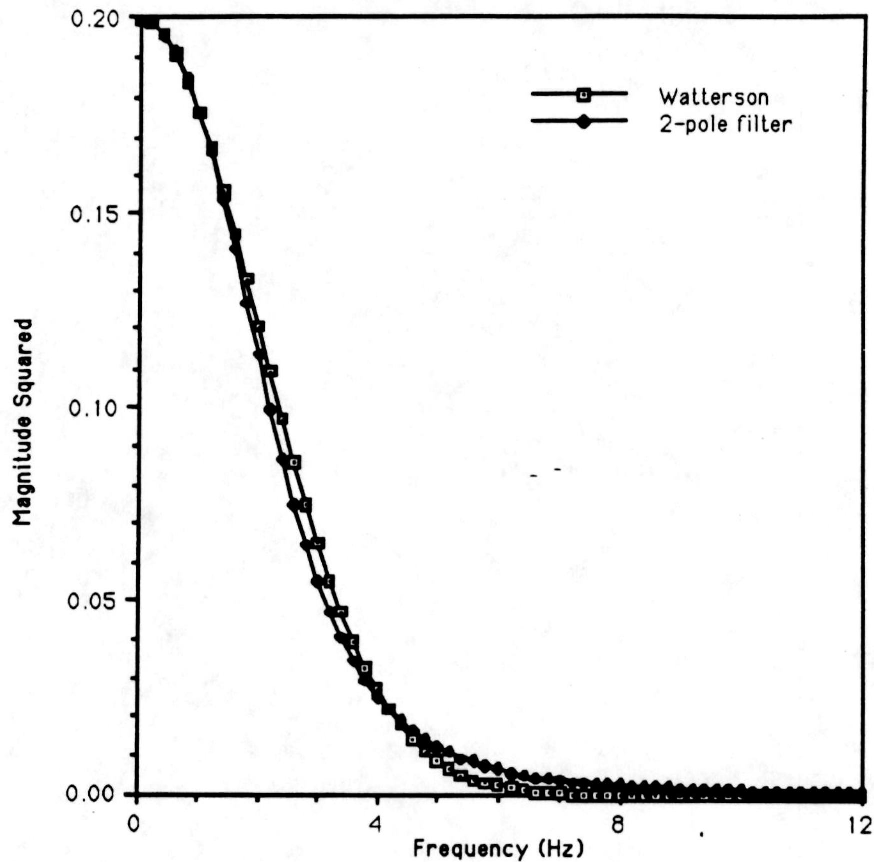


Figure 2: Comparison of Power Spectra (2 Hz Frequency Spread)

The bilinear transformation on $H_a(s)$ produces an H(z) in the z-transform domain:

$$H(z) = \frac{\sqrt{G}\,(z+1)^2}{\frac{4c^2}{4\pi^2\sigma^2T^2}(z-1)^2 + \frac{2a}{2\pi\sigma T}(z-1)(z+1) + (z+1)^2}$$

$$= \frac{\sqrt{G}\left(z^2+2z+1\right)}{\left(\frac{a}{\pi\sigma T} + \left(\frac{c}{\pi\sigma T}\right)^2 + 1\right)z^2 + 2\left(1 - \left(\frac{c}{\pi\sigma T}\right)^2\right)z + \left(\left(\frac{c}{\pi\sigma T}\right)^2 - \frac{a}{\pi\sigma T} + 1\right)}$$

Comparing this expression for the desired H(z) to the standard form for a DSP filter

$$H(z) = \sqrt{G}\left(\frac{\sum_{r=0}^{2} b_r z^{-r}}{\sum_{k=0}^{2} a_k z^{-k}}\right),$$

we extract the values for the coefficients by inspection:

$$a_0 = A+C+1 \quad a_1 = 2(1-C) \quad a_2 = C-A+1$$

$$b_0 = b_2 = 1 \qquad b_1 = 2$$

where

$$A = \frac{a}{\pi\sigma T} \text{ and } C = \left(\frac{c}{\pi\sigma T}\right)^2.$$

The current output y[n] is computed from the past outputs y[n-1] and y[n-2] and the current and past inputs x[n], x[n-1], and x[n-2] using the following recurrence relation, where $g = \sqrt{G}$:

$$a_0\, y[n] + a_1\, y[n-1] + a_2\, y[n-2] \;=\; g\,(b_0\, x[n] + b_1\, x[n-1] + b_2\, x[n-2])$$

$$y[n] \;=\; (g\,(b_0\, x[n] + b_1\, x[n-1] + b_2\, x[n-2]\,) - a_1\, y[n-1] - a_2\, y[n-2])\,/\,a_0$$

This expression is used in the code for `Gauss_Filter` listed below.

```
void Gauss_Filter(Fade)
     double *Fade;
{
    /* Fade array of input/output data:              */
    /*  Fade[0] is the current filter output         */
    /*  Fade[0-2] are the current and past outputs */
    /*  Fade[3-5] are the current and past inputs   */

    /* Gaussian filter:  2-pole, 2-zero IIR */
    Fade[0]=(g*(Fade[3]+2*Fade[4]+Fade[5])-a1*Fade[1]-a2*Fade[2])/a0;

    /* adjust the history terms */
    Fade[2]=Fade[1];
    Fade[1]=Fade[0];
    Fade[5]=Fade[4];
    Fade[4]=Fade[3];
}
```

2.8 CompMult

CompMult simply multiplies two complex arguments to produce a complex result. Because this function is called twice in the inner loop, some speedup could be achieved by making this a macro.

```
void CompMult (Xi, Xq, Yi, Yq, Zi, Zq)          /*  Z = X * Y     */
     double Xi, Xq, Yi, Yq, *Zi, *Zq;
{
    *Zi = Xi*Yi - Xq*Yq;
    *Zq = Xi*Yq + Xq*Yi;
}
```

2.8 Initialize

The initialization routine listed here directly queries the user for the multipath delay and frequency spread parameters. Where this is not appropriate, these two parameters should be passed to `Initialize` as arguments.

```
/**********************  INITIALIZATION *************************************/

void Initialize()
{
    register int i;
    double DlyTime, a, c, A, C, dt;
```

`DlyTime` is the floating point value of multipath delay provided by the user; it is converted to the global variables `DelayLineLen` and `DlyTF` for use by `DelayLine`. `dt` is the time step (sample interval) used, and is set to `Ts/NPTS`.

```
/******************** SET UP DELAY LINES *********************/
    do {
      printf("Enter delay time (ms):  ");
      scanf("%lf", &DlyTime);
    } while (DlyTime < 0.0);

    dt = Ts/NPTS;               /*  dt = sample period  */
    DlyTime /= (dt * 1e3);  /* scale DlyTime from milliseconds to samples */
    DlyTF = DlyTime - (DelayLineLen = (int)DlyTime);
    DelayLineLen++;
    iptr = qptr = 0;
    for (i=0; i<65; i++) IDLine[i] = QDLine[i] = 0.0;
```

The variables a, c, A, and C are as described in the Gauss_Filter section. However, because FreqSpread is scaled by 2π times dt, a and c have twice the values described earlier.

Half of the total power is allotted to each path, resulting in the expression shown for g.

The initialization of the fading filter state variables is used to overcome initial transients.

```
    /*****************  SET UP FADING GENERATOR  *****************/
    do {
      printf("Enter frequency spread (Hz):  ");
      scanf("%lf", &FreqSpread);
    } while (FreqSpread < 0.0);

    if (FreqSpread > 0.0) {
      FreqSpread *= dt*TPi;    /*  scale from Hz to radians per sample  */

      /* magic numbers for Gaussian filter */
      a = sqrt(TPi);
      c = 1.5;
      A = a/FreqSpread;
      C = c/FreqSpread;     C *= C;

      /* Gaussian filter coefficients */
      g = sqrt(0.5*sqrt(TPi)/FreqSpread);
      a0 = A + C + 1.0;
      a1 = 2*(1.0 - C);
      a2 = C + 1.0 - A;

      for (i=0; i<6; i++)   /* clear fading filter state variables . . . */
        *(IFade0+i) = *(QFade0+i) = *(IFade1+i) = *(QFade1+i) = 0.0;

      for (i=0; i<1.0/FreqSpread; i++) {          /* and initialize them */
        FadeGains(IFade0, QFade0);
        FadeGains(IFade1, QFade1);
      }
    }
    else          /* fading generator disabled; set fixed gains for taps */
      *(IFade0) = *(QFade0) = *(IFade1) = *(QFade1) = sqrt2/2.0;
}
```

3 UTILITIES

3.1 Uniform

This random number generator with uniform distribution is described in the *Communications of the ACM*, **(31)** 6, pp. 742-749+, June 1988.

```
double uniform()
{
    static long s1 = 123456787, s2 = 987654323;
    long z, k;

    k = s1 / 53668;
    s1 = 40014 * (s1 - k * 53668) - (k * 12211);
    if (s1 < 0) s1 += 2147483563;

    k = s2 / 52774;
    s2 = 40692 * (s2 - k * 52774) - (k * 3791);
    if (s2 < 0) s2 += 2147483399;

    z = s1 - s2;
    if (z < 1) z += 2147483562;

    return (z * 4.656613E-10);
}
```

3.2 RandInt

`RandInt` generates random integers in a specified range. It is useful for generating "random" tri-bits to be sent through the simulator.

```
int RandInt(i)   /* Generates random integers in the range 0 to i-1.      */
    int i;       /* In run of 100,000 with 32 bins, had variation of      */
{                /* 6% of mean from max bin to min bin.                    */
    return (i*uniform());
}
```

3.3 HammingWt3

`HammingWt3` returns the number of 1 bits in the 3 least-significant bits of its argument.

```
int hammingwt3(w)
int w;
{
    static int wt[8] = {0, 1, 1, 2, 1, 2, 2, 3};

    return (wt[w & 7]);
}
```

4 ALE MODEM SIMULATOR

The ALE modem simulation routines convert tri-bits to and from symbol vectors suitable for processing by the channel simulator.

4.1 Data Structures

The `tone` array is pre-computed by `InitModem`, and contains the complex symbol vector corresponding to each of the FSK tones.

```
static double tone[m_ary][2][NPTS];
```

4.2 FSKmod

`FSKmod` simply returns a pointer to the appropriate `tone`.

```
SymVect *FSKmod(InSym)              /*  use: SymVect *XmitSig         */
     int InSym;                     /*       XmitSig = FSKmod(InSym)  */
{                                   /*           ... (*XmitSig)[i][j] */
  return (tone[InSym]);
}
```

4.3 FSKdemod

The FSK demodulator shown below employs a "guard band" technique: it ignores the first and last 25% of the baud, and only integrates over the middle 50%; this reduces the effect of multipath delays of up to 2 ms. Also, the demodulator stays synchronized with the direct path signal, even when the delayed path is delivering more power; this may be unrealistic and sub-optimal.

```
int FSKdemod(RcvSig)
     SymVect *RcvSig;
{
  register int i, j, out;
  register double sumi, sumq, sum, max=0.0;
  register SymVect *in;

  in = RcvSig;
  for (i=0; i<m_ary; i++)  {
       sumi = sumq = 0.0;
       for (j=NPTS/4; j<(3*NPTS)/4; j++)  {
            sumi += (*in)[0][j]*tone[i][0][j] + (*in)[1][j]*tone[i][1][j];
            sumq += (*in)[1][j]*tone[i][0][j] - (*in)[0][j]*tone[i][1][j];
       }
       if ((sum = sumi*sumi + sumq*sumq) >= max) {
            max = sum;
            out = i;
       }
  }
  return (out);
}
```

4.4 InitModem

The purpose of `InitModem` is to initialize the `tone` vectors for the eight FSK tones.

```
void InitModem()
{
  register int i, iprime, j;
  double f[m_ary], dt;
  register double x, dx;

  f[0] = -(f[7] = 875.0);          /* 8-ary FSK tone frequencies */
  f[1] = -(f[6] = 625.0);
  f[2] = -(f[5] = 375.0);
  f[3] = -(f[4] = 125.0);

  dt = Ts/NPTS;

  for (i=0; i<m_ary/2; i++) {      /* generate tone vectors    */
      iprime = m_ary-i-1;          /* index of conjugate frequency */
      x = 0.0;
      dx = TPi*f[iprime]*dt;
      for (j=0; j<NPTS; j++) {
          tone[i][0][j] = tone[iprime][0][j] = cos(x);  /* real part of tone */
          tone[i][1][j] = -(tone[iprime][1][j] = sin(x));/* imag part of tone*/
          x += dx;      if (x >= TPi) x -= TPi;
      }
  }
}
```

5 BER SIMULATOR

The BER simulator is a simple program that illustrates the use of the channel and modem modules.

```
main()
{
  long i, insym, outsym, RunLength, errors=0;

  printf("SNR (dB):  ");
  scanf("%lf", &SigLev);
  SigLev = pow(10.0, SigLev/20.0); /* convert from dB to voltage ratio */

  Initialize();
  InitModem();

  printf("Run length:  ");
  scanf("%ld", &RunLength);
  printf("\n");

  for (i=0; i<RunLength; i++)
    if ((outsym = ALEmodem(insym = RandInt(8))) != insym) {
      printf("*");
      errors += hammingwt3(insym ^ outsym);
    }
    else printf(".");

  printf("\nBER = %5.3lf\n", (double)errors/(3.0 * RunLength));
}
```

6 MEASUREMENTS

6.1 SNR Calibration

Good Channel

```
Multipath delay = 0.5 ms
Fading BW = 0.1 Hz

Bauds =     10  Average SNR =  -9.73
Bauds =    100  Average SNR =  -3.53
Bauds =   1000  Average SNR =  -0.26
Bauds =  10000  Average SNR =  -0.06
Bauds =  50000  Average SNR =  -0.08
```

Poor Channel

```
Multipath delay = 2 ms
Fading bandwidth = 2 Hz

Bauds =     10  Average SNR =  -5.93
Bauds =    100  Average SNR =  -1.34
Bauds =   1000  Average SNR =   0.18
Bauds =  10000  Average SNR =  -0.07
Bauds =  50000  Average SNR =   0.05
```

Gaussian Channel

```
Multipath delay = 0
Fading bandwidth = 0

Bauds =     10  Average SNR =  -0.39
Bauds =    100  Average SNR =   0.22
Bauds =   1000  Average SNR =   0.16
Bauds =  10000  Average SNR =   0.15
Bauds =  50000  Average SNR =   0.14
```

Flat Fading Channel

```
Multipath delay = 0
Fading bandwidth = .1

Bauds =     10  Average SNR =  -8.14
Bauds =    100  Average SNR =  -2.06
Bauds =   1000  Average SNR =   1.55
Bauds =  10000  Average SNR =  -0.48
Bauds =  50000  Average SNR =  -0.49
```

Multipath-Only Channel

```
Multipath delay = .1
Fading bandwidth = 0

Bauds =     10  Average SNR =  -0.88
Bauds =    100  Average SNR =  -0.27
Bauds =   1000  Average SNR =  -0.33
Bauds =  10000  Average SNR =  -0.34
Bauds =  50000  Average SNR =  -0.35
```

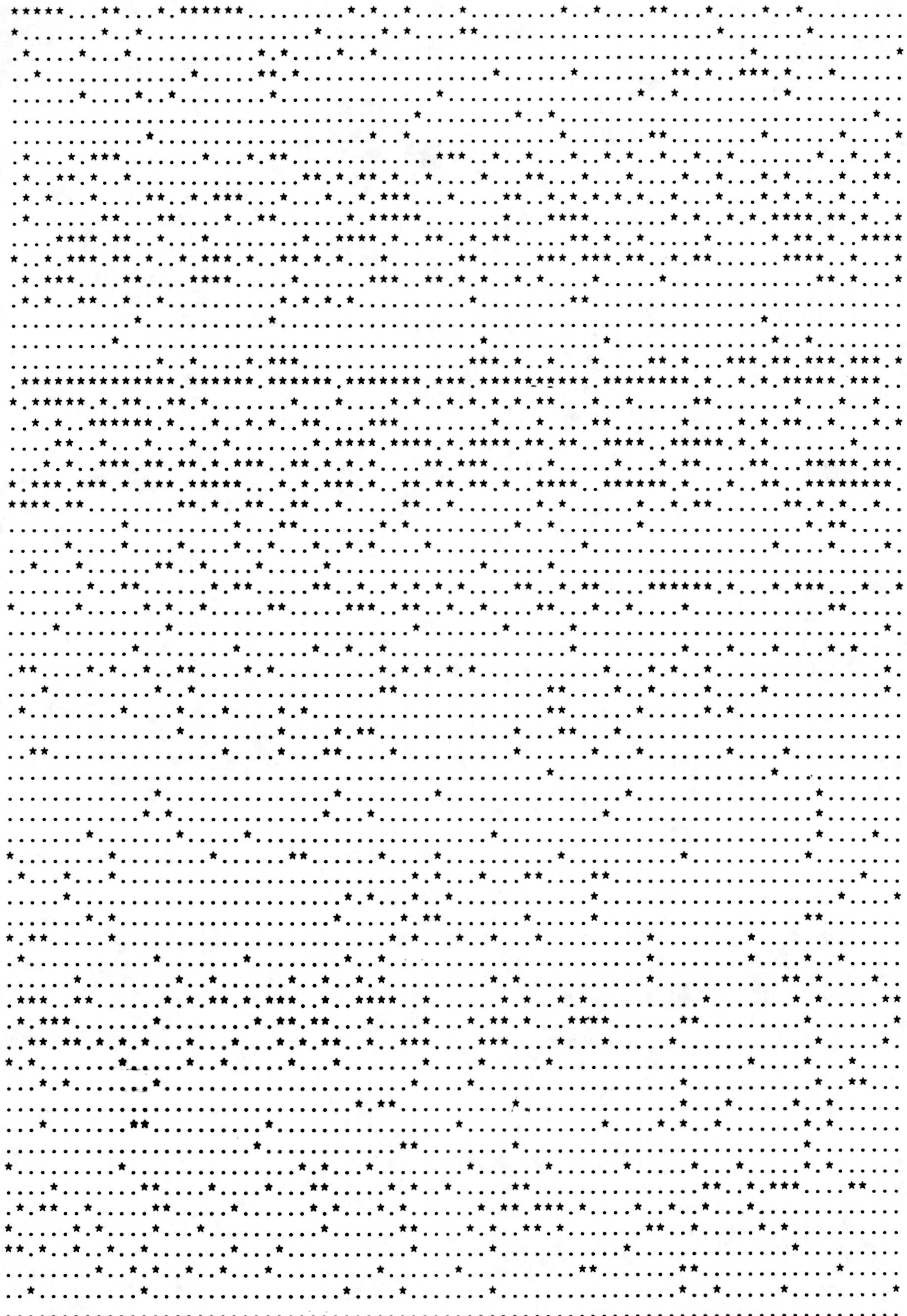
6.2 BER Simulations

The BER simulator produced the results shown on the following pages. The error patterns for the –3 dB simulations are listed to show the characteristics of the simulations. '*' indicates an error in at least one bit of a received tone, while '.' indicates a tone received error-free.

6.2.1 BER — Good Channel

```
SNR (dB):  -3
Enter delay time (ms):  .5
Enter frequency spread (Hz):  .1
Run length:  10000

****.*****.**.*.***.***..*.**.....***.*.*...*.**..*...*.*..*..*..*.........**.*.*
...*...*...........*.....**..................**......**...*...............*.*.*....*
*...*.*............*...*.....*.........*..*..***......*..................**....*.
..*..*.......*..*..*.....***.*......***.***.*.................*.**....**.**.**.****
****.*..***.*******************.******.***.***..*.*.***..**.***.**..*.*******.*
.****..*..***..***.****.*..*...*..**.**.*...**....*..*.......*.*..........***......
.....**.....**..**.....*..*.*..*...**.........*..*..*..*..*..**.........**.**..*...**
**.*.......*..**.*..*.........................*.***.**.....*...*...****.*.**....*..*.*.*
.**.*..*****.*...........*.*........**.......**......*.*.*.*.*.......*.....*....*..**.
*..*...**..*....**.*....***....***..*..*...*....*....*.....*.....*.*.**.**.*.**...
***....*.**.*.****.**.*.*.****..*****.**...****.**..**.********....*.******.***.
.**....****..*...**.*******......*.....*......*..-.*..*.**............................
.....*..*......**...........*.*........*..**...................*.*......................
*..**.....*........*..*........*.*.......*......***..........*....**....*........*...**..
.*.....*......*....*.*.*..........................*......*.......*.....*.......*.*.*...*.*
*.....*.......*..*...**.*...*.*.*............*......*.**..................*.........*..**.
...*..*....*........*.**......**.....................*.*..........*.......*......*....*..*.
...*...*...**..*...***..**.....*..**.*.*..*..*.......*.*.***..**......*.........*..*
.*..**..*.......*.*...**..*.....*.....**.***.*...*....*.*....*.....*.......**........*
.....*.....*.....*....**.*....*..*.................*.*.*........*....*......*....*..*.
.*....*.......*.*........*..*........*.**.**........*....*...........................
...*..................*..................**.....*..........*....*....**.........*.*
.....*.......*.......*.*.......*..........*...............*.......*.*....*..*.*......
..........*...............................................*.........*........*......
.....*.*....**.......*..........*..........*.**...*.*.....*.*.*.*.**........**...
..................*.....*.*.......**..*.......**..*.....*.......*.....**...**..*
**.*.*.*.*.......*.*......*.**......**..*.*.**..*.*...******.......*.*.*.***.**..***
.*.**.*.*.*****.****...*..*..**********.*.*****.**.***..****.********.*.**.***.*.
********.*********.*********.*******.*********....**..**********.**.*.***.**.***
*.*.*.****..*****..*.*.**.*.***....*******..*.***.*.*.**.*.....*.*....*.....**..*.
........*.....*.*...*..........****...*...................*...*..........*.....*..........
...........................................*.......................*....*..........*......
.........................*...........*......*........................**.......*.*............*
*..*.*...................*..*......*.........*...............*....*..................*......
........................................................................*.........
...................*.............*.......*...........*.....*..............*.
.......................................*.....*......*....*......**......*.....*.*......
*..*....*.......**....*...****..**...**.*..*.*..*..........*....*.**.**.*.........*.***
....*.*..*..*.**..*......*..*.**.......**..**....*.*..*....*.*.*.*.**....***.....
*..*...*.....*..........*.....*..........*..............*.**...**..*...**.......*..*..
..*.**.......*.**......*.........*.*..*...*.*.**.........*.*..*.**..*.*..*.*.....*.**..*.
.*.....*....*..*...........*...*...*...*.....*...........................*..........
.....................................*....*..............*...............................
.*..*.............*........*.*......*...*.*..........*..*...*.....*...........*.....*..
.**........................*...*..........*.*..........*..*.....*..........*..........
.*.............................................*...........*.....*....*..*..........
......*.....*................................*.....................*........*.......
............*.........*.........................................................
.........................*.*..**...........................*.........*.....**.....*...*.*..
.........*.*.......*.*.....*.....**....*.******..............*..*.........**.........*...*
***..**.....***.*..****..*..**.*.*.**..*.**.*.*..***..*..***..**...**.**..*..**.
.*....*.*.....*..*.**.*.*..*.*.....*....*..***...**.**.***..***.*...***.****...*
**.*.**.*.*..****.*..*.*..*.****...*....*....***...*..**.**.*.........*....*....*.*
.**......**........*.......*......*........*..*...*.*.............*.......................
.....................**.............*..............................*........
...*.....................*......*.*....................*.........*..........*.*...*..
.........*.....*.....*.........*.........................*.*...........*.......**.........
.......*..*.....................*.......*.....**......*....**..*.............**...........
.*.........*.*............**..........*.....*.*...*......*.*.......*.....**.**.*..**.......
....*******.***..**.***..*.*.**.****.**..****..**.*..*****.****..***...******....
**.********..*.****.***..*.***.-**....****.**.********.***.*.***..*.*.*.*****..**
```

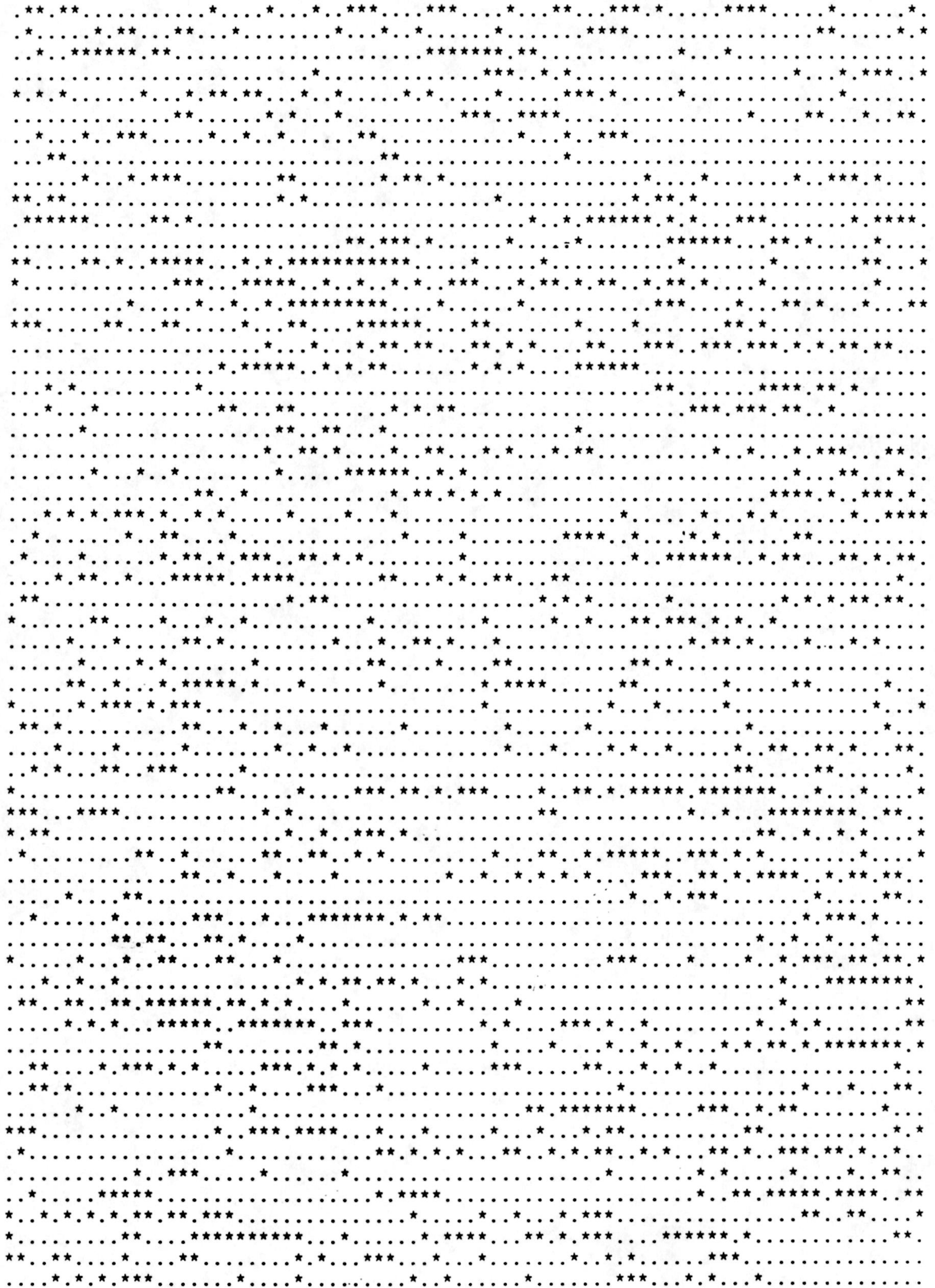


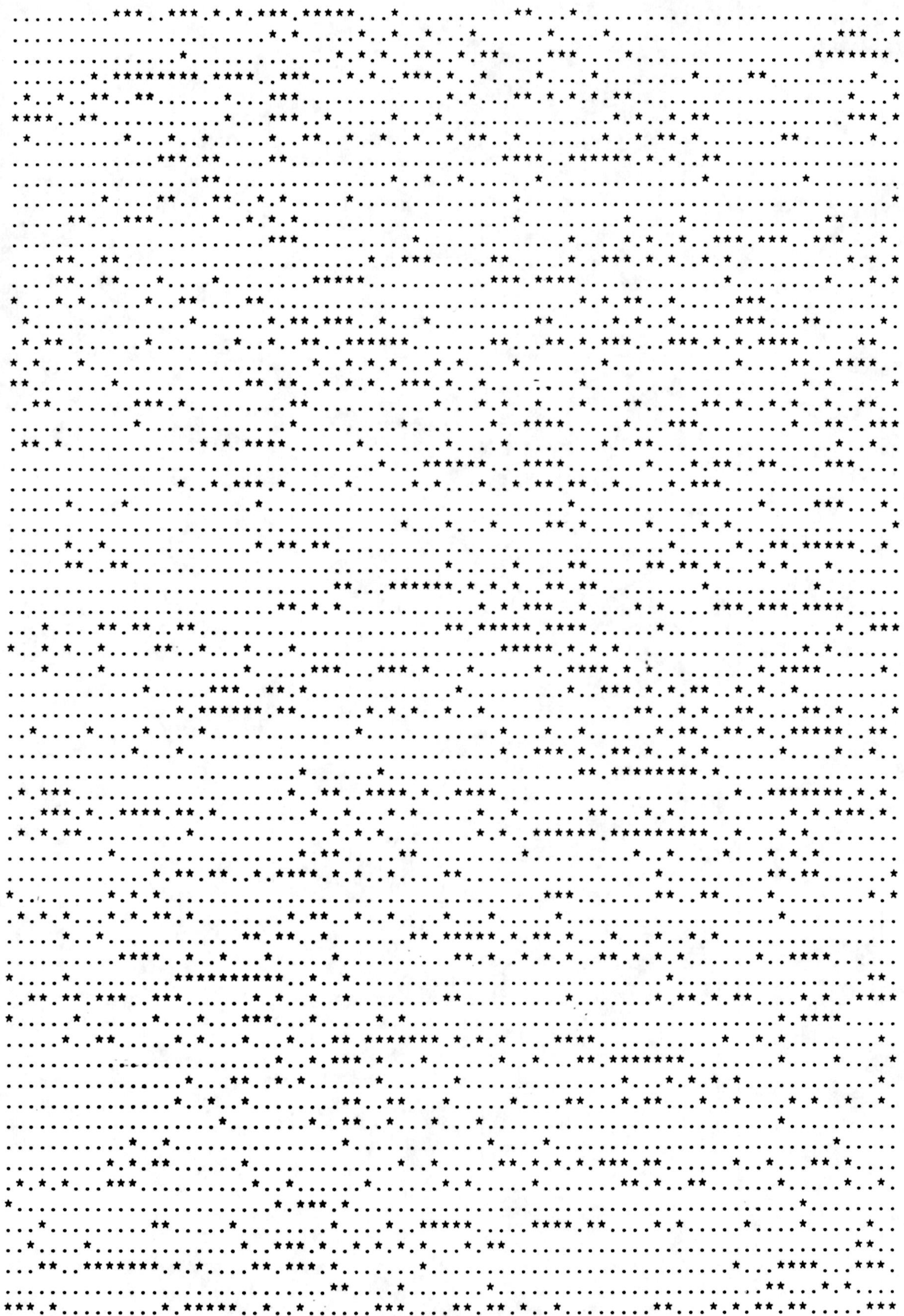
BER = 0.136

At SNR = +3 dB, BER = 0.047 (10,000 tones)
At SNR = +9 dB, BER = 0.015 (10,000 tones)

6.2.2 BER — Poor Channel

```
SNR (dB):  -3
Enter delay time (ms):  2
Enter frequency spread (Hz):  2
Run length:  10000
```





BER = 0.140

At SNR = +3 dB, BER = 0.043 (10,000 tones)
At SNR = +9 dB, BER = 0.011 (10,000 tones)

LISTINGS

```
/*      hflib.c           routines for HF channel simulator    */
/*      eej, rsm, ccw              6/21/91                     */

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

#include "hfdef.h"

#define sqrt2 1.414213562


/***************************** UTILITIES *******************************/

/*  uniform.c       eej          8/12/89                  */
/*  random number generator with uniform distribution     */
/*  from CACM June 1988, Vol 31, No 6, pp. 742-749+       */

double uniform()
{
    static long s1 = 123456787, s2 = 987654323;
    long z, k;

    k = s1 / 53668;
    s1 = 40014 * (s1 - k * 53668) - (k * 12211);
    if (s1 < 0) s1 += 2147483563;

    k = s2 / 52774;
    s2 = 40692 * (s2 - k * 52774) - (k * 3791);
    if (s2 < 0) s2 += 2147483399;

    z = s1 - s2;
    if (z < 1) z += 2147483562;

    return (z * 4.656613E-10);
}

int RandInt(i)  /* Generates random integers in the range 0 to i-1.       */
    int i;      /* In run of 100,000 with 32 bins, had variation of       */
{               /* 6% of mean from max bin to min bin.                    */
    return (i*uniform());
}

int hammingwt3(w)
int w;
{
    static int wt[8] = {0, 1, 1, 2, 1, 2, 2, 3};

    return (wt[w & 7]);
}
```

```
/**********************  DELAY LINE MODULE  ********************************/

static int DelayLineLen;          /* if more than one set of delay lines is  */
static double DlyTF;              /* needed, must pass these as parms        */

static int iptr, qptr;            /* these are passed as parms */
static double IDLine[65], QDLine[65];

double DelayLine(InSig, Ptr, LName)
     double InSig;     /* current sample: put into delay line */
     int *Ptr;         /* current offset into delay line      */
     double *LName;    /* ptr to delay line to use            */
{
    register double *tempptr, temp;
    register int nt;

    if (DelayLineLen > 1) {  /* use the delay line */
        nt = (++(*Ptr)+1) % DelayLineLen;
        tempptr = LName + (*Ptr %= DelayLineLen);
        temp = *tempptr * DlyTF + (*(LName+nt)) * (1.0 - DlyTF);
        *tempptr = InSig;
    }
    else {  /* use only first element of delay line */
        temp = InSig * (1.0 - DlyTF) + *LName * DlyTF;
        *LName = InSig;
    }
    return (temp);
}


/*********************  RAYLEIGH (NOISE) GENERATOR  **************************/

void Rayleigh (x, y)
     double *x, *y;
{
    register double r, a;

    r = sqrt(-2.0*log(uniform()));
    a = TPi * uniform();
    *x = r * cos(a);
    *y = r * sin(a);
}
```

```
/*********************  FADING GAINS MODULE  *********************************/

static double  FreqSpread, g, a0, a1, a2;
static double  IFade0[6], QFade0[6], /* direct-path fading filter state vars */
               IFade1[6], QFade1[6];/* delayed-path fading filter state vars */

void Gauss_Filter(Fade)
     double *Fade;
{
    /* Fade array of input/output data:              */
    /*  Fade[0] is the current filter output         */
    /*  Fade[0-2] are the current and past outputs */
    /*  Fade[3-5] are the current and past inputs   */

    /* Gaussian filter:  2-pole, 2-zero IIR */
    Fade[0]=(g*(Fade[3]+2*Fade[4]+Fade[5])-a1*Fade[1]-a2*Fade[2])/a0;

    /* adjust the history terms */
    Fade[2]=Fade[1];
    Fade[1]=Fade[0];
    Fade[5]=Fade[4];
    Fade[4]=Fade[3];
}

void FadeGains(i, q)
     double *i, *q;
{
    /* generate Rayleigh-distributed fade gain perturbations */
    Rayleigh(i+3, q+3); /* assign to current input positions */

    /* Run through gaussian filter */
    Gauss_Filter(i);
    Gauss_Filter(q);
}

void CompMult (Xi, Xq, Yi, Yq, Zi, Zq)        /*  Z = X * Y     */
     double Xi, Xq, Yi, Yq, *Zi, *Zq;
{
    *Zi = Xi*Yi - Xq*Yq;
    *Zq = Xi*Yq + Xq*Yi;
}
```

```
/*********************  HF CHANNEL SIMULATION  ******************************/

double SigLev = 1.0;             /* global so it can be set by main routine */
static SymVect RcvSig;              /* used to return resulting symbol vector  */

SymVect *HFchan(XmitSig)
     SymVect *XmitSig;
{
    register int i;
    register SymVect *in, *out;
    double   ni, nq, Ri, Rq, RDi, RDq, tmp1, tmp2;
    register double Xi, Xq, Di, Dq;  /* in-phase and quadrature main and     */
                                      /* delayed path samples, respectively */
    in = XmitSig;
    out = RcvSig;                     /* RcvSig is global */
    for (i=0; i<NPTS; i++) {
        Rayleigh(&ni, &nq);  /*                  GENERATE NOISE                */
                             /* Rayleigh generates in-phase and quadrature */
                             /* components, jointly normal, with each      */
                             /* component having RMS amplitude of unity;   */
                             /* RMS noise power is also unity.             */

        if (FreqSpread > 0.0) {          /*      GENERATE FADING GAINS        */
            FadeGains(IFade0, QFade0); /* generate direct-path fade gains  */
            FadeGains(IFade1, QFade1); /* generate delayed-path fade gains */
        }
        else {
            Rayleigh(&tmp1, &tmp2);      /* don't care (to draw RNs)          */
            Rayleigh(&tmp1, &tmp2);      /* leave fade gains at initialized  */
                                         /*   values (phase shift only)       */
        }

        if (DelayLineLen + DlyTF > 1.0) {          /*  multipath?  */
            Di = DelayLine(Xi=SigLev*(*in)[0][i], &iptr, IDLine);/* generate */
            Dq = DelayLine(Xq=SigLev*(*in)[1][i], &qptr, QDLine);/* delayed  */
                                                                /* signals  */

           CompMult(Xi, Xq, *IFade0, *QFade0, &Ri, &Rq);/* introduce fading */
            CompMult(Di, Dq, *IFade1, *QFade1, &RDi, &RDq);
        }
        else {                        /* all power in single path */
            CompMult((*in)[0][i], (*in)[1][i], *IFade0, *QFade0, &Ri, &Rq);
            RDi = RDq = 0.0;
            Ri *= sqrt2;
            Rq *= sqrt2;
        }

        (*out)[0][i] = Ri + RDi + ni;     /* compute output */
        (*out)[1][i] = Rq + RDq + nq;
    }
    return (out);
}
```

```
/**********************  INITIALIZATION ************************************/

void Initialize()
{
    register int i;
    double DlyTime, a, c, A, C, dt;

    /******************** SET UP DELAY LINES *********************/
    do {
      printf("Enter delay time (ms):  ");
      scanf("%lf", &DlyTime);
    } while (DlyTime < 0.0);

    dt = Ts/NPTS;              /*  dt = sample period  */
    DlyTime /= (dt * 1e3);   /* scale DlyTime from milliseconds to samples */
    DlyTF = DlyTime - (DelayLineLen = (int)DlyTime);
    DelayLineLen++;
    iptr = qptr = 0;
    for (i=0; i<65; i++) IDLine[i] = QDLine[i] = 0.0;


    /*****************  SET UP FADING GENERATOR  *****************/
    do {
      printf("Enter frequency spread (Hz):  ");
      scanf("%lf", &FreqSpread);
    } while (FreqSpread < 0.0);

    if (FreqSpread > 0.0) {
      FreqSpread *= dt*TPi;    /*  scale from Hz to radians per sample  */

      /* magic numbers for Gaussian filter */
      a = sqrt(TPi);
      c = 1.5;
      A = a/FreqSpread;
      C = c/FreqSpread;     C *= C;

      /* Gaussian filter coefficients */
      g = sqrt(0.5*sqrt(TPi)/FreqSpread);
      a0 = A + C + 1.0;
      a1 = 2*(1.0 - C);
      a2 = C + 1.0 - A;

      for (i=0; i<6; i++)  /* clear fading filter state variables . . . */
        *(IFade0+i) = *(QFade0+i) = *(IFade1+i) = *(QFade1+i) = 0.0;

      for (i=0; i<1.0/FreqSpread; i++) {         /* and initialize them */
        FadeGains(IFade0, QFade0);
        FadeGains(IFade1, QFade1);
      }
    }
    else           /* fading generator disabled; set fixed gains for taps */
      *(IFade0) = *(QFade0) = *(IFade1) = *(QFade1) = sqrt2/2.0;
}
```

```
/*  ALEmodem.c          eej       6/13/91          */
/*          8-ary FSK modem for HF ALE simulator */

#include <math.h>
#include "hfdef.h"
#define m_ary 8

static double tone[m_ary][2][NPTS];

void InitModem()
{
  register int i, iprime, j;
  double f[m_ary], dt;
  register double x, dx;

  f[0] = -(f[7] = 875.0);        /* 8-ary FSK tone frequencies */
  f[1] = -(f[6] = 625.0);
  f[2] = -(f[5] = 375.0);
  f[3] = -(f[4] = 125.0);

  dt = Ts/NPTS;

  for (i=0; i<m_ary/2; i++) {    /* generate tone vectors    */
      iprime = m_ary-i-1;        /* index of conjugate frequency */
      x = 0.0;
      dx = TPi*f[iprime]*dt;
      for (j=0; j<NPTS; j++) {
          tone[i][0][j] = tone[iprime][0][j] = cos(x);  /* real part of tone */
          tone[i][1][j] = -(tone[iprime][1][j] = sin(x));/* imag part of tone*/
          x += dx;      if (x >= TPi) x -= TPi;
      }
  }
}

SymVect *FSKmod(InSym)              /*  use: SymVect *XmitSig         */
     int InSym;                     /*       XmitSig = FSKmod(InSym)  */
{                                   /*           ... (*XmitSig)[i][j] */
  return (tone[InSym]);
}

int FSKdemod(RcvSig)
     SymVect *RcvSig;
{
  register int i, j, out;
  register double sumi, sumq, sum, max=0.0;
  register SymVect *in;

  in = RcvSig;
  for (i=0; i<m_ary; i++)  {
      sumi = sumq = 0.0;
      for (j=NPTS/4; j<(3*NPTS)/4; j++)  {
          sumi += (*in)[0][j]*tone[i][0][j] + (*in)[1][j]*tone[i][1][j];
          sumq += (*in)[1][j]*tone[i][0][j] - (*in)[0][j]*tone[i][1][j];
      }
      if ((sum = sumi*sumi + sumq*sumq) >= max) {
          max = sum;
          out = i;
      }
  }
  return (out);
}
```

```
/* BER.c        eej          6/16/91           */
/*              BER simulator using hflib      */
/*                    and ALEmodem             */

#include <stdio.h>

#include <math.h>

#include "../hfdef.h"
#include "../hflib.h"
#include "../ALEmodem.h"

main()
{
  long i, insym, outsym, RunLength, errors=0;

  printf("SNR (dB):  ");
  scanf("%lf", &SigLev);
  SigLev = pow(10.0, SigLev/20.0); /* convert from dB to voltage ratio */

  Initialize();
  InitModem();

  printf("Run length:  ");
  scanf("%ld", &RunLength);
  printf("\n");

  for (i=0; i<RunLength; i++)
    if ((outsym = ALEmodem(insym = RandInt(8))) != insym) {
      printf("*");
      errors += hammingwt3(insym ^ outsym);
    }
    else printf(".");

  printf("\nBER = %5.3lf\n", (double)errors/(3.0 * RunLength));
}
```

```
/* hfdef.h            eej           6/16/91                  */
/*       definitions for HF channel simulator routines    */

#define Pi  3.141592654
#define TPi 6.283185307

#define Ts  0.008
#define NPTS 32

typedef double  SymVect[2][NPTS];

extern double SigLev;       /* declared in hflib.c */
```

```
/*  hflib.h     header file for HF propagation simulator routines  */
/*      eej     6/16/91                                            */

/* hfdef.h must be #included before this file */

extern double   uniform();
extern int      RandInt();
extern int      hammingwt3();

extern void     Initialize();
extern SymVect  *HFchan();
```

```
/*  ALEmodem.h        eej           6/13/91             */
/*             8-ary FSK modem for HF ALE simulator     */

extern void     InitModem();
extern SymVect *FSKmod();
extern int      FSKdemod();

#define ALEmodem(x) FSKdemod(HFchan(FSKmod(x)))
```