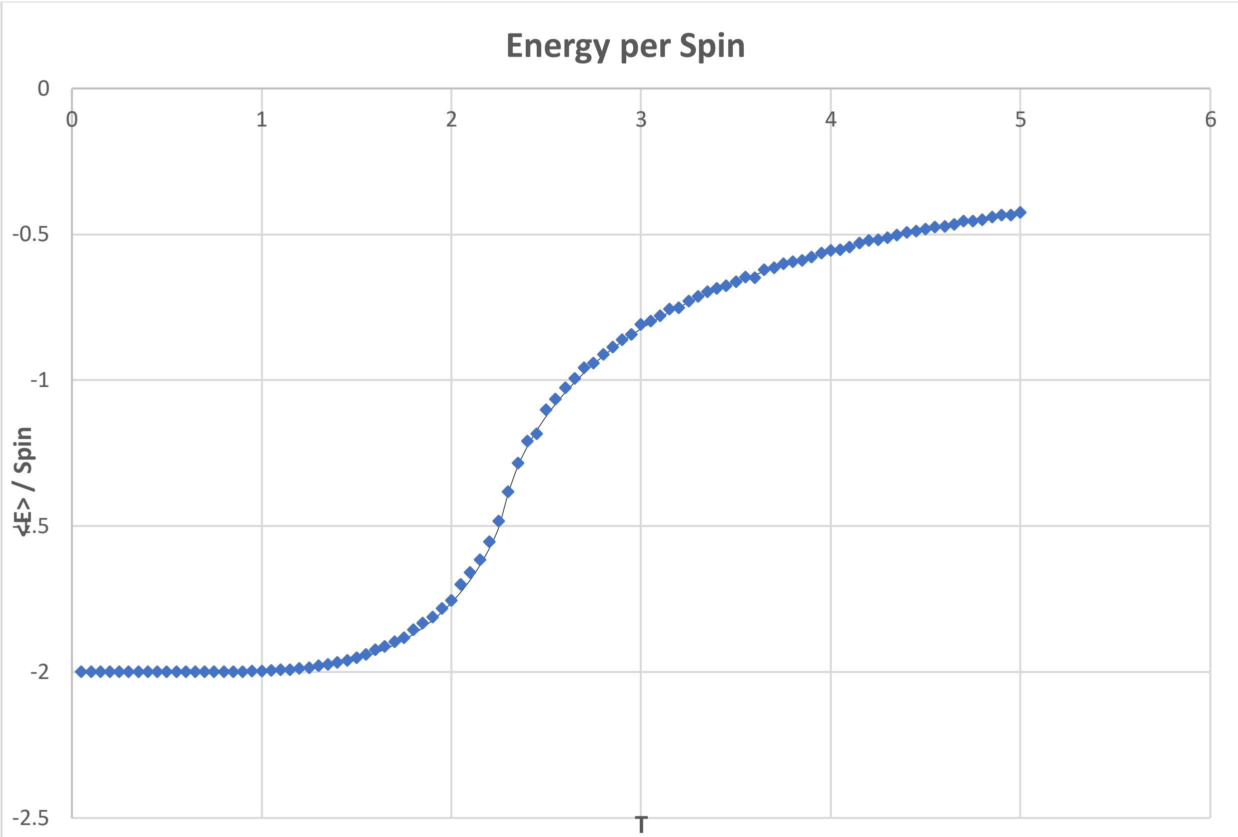
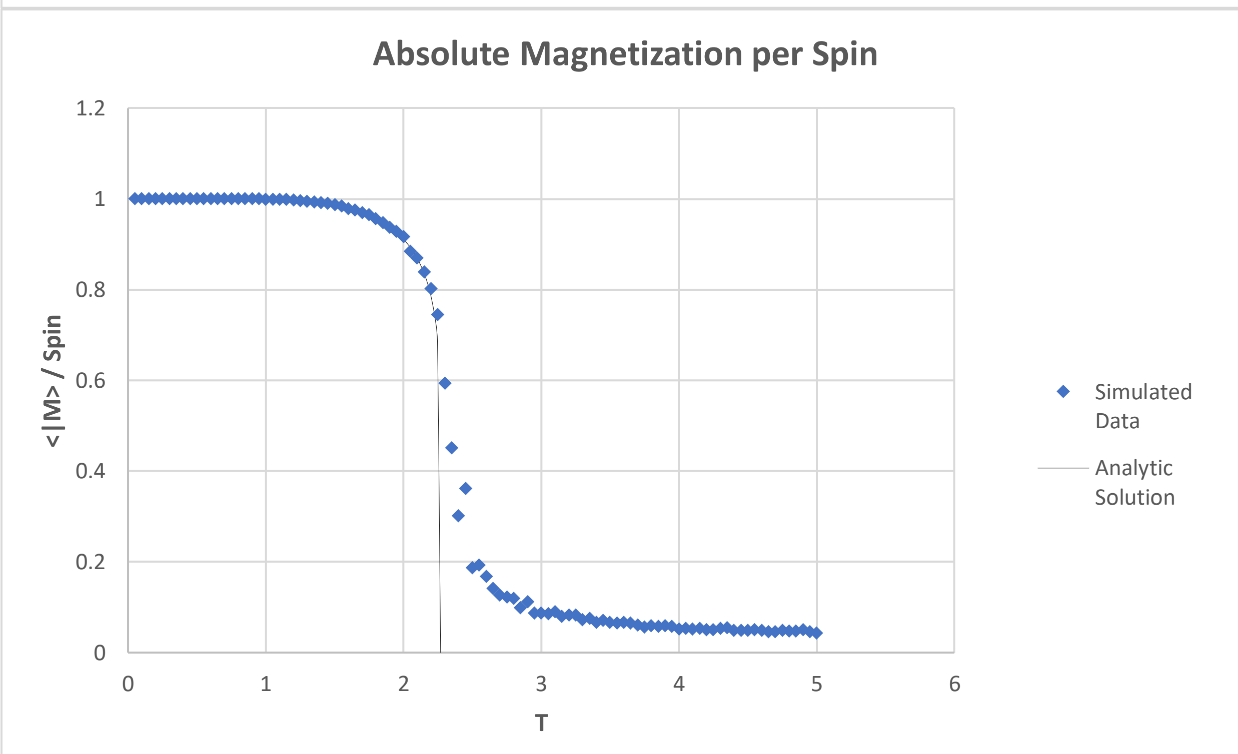
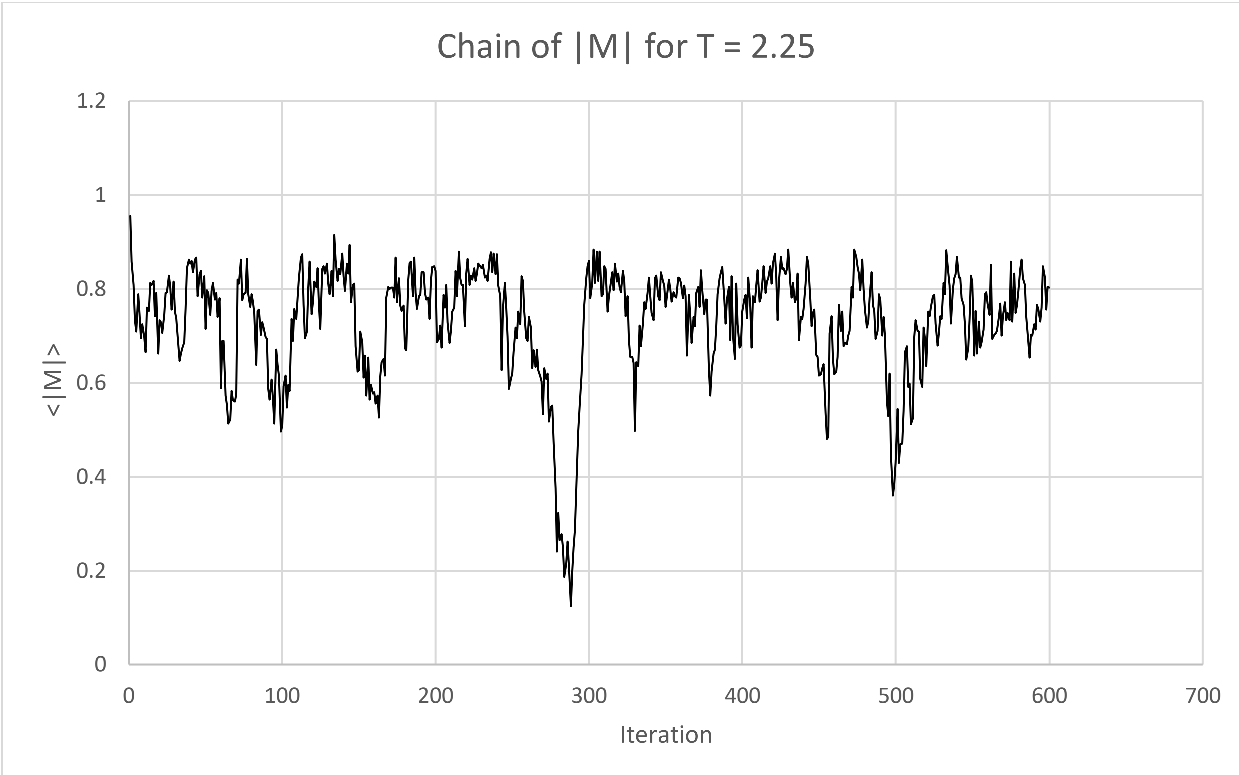
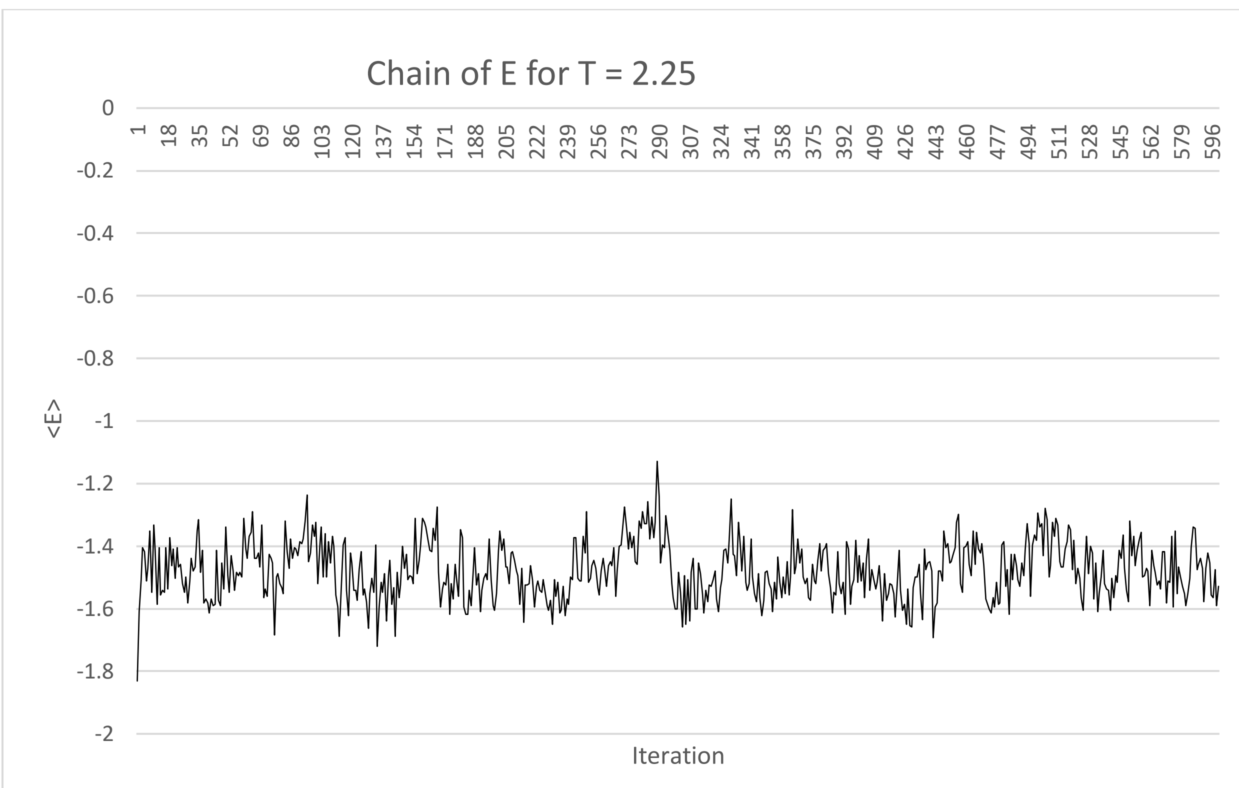

Energy per Spin
<E> / Spin
T
0
-0.5
-1
-1.5
-2
-2.5
0 1 2 3 4 5 6


Absolute Magnetization per Spin
<|M> / Spin
T
1.2
1
0.8
0.6
0.4
0.2
0
0 1 2 3 4 5 6
Simulated Data
Analytic Solution





Driver.c: Main Driver for the Program

```c
#include "MCMC.h"
#include <unistd.h>

int main() {

  // Simulation parameters and values
  int nchains, nburnin, niters, nthin, nrows, ncols, Mag, Ham, J, ntemps,
magsum, hamsum, temp, i, j, dE;
  double *tempPtr, spacing, currenttemp, magensemble, hamensemble, prob,
rand;
  // Output for final results
  FILE *resultsfile = fopen("ResultsFile.csv", "w");;

  // Seed random generator and set values
  sgenrand(time(0));

  nburnin = 3000;
  niters = 3000;
  nthin = 10;
  nrows = 30;
  ncols = 30;
  J = 1;
  ntemps = 100;

  // Declare and populate array for simulation temperatures
  tempPtr = calloc(ntemps, sizeof(double ) );

  spacing = (double)5/(double)ntemps;

  for( int n = 0 ; n < ntemps ; n++ ){
    double *pos = tempPtr + n;
    *pos = spacing * (double)(n+1);
  }


  // Perform MCMC simulation for each temperature
  for( int temp = 0 ; temp < ntemps ; temp++ ){

    // Declare array for lattice
    int arrPtr [nrows][ncols];
    currenttemp = *(tempPtr + temp);

    // Output files for results of burn in period and sampling period
    char burnbuf[0x100], samplebuf[0x100];
    snprintf(burnbuf, sizeof(burnbuf), "BurnFile%f.csv", currenttemp);
    snprintf(samplebuf, sizeof(samplebuf), "SampleFile%f.csv", currenttemp);
    FILE *burnfile = fopen(burnbuf, "w");
    FILE *samplefile = fopen(samplebuf, "w");

    printf("Simulating temperature %f ...\n", currenttemp);

    // Randomize initial configuration of lattice
    randomizeLattice( (int *) arrPtr, nrows, ncols, 1);

    Mag = totalMagnetization( (int *) arrPtr, nrows, ncols );
    Ham = totalHamiltonian( (int *) arrPtr, nrows, ncols, J );
```

```c
    // Burn in phase
    printf("Burn in...\n");

    // Compute nburnin MCMC steps
    for( int burn = 0 ; burn < nburnin ; burn++ ){
      // Explore nrows-by-ncols dimensional space
        for ( int dim = 0 ; dim < nrows * ncols ; dim++ ){

            // Randomly select one dimension and calculate change in
Hamiltonian if state flipped
            i = floor(genrand() * (double) nrows);
            j = floor(genrand() * (double) ncols);
            dE = dESpin( (int *) arrPtr, i, j, nrows, ncols, J );

            // Compute transition probability and update state
            prob = transitionProbability(dE, currenttemp);
            if ( prob > genrand() ){
              flipSpin( (int *) arrPtr, i, j, nrows, ncols);
              Mag = Mag + 2 * *((int *) arrPtr + i * ncols + j);
              Ham = Ham + dE;
            }
        }
    }

      // Accept every nthin-th MCMC step
      if ( modulo( burn, nthin) == 0) fprintf(burnfile, "%d,%d,%d,\n",
burn/nthin, Mag, Ham);

    }

    fclose(burnfile);
    printf("Burn in finished...\n\n");

    // Print lattice after burn in period
    //printLattice((int *) arrPtr, nrows, ncols);

    // Sampling period
    printf("Sampling...\n");

    // Initialize statistical measures
    magsum = 0;
    hamsum = 0;
    hamensemble = 0;
    magensemble = 0;

    // Compute niters MCMC steps
    for ( int step = 0 ; step < niters ; step++ ){

      // Explore nrows-by-ncols dimensional space
        for( int dim = 0 ; dim < nrows * ncols ; dim++ ){
            // Randomly select one dimension and calculate change in
Hamiltonian if state flipped
            i = floor(genrand() * (double) nrows);
            j = floor(genrand() * (double) ncols);
            dE = dESpin( (int *) arrPtr, i, j, nrows, ncols, J);

            // Compute transition probability and update state
```

```
            prob = transitionProbability(dE, currenttemp);
            if ( prob > genrand() ){
              flipSpin( (int *) arrPtr, i, j, nrows, ncols);
              Mag = Mag + 2 * *((int *) arrPtr + i * ncols + j);
              Ham = Ham + dE ;
            }
        }

        // Accept nthin-th accept MCMC step
        if( modulo(step, nthin) == 0 ){
            fprintf(samplefile, "%d,%d,%d,\n", step/nthin, Mag, Ham);
            hamsum += Ham;
            magsum += abs(Mag);
        }
    }

    // Compute summary statistics
    hamensemble =  (double) hamsum / (double) ( niters / nthin );
    magensemble = (double) magsum / (double) ( niters / nthin );

    // Write out to results file
    fprintf(resultsfile, "%f,%f,%f,%f,%f,\n", currenttemp, hamensemble,
magensemble, hamensemble / (double) (nrows*ncols), magensemble / (double)
(nrows*ncols));

    // Print final lattice
    printLattice( (int *) arrPtr, nrows, ncols);

    printf("Complete\n\n");
    fclose(samplefile);
  }
  fclose(resultsfile);
  return 0;
}
```

MCMC.c: Implementation of MCMC Related Functions for the Model

```
#include "MCMC.h"
```

```
void randomizeLattice(arrPtr p, int n, int m) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {

      double r = (double) genrand();
      if ( r < 0.5 ) *(( p + i * m) + j) = 1;
      else *(( p + i*m ) + j ) = -1;
    }
  }
}

int totalHamiltonian(arrPtr p, int n, int m, int J){
  int total = 0;
  int spinij, spinijright, spinijbottom;
  for ( int i = 0 ; i < n ; i++ ){
    for ( int j = 0 ; j < m ; j++ ){
      spinij = *( p + i*m + j);
      spinijright = *( p + i*m + modulo( j + 1, m ) );
      spinijbottom = *( p + modulo( i * m + j + m, n*m ));
      total += spinij * ( spinijright + spinijbottom);
    }
  }
  return (total * -1 * J);
}

int totalMagnetization(arrPtr p, int n, int m){
  int total = 0;
  for ( int i = 0 ; i < n ; i++ ){
    for ( int j = 0 ; j < m; j++ ){
      total +=  *(p + i * m + j);
    }
  }
  return total;
}

double transitionProbability(int dE, double t) {

  double exponent =  (double) (-1 * dE) / (double) (KB*t) ;
  double numer = powf(E, exponent);
  double denom = 1 + powf(E, exponent);
  double prob = numer / denom;
  return prob;
}

int dESpin(arrPtr p, int i, int j, int n, int m, int J) {


  int topshift, bottomshift, rightshift, leftshift;

  int top, bottom, right, left;

  int sum;


  int spinij = *( p + i * m  + j);
```

```
  topshift = modulo( (i * m + j - m), ( n*m ));
  bottomshift = modulo( (i * m + j + m), ( n*m ));
  rightshift = ( i * m ) + modulo( j + 1 , m );
  leftshift = ( i * m ) + modulo( j - 1, m );

  top = *(p + topshift);
  bottom = *(p + bottomshift);
  right = *(p + rightshift);
  left = *(p + leftshift);

  return -2 * spinij * (top + bottom + right + left );
}

void flipSpin( arrPtr p, int i, int j, int n, int m){
  int *spinij = ( p + i * m + j);
  *spinij = -1 * *(spinij);
}

void printLattice(arrPtr p, int n, int m) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      printf("%3d", *( p + i * m  + j ));
    }
    printf("\n");
  }
  printf("\n");
}

int modulo(int a, int n){
  int mod = a % n;
  if ( a < 0 ){
    mod += n;
  }
  return mod;
}
```

<u>MCMC.h: Header File for All Functions</u>

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

```c
#include <stdlib.h>
#include <math.h>
#include <time.h>

/* MCMC parameters */
#define E 2.718281828
#define KB 1

typedef int nchains;
typedef int nburnin;
typedef int niters;
typedef int nthin;
typedef int nrows;
typedef int ncols;
typedef int J;
typedef int ntemps;
typedef int *arrPtr;

/* Mersenne twister parameters */
/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0df   /* constant vector a */
#define UPPER_MASK 0x80000000 /* most significant w-r bits */
#define LOWER_MASK 0x7fffffff /* least significant r bits */

/* Tempering parameters */
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y)  (y >> 11)
#define TEMPERING_SHIFT_S(y)  (y << 7)
#define TEMPERING_SHIFT_T(y)  (y << 15)
#define TEMPERING_SHIFT_L(y)  (y >> 18)
```

```c
/* Random generator seed */
typedef unsigned long seed;


/* MCMC functions */
void randomizeLattice(arrPtr p, int n, int m, int nonrandom);
int totalHamiltonian(arrPtr p, int n, int m, int J);
int totalMagnetization(arrPtr p, int n, int m);
double transitionProbability(int dE, double t);
void flipSpin(arrPtr p, int i, int j, int n, int m);
int dESpin(arrPtr p, int i, int j, int n, int m, int J);
int modulo( int a, int n);
void printLattice(arrPtr p, int n, int m);

/* Mersenne twister functions */
void sgenrand(seed s);
double genrand();
```

MT19937.c: MERSENNE TWISTER CODE

```c
/* A C-program for MT19937: Real number version  (1998/4/6)    */
```

```c
/*   genrand() generates one pseudorandom real number (double) */
/* which is uniformly distributed on [0,1]-interval, for each  */
/* call. sgenrand(seed) set initial values to the working area */
/* of 624 words. Before genrand(), sgenrand(seed) must be      */
/* called once. (seed is any 32-bit integer except for 0).     */
/* Integer generator is obtained by modifying two lines.       */
/*   Coded by Takuji Nishimura, considering the suggestions by */
/* Topher Cooper and Marc Rieffel in July-Aug. 1997.           */

/* This library is free software; you can redistribute it and/or   */
/* modify it under the terms of the GNU Library General Public      */
/* License as published by the Free Software Foundation; either     */
/* version 2 of the License, or (at your option) any later          */
/* version.                                                         */
/* This library is distributed in the hope that it will be useful, */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of  */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.             */
/* See the GNU Library General Public License for more details.    */
/* You should have received a copy of the GNU Library General      */
/* Public License along with this library; if not, write to the    */
/* Free Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA   */
/* 02111-1307  USA                                                  */

/* Copyright (C) 1997 Makoto Matsumoto and Takuji Nishimura.       */
/* When you use this, send an email to: matumoto@math.keio.ac.jp   */
/* with an appropriate reference to your work.                      */

/* REFERENCE                                                        */
/* M. Matsumoto and T. Nishimura,                                   */
/* "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform  */
/* Pseudo-Random Number Generator",                                 */
/* ACM Transactions on Modeling and Computer Simulation,           */
/* Vol. 8, No. 1, January 1998, pp 3--30.                          */

#include<stdio.h>

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0df   /* constant vector a */
#define UPPER_MASK 0x80000000 /* most significant w-r bits */
#define LOWER_MASK 0x7fffffff /* least significant r bits */

/* Tempering parameters */
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y)  (y >> 11)
#define TEMPERING_SHIFT_S(y)  (y << 7)
#define TEMPERING_SHIFT_T(y)  (y << 15)
#define TEMPERING_SHIFT_L(y)  (y >> 18)

static unsigned long mt[N]; /* the array for the state vector  */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */

/* initializing the array with a NONZERO seed */
void
sgenrand(seed)
```

```c
    unsigned long seed;
{
    /* setting initial seeds to mt[N] using         */
    /* the generator Line 25 of Table 1 in          */
    /* [KNUTH 1981, The Art of Computer Programming */
    /*    Vol. 2 (2nd Ed.), pp102]                  */
    mt[0]= seed & 0xffffffff;
    for (mti=1; mti<N; mti++)
        mt[mti] = (69069 * mt[mti-1]) & 0xffffffff;
}

double /* generating reals */
/* unsigned long */ /* for integer generation */
genrand()
{
    unsigned long y;
    static unsigned long mag01[2]={0x0, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if sgenrand() has not been called, */
            sgenrand(4357); /* a default initial seed is used   */

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1];

        mti = 0;
    }

    y = mt[mti++];
    y ^= TEMPERING_SHIFT_U(y);
    y ^= TEMPERING_SHIFT_S(y) & TEMPERING_MASK_B;
    y ^= TEMPERING_SHIFT_T(y) & TEMPERING_MASK_C;
    y ^= TEMPERING_SHIFT_L(y);

    return ( (double)y * 2.3283064370807974e-10 ); /* reals */
    /* return y; */ /* for integer generation */
}

/* this main() outputs first 1000 generated numbers  */
/*main()
 *{
 *   int j;

 *   sgenrand(4357); /* any nonzero integer can be used as a seed */
/*   for (j=0; j<1000; j++) {
 *       printf("%10.8f ", genrand());
```

```
*       if (j%8==7) printf("\n");
*   }
*   printf("\n");
}*/
```