```
/*********************************************************************/
/*** Markov chain for Bernoulli distributed vectors               ***/
/*** Lectures in computational physics                            ***/
/*** University of Oldenburg, Germany 2024                       ***/
/*** compile: cc -o mc_bernoulli mc_bernoulli.c -lm              ***/
/*********************************************************************/

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

/*************** bernoulli_mc_step0() ********************/
/** Perform an MC step for Binomial distribution with   **/
/** changes directly distributed according to Binomial  **/
/** PARAMETERS: (*)= return-paramter                    **/
/**           n: number of coin tosses                  **/
/**           y: state vector                           **/
/**       alpha: "1" probability                        **/
/** num_changes: ... to state vector y                  **/
/** RETURNS:                                            **/
/**             (nothing)                               **/
/*************************************************************/
void  bernoulli_mc_step0(int n, double *y, double alpha, int num_changes)
{
  int change;                                  /* change counter */
  int pos;                                    /* where to change */

  for(change=0; change<num_changes; change++)      /* several changes */
  {
    pos = (int) floor(n*drand48());     /* select change position, for */
                          /* simplicity, allow for double changes */
    if(drand48() < alpha)                  /* redraw state at positon */
      y[pos] = 1;
    else
      y[pos] = 0;
  }
}


/********** bernoulli_mc_step_bias() ********************/
/** Perform an MC step for Binomial distribution with   **/
/** changes directly distributed according to Bernoulli **/
/** plus exponential bias exp(-S/theta)                 **/
/** PARAMETERS: (*)= return-paramter                    **/
/**           n: number of coin tosses                  **/
/**           y: state vector                           **/
/**       alpha: "1" probability                        **/
/** num_changes: ... to state vector y                  **/
/**       theta: temperature                            **/
/**      calc_S: function which calculates S            **/
/** RETURNS:                                            **/
/**             1 if accepted move, 0 else              **/
/*************************************************************/
int  bernoulli_mc_step_bias(int n, double *y, double alpha, int num_changes,
                            double theta, double (* calc_S)(int n, double *y))
{
  int change;                                     /* change counter */
  int pos;                                       /* where to change */
  int S_old, S_new;                        /* current/trial "energy" values */
  int *change_pos;                  /* remember which entries were changed */
  double *change_val;                             /* remember old entries */
  double prob;                        /* Metropolis-Hastings probability */
  int accept;                                    /* is step accepted ? */

  change_pos = (int *) malloc(num_changes*sizeof(int));
  change_val = (double *) malloc(num_changes*sizeof(double));

  S_old = calc_S(n, y);
  for(change=0; change<num_changes; change++)          /* several changes */
  {
    pos = (int) floor(n*drand48());        /* select change position, for */
                            /* simplicity, allow for double changes */

    change_pos[change] = pos;                   /* remember previous value */
    change_val[change] = y[pos];
    if(drand48() < alpha)                       /* redraw state at positon */
      y[pos] = 1;
    else
      y[pos] = 0;
  }
  S_new = calc_S(n, y);

  prob = exp(-(S_new-S_old)/theta);
  if(drand48() > prob)                                  /* reject ? */
```

```
  {
    accept = 0;
    for(change=num_changes-1; change >=0; change--)     /* reverse order! */
      y[change_pos[change]] = change_val[change];
  }
  else
    accept = 1;

  free(change_pos);
  free(change_val);
  return(accept);
}



/*************** bernoulli_metropolis() ******************/
/** Perform an Metropilis step for Bernoulli system.    **/
/** Accept a reverse of an entry with Metropolis prob.  **/
/** PARAMETERS: (*)= return-paramter                    **/
/**           n: number of coin tosses                  **/
/**           y: state vector                           **/
/**       alpha: "1" probability                        **/
/** RETURNS:                                            **/
/**   1 if flip accepted, 0 else                        **/
/*********************************************************/
int bernoulli_metropolis(int n, double *y, double alpha)
{
  double prob;                                /* Metropolis probability */
  int pos;                                    /* where to change */

  pos = (int) floor(n*drand48());             /* select change position */

  prob = pow(alpha, 1-2*y[pos])*pow(1-alpha, 2*y[pos]-1);
  if(prob > 1)
    prob = 1;                   /* actually not needed, for completeness */

  if(drand48() < prob)                        /* flip variable ? */
  {
    y[pos] = 1 - y[pos] ;
    return(1);
  }
  else
    return(0);
}


/****************** bernoulli_count() *********************/
/** Counter number of 1s in state vector x              **/
/** PARAMETERS: (*)= return-paramter                    **/
/**           n: number of coin tosses                  **/
/**           x: state vector                           **/
/** RETURNS:                                            **/
/**         number of 1s (as double to fit code)        **/
/*********************************************************/
double bernoulli_count(int n, double *x)
{
  int t, count;

  count = 0;
  for(t=0; t<n; t++)
    if(x[t] == 1)
      count++;

  return(count);
}


/****************** bernoulli_S3plus() *******************/
/** Counter number of squences 111+ in state vector x   **/
/** PARAMETERS: (*)= return-paramter                    **/
/**           n: number of coin tosses                  **/
/**           x: state vector                           **/
/** RETURNS:                                            **/
/**         number of 1s (as double to fit code)        **/
/*********************************************************/
double bernoulli_S3plus(int n, double *x)
{
  int t, count, loc_count;

  count = 0; loc_count = 0;
  for(t=0; t<n; t++)
    if(x[t] == 1)
      loc_count++;
    else
```

```
    {
      if(loc_count >= 3)
        count++;
      loc_count = 0;
    }
  if(loc_count >= 3)
        count++;


  /*for(t=0; t<n; t++)
    printf("%d", (int) x[t]);
    printf(" S3+: %d\n", count);*/

  return(count);
}

/****************** bernoulli_largest() ******************/
/** Determines largest 111.. sequence in state vector x **/
/** PARAMETERS: (*)= return-paramter                    **/
/**           n: number of coin tosses                  **/
/**           x: state vector                           **/
/** RETURNS:                                            **/
/**       size of of largest 1s cluster (as double)     **/
/*************************************************************/
double bernoulli_largest(int n, double *x)
{
  int t, largest_count, loc_count;

  largest_count = 0; loc_count = 0;
  for(t=0; t<n; t++)
    if(x[t] == 1)
      loc_count++;
    else
    {
      if(loc_count > largest_count)
        largest_count = loc_count;
      loc_count = 0;
    }

  if(loc_count > largest_count)
    largest_count = loc_count;

  /*for(t=0; t<n; t++)
    printf("%d", (int) x[t]);
    printf(" largest: %d\n", largest_count);*/

  return(largest_count);
}


int main(int argc, char *argv[])
{
  int n;                                    /* number of values */
  double *x;                                /* actual values */
  double init_val;                          /* initial value for x entries */
  int num_bins;                             /* number of bins */
  double *histo;                            /* histogram */
  int num_entries;                          /* ... in histo */
  double start_histo, end_histo;            /* range of histogram */
  double delta;                             /* width of bin */
  int bin;                                  /* ID of current bin */
  int t;                                    /* loop counters */
  int num_chains;                           /* number of generated MC chains */
  int chain;                                /* MC chain counter */
  int num_steps, step;                      /* number of MC steps, counter */
  int num_changes;                          /* per MC step */
  int do_alg;                               /* which algorithm */
  double *data;                             /* time series */
  double alpha;                             /* parameter of distribution */
  double value;                             /* generated number */
  double theta;                             /* temperature for bias */
  int do_histo, num_equi;       /* measure histogram? After num_equi steps */
  int argz = 1;                             /* argument counter */
  double (* calc_S)(int, double *);         /* ptr to function to obtain S */
  double num_accept;                        /* number of accepted steps */

  num_chains = 1; num_changes = 1; do_histo=0; num_equi=0; init_val = 0;
  do_alg = 0; theta = 1e8; calc_S = bernoulli_count;

  while((argz<argc)&&(argv[argz][0] == '-'))
  {
    if(strcmp(argv[argz], "-init") == 0)
    {
      sscanf(argv[++argz], "%lf", &init_val);
    }
```

```
  else if(strcmp(argv[argz], "-change") == 0)
  {
    sscanf(argv[++argz], "%d", &num_changes);
  }
  else if(strcmp(argv[argz], "-runs") == 0)
  {
    sscanf(argv[++argz], "%d", &num_chains);
  }
  else if(strcmp(argv[argz], "-theta") == 0)
  {
    sscanf(argv[++argz], "%lf", &theta);
  }
  else if(strcmp(argv[argz], "-alg") == 0)
  {
    sscanf(argv[++argz], "%d", &do_alg);
  }
  else if(strcmp(argv[argz], "-S3+") == 0)
  {
    calc_S = bernoulli_S3plus;
  }
  else if(strcmp(argv[argz], "-S_l") == 0)
  {
    calc_S = bernoulli_largest;
  }
  else if(strcmp(argv[argz], "-histo") == 0)
  {
    do_histo = 1;
    sscanf(argv[++argz], "%d", &num_equi);
  }
  else
  {
    fprintf(stderr, "unkown option: %s\n", argv[argz]);
    exit(1);
  }
  argz++;
}


if(argc-argz < 3)
{
  fprintf(stderr, "USAGE: %s <n> <alpha> <num_mc_steps>\n", argv[0]);
  fprintf(stderr, "   options: -init <x> : initial value (d:%d)\n",
          (int) init_val);
  fprintf(stderr, "          -change <c> : number of changes entries per set"
          " (d: %d)\n", num_changes);
  fprintf(stderr, "          -theta <T> : set temperature (for biased)"
          " (d: %2.3e)\n", theta);
  fprintf(stderr, "          -runs <r> : number of independent runs"
          " (d: %d)\n", num_chains);
  fprintf(stderr, "          -histo <equi>: show histogram after <equi> steps\n");
  fprintf(stderr, "          -S3+ : measure number of 111+ blocks"
          " (d: 1s)\n");
  fprintf(stderr, "          -S_l : measure size of largest 1+ blocks"
          " (d: 1s)\n");
  fprintf(stderr, "          -alg <a> : choose algorithm (d: %d)\n", do_alg);
  fprintf(stderr, "                     0= direct changes\n");
  fprintf(stderr, "                     1= Metropolis\n");
  fprintf(stderr, "                     2= biased\n");
  exit(1);
}

n = atoi(argv[argz++]);                         /* read parameters */
sscanf(argv[argz++], "%lf", &alpha);
num_steps = atoi(argv[argz++]);

x = (double *) malloc(n*sizeof(double));              /* state vector */

data = (double *) malloc( (num_steps+1)*sizeof(double));     /* output */
for(step=0; step<=num_steps; step++)        /* reset output time series */
  data[step] = 0;

num_bins = n+1;                                 /* initialise histogram */
histo = (double *) malloc( num_bins*sizeof(double));
for(t=0; t< num_bins; t++)
    histo[t] = 0;
num_entries = 0;
start_histo = -0.5; end_histo = n+0.5;
delta = (end_histo - start_histo)/num_bins;


num_accept = 0;
for(chain=0; chain<num_chains; chain++)    /* average over several runs */
{
  for(t=0; t<n; t++)   /* initialize state all 1 */
    x[t] = init_val;
```

```
    value = (double) calc_S(n, x);
    data[0] += value;

    for(step=0; step<num_steps; step++)          /* run one MC chain */
    {
      if(do_alg == 0)
      {
        bernoulli_mc_step0(n, x, alpha, num_changes);
        num_accept++;
      }
      else if(do_alg == 1)
        num_accept += bernoulli_metropolis(n, x, alpha);
      else if(do_alg == 2)
        num_accept +=
          bernoulli_mc_step_bias(n, x, alpha, num_changes, theta, calc_S);


      value = (double) calc_S(n, x);

      data[step+1] += value;

      if(do_histo && (step >= num_equi) )
      {
        bin = (int) floor((value-start_histo)/delta);
        if( (bin >= 0)&&(bin < num_bins))          /* inside range ? */
        {
          histo[bin]++;                            /* count event */
          num_entries++;
        }
      }
    }
  }

  printf("#");
  for(t=0; t<argc; t++)
    printf(" %s", argv[t]);
  printf("\n");
  if(do_histo)
  {
    printf("# num_entries: %d\n", num_entries);
    for(t=0; t<num_bins; t++)          /* print normalized histogram */
      printf("%f %e\n", start_histo + (t+0.5)*delta,
             histo[t]/(delta*num_entries));
  }
  else
  {
    for(step=0; step<=num_steps; step++)   /* print data time series */
      printf("%d %f\n", step, data[step]/num_chains);
    printf("# ");
    for(t=0; t<n; t++)
      printf("%d", (int) x[t]);
    printf(" S: %f\n", calc_S(n, x));
  }
  printf("# fraction accepted: %f\n", num_accept/num_steps/num_chains);

  free(histo);
  free(data);

  return(0);
}
```