Part II.
NUMERICAL OPTIMIZATION

Chapter 5
Binary or Float

- Dynamic control problem:

$$\min\left(x_N^2 + \sum_{k=0}^{N-1} (x_k^2 + u_k^2) \right)$$

subject to

$$x_{k+1} = x_k + u_k\,, \quad k = 0,1,\ldots., N-1, \quad N = 45, \quad x_0 = 0$$

Binary

- Domain size
- Continuous domain
- Domain independence
- Constraint problems

Gray Coding (1)

```
procedure **Binary-to-Gray**
begin
    $g_1 = b_1$
        for $k = 2$ to $m$ do
            $g_k = b_{k-1}$ XOR $b_k$
end
procedure **Gray-to-Binary**
begin
    value = $g_1$
    $b_1$ = value
    for $k = 2$ to $m$ do
    begin
        if $g_k = 1$ then value = not value
        $b_k$ = value
    end
end
```

Binary	Gray
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Gray Coding (2)

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \qquad A^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$g = Ab$$

$$b = A^{-1}g$$

(*for m = 4*)

Float

- More problem specific/Easier design
- Higher precision / accuracy
- Larger continuous domains
- Faster computing
- Easier for designing operators
- Easier to operate constraints

- Representation
  - Real-value vector

- Crossover (p_c)

$$v_1 = (x_1, x_2, x_3, x_4) \qquad v_1' = (y_1, x_2, x_3, x_4)$$
$$v_2 = (y_1, y_2, y_3, y_4) \qquad v_2' = (x_1, y_2, y_3, y_4)$$

- Mutation (p_m)

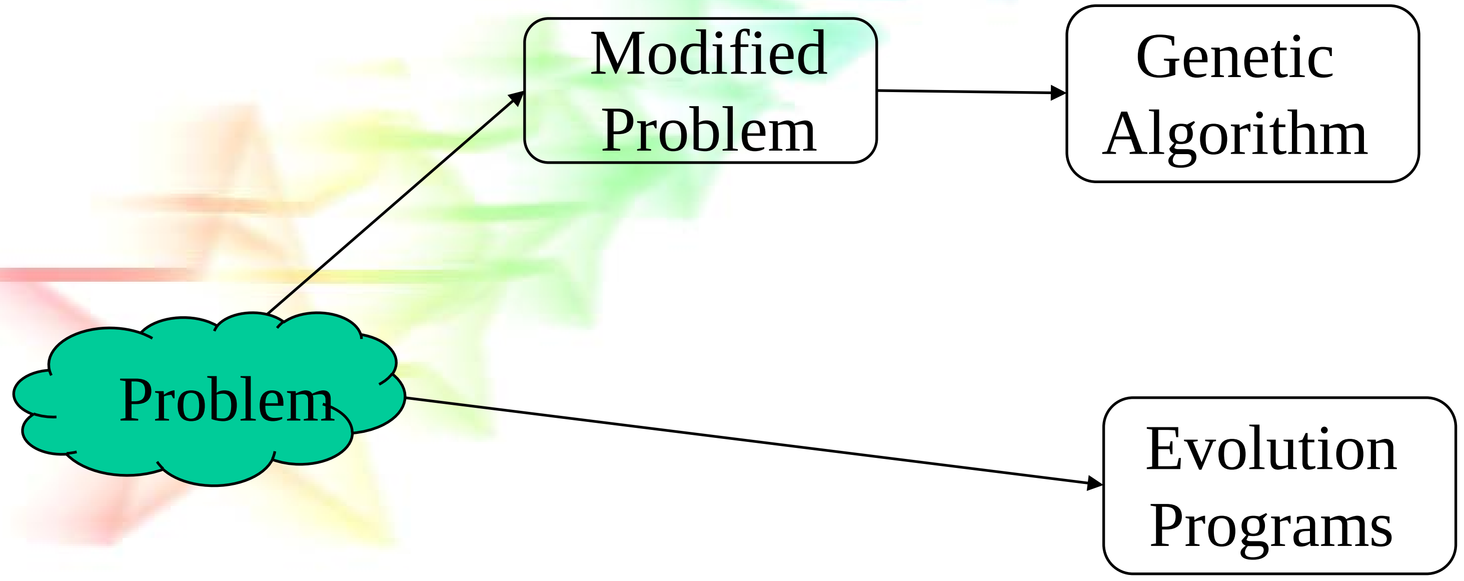
$$v_1 = (x_1, x_2, x_3, x_4) \longrightarrow v_1' = (x_1, x_2, x_3, [\mathrm{LB}x_4..\mathrm{UB}x_4])$$

Genetic Algorithm & Evolution Program

Evolution Program Approach

Genetic Algorithms + Data Structures

= Evolution Programs



Example

The dynamic control problem:

$$\min\left(x_N^2 + \sum_{k=0}^{N-1} (x_k^2 + u_k^2) \right)$$

Subject to

$$x_{k+1} = x_k + u_k, \quad k = 0,1,\ldots,N-1$$

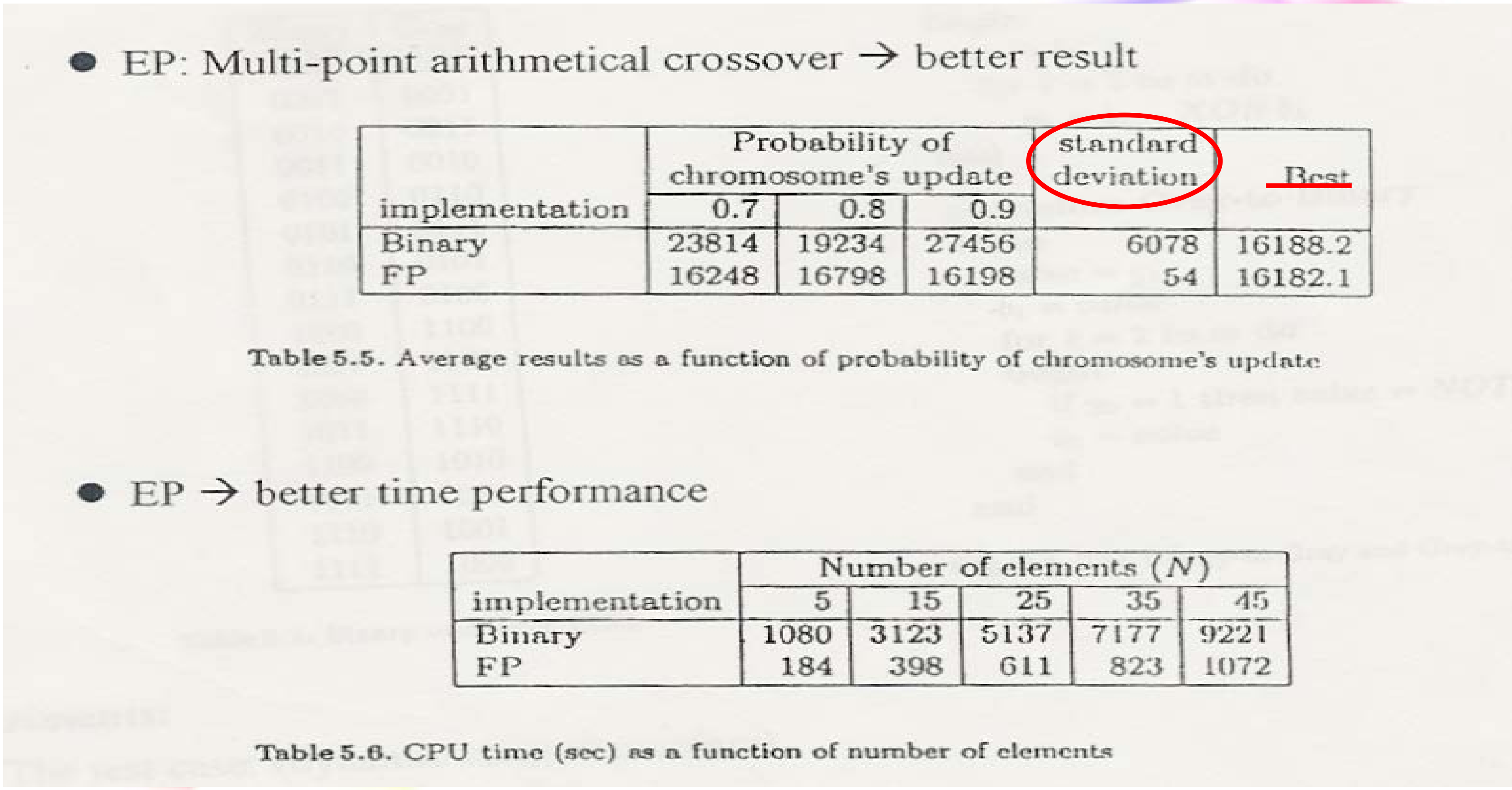
The optimal value:

$$J^* = K_0 x_0^2$$

where K_k is the solution of the Riccati equation:

$$K_k = 1 + K_{k+1}/(1 + K_{k+1}) \quad \text{and} \quad K_N = 1$$

- EP: Multi-point arithmetical crossover → better result



implementation	Probability of chromosome's update			standard deviation	Best
	0.7	0.8	0.9		
Binary	23814	19234	27456	6078	16188.2
FP	16248	16798	16198	54	16182.1

Table 5.5. Average results as a function of probability of chromosome's update

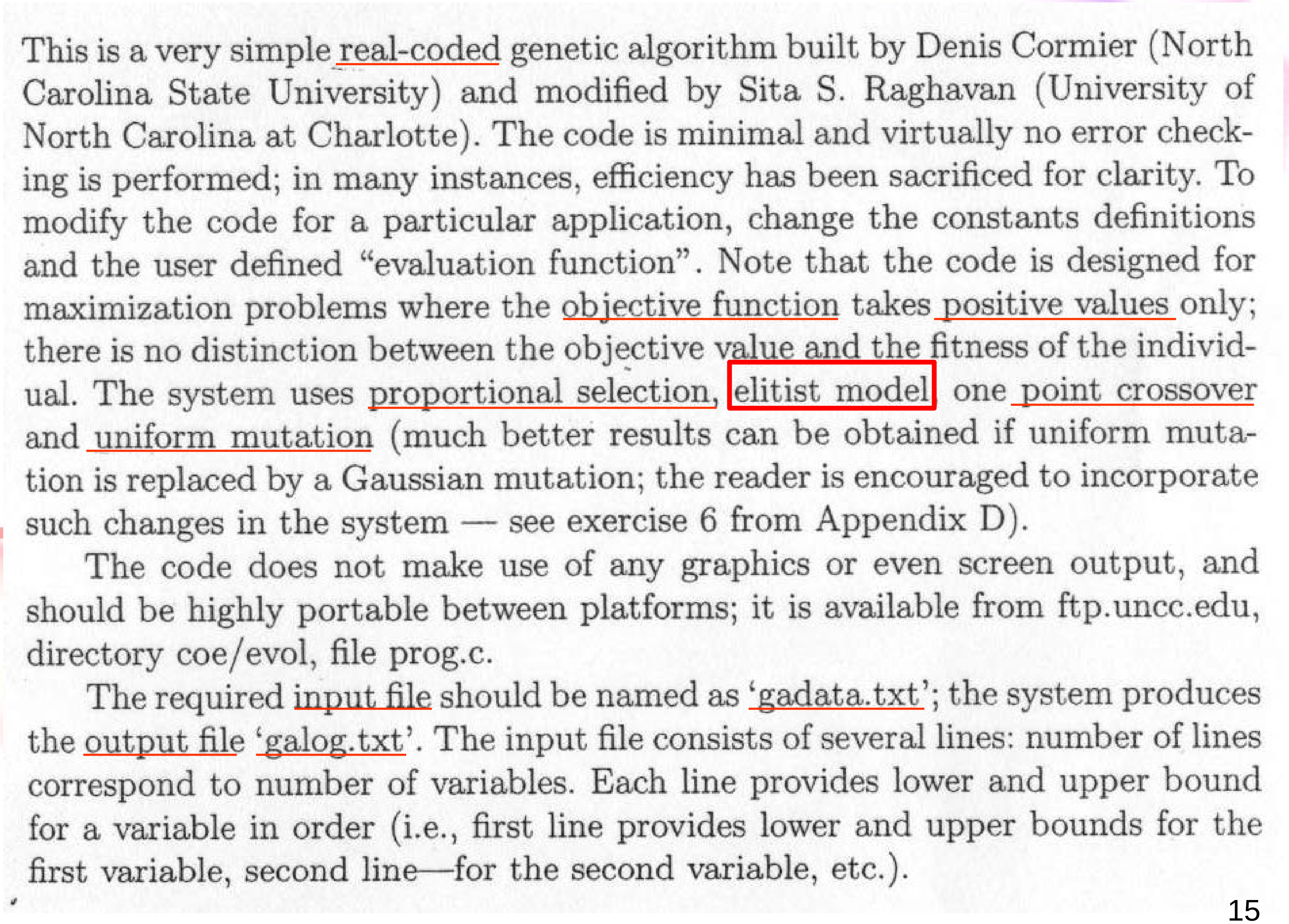
- EP → better time performance

implementation	Number of elements (N)				
	5	15	25	35	45
Binary	1080	3123	5137	7177	9221
FP	184	398	611	823	1072

Table 5.6. CPU time (sec) as a function of number of elements

Appendix A

$$\max\ f(x, y, z) = x^2 - xy + z$$



This is a very simple real-coded genetic algorithm built by Denis Cormier (North Carolina State University) and modified by Sita S. Raghavan (University of North Carolina at Charlotte). The code is minimal and virtually no error checking is performed; in many instances, efficiency has been sacrificed for clarity. To modify the code for a particular application, change the constants definitions and the user defined "evaluation function". Note that the code is designed for maximization problems where the objective function takes positive values only; there is no distinction between the objective value and the fitness of the individual. The system uses proportional selection, $\boxed{\text{elitist model}}$, one point crossover and uniform mutation (much better results can be obtained if uniform mutation is replaced by a Gaussian mutation; the reader is encouraged to incorporate such changes in the system — see exercise 6 from Appendix D).

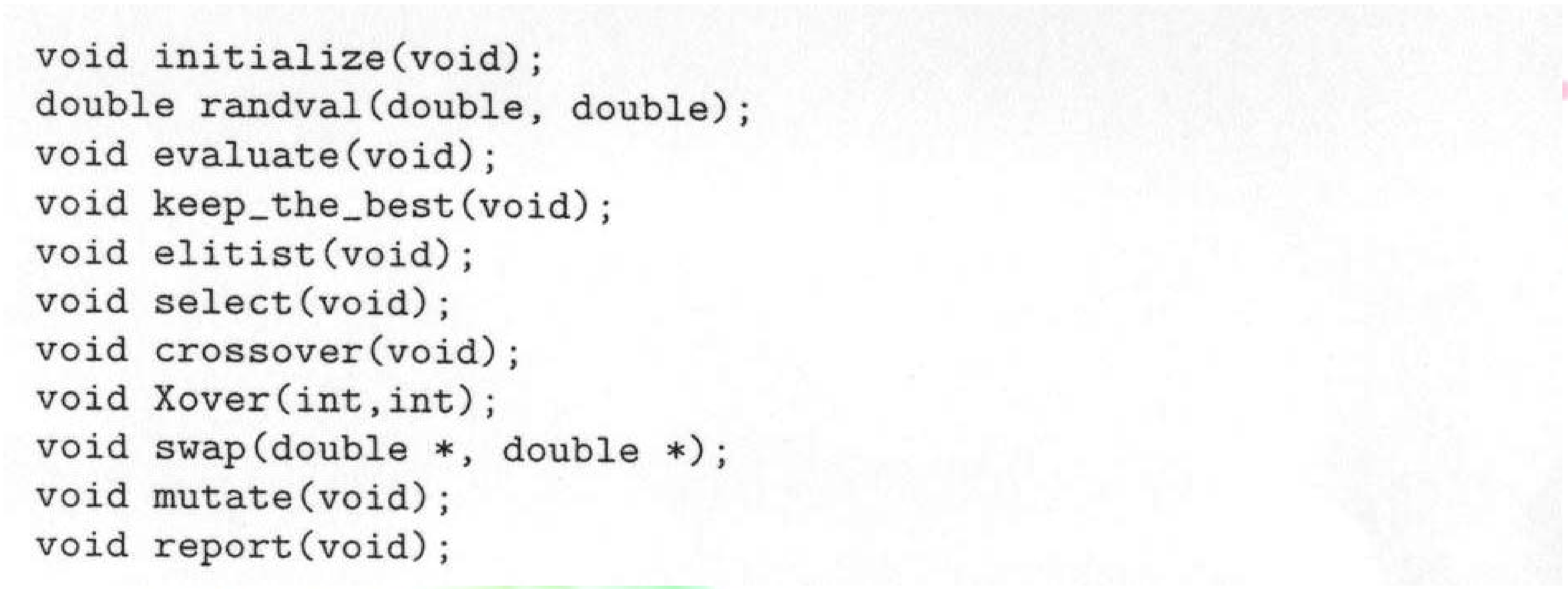
The code does not make use of any graphics or even screen output, and should be highly portable between platforms; it is available from ftp.uncc.edu, directory coe/evol, file prog.c.

The required input file should be named as 'gadata.txt'; the system produces the output file 'galog.txt'. The input file consists of several lines: number of lines correspond to number of variables. Each line provides lower and upper bound for a variable in order (i.e., first line provides lower and upper bounds for the first variable, second line—for the second variable, etc.).

General Scheme of an GA

1. A genetic **representation**
   - *Real-value vector*
2. An I**nitial population**
   - *Randomly*
3. Genetic **operators**
   - *Crossover: one-point crossover*
   - *Mutation: uniform mutation*
   - *Selection: Proportional selection (Roulette Wheel)* 、*Elitist model*
4. An **evaluation function**
   - *Fitness Function : (Problem Requirement)*
5. Values for various **parameters**
   - *population _size = N=100*
   - *probability _crossover =* P_c*=0.8*
   - *probability _mutation =* P_m*=0.15*
6. A way to **terminate** the algorithm
   - *t = MaxT=1000*
7. GA **algorithm**

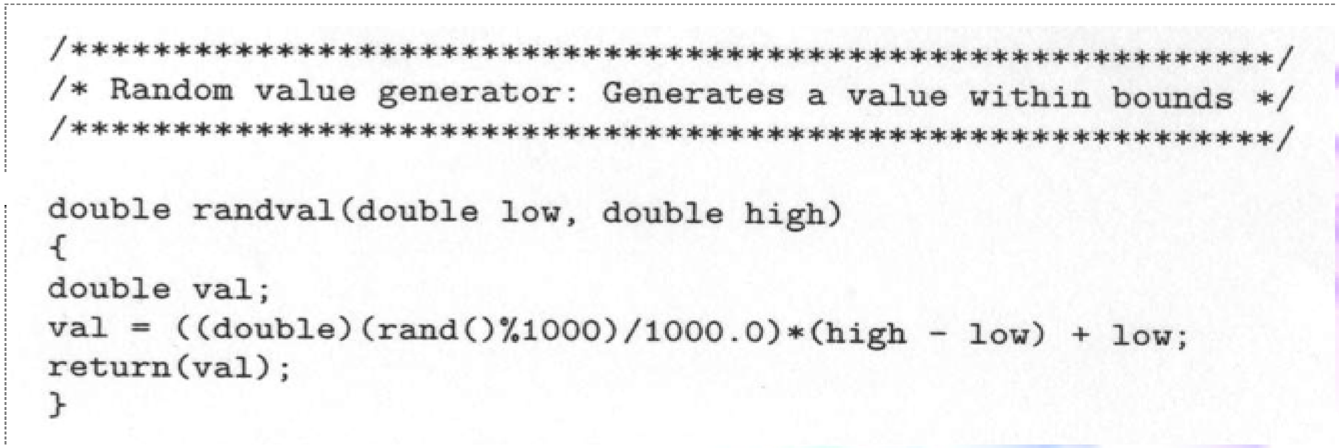
```
/**************************************************************/
/* This is a simple genetic algorithm implementation where the */
/* evaluation function takes positive values only and the      */
/* fitness of an individual is the same as the value of the    */
/* objective function      .                                   */
/**************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/* Change any of these parameters to match your needs */

#define POPSIZE 50                  /* population size */
#define MAXGENS 1000                /* max. number of generations */
#define NVARS 3                     /* no. of problem variables */
#define PXOVER 0.8                  /* probability of crossover */
#define PMUTATION 0.15              /* probability of mutation */
#define TRUE 1
#define FALSE 0

int generation;                    /* current generation no. */
int cur_best;                      /* best individual */
FILE *galog;                       /* an output file */

struct genotype /* genotype (GT), a member of the population */
{
  double gene[NVARS];            /* a string of variables */
  double fitness;                /* GT's fitness */
  double upper[NVARS];           /* GT's variables upper bound */
  double lower[NVARS];           /* GT's variables lower bound */
  double rfitness;               /* relative fitness */
  double cfitness;               /* cumulative fitness */
};

struct genotype population[POPSIZE+1];    /* population */
struct genotype newpopulation[POPSIZE+1]; /* new population; */
                                          /* replaces the */
                                          /* old generation */

/* Declaration of procedures used by this genetic algorithm */
```



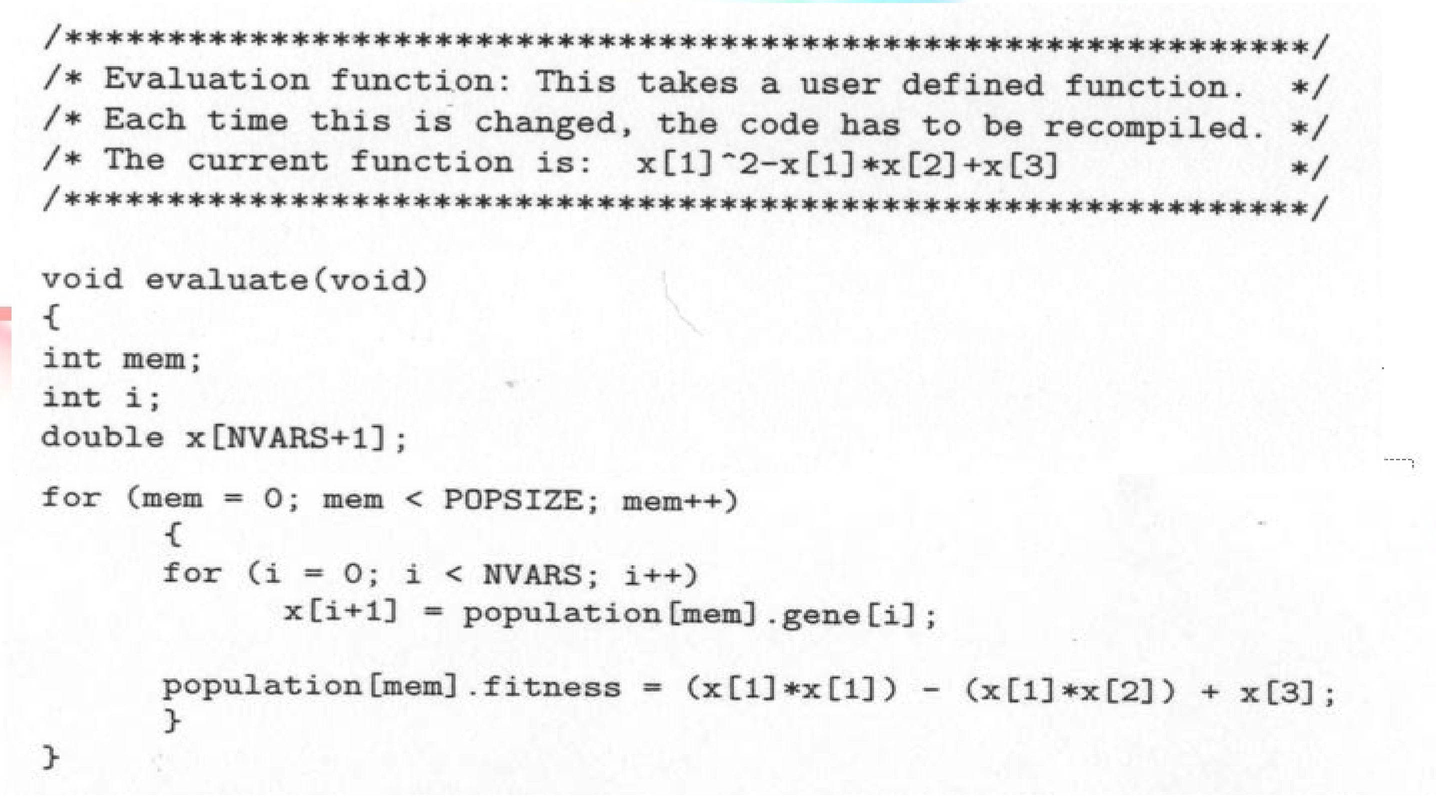
```
void initialize(void);
double randval(double, double);
void evaluate(void);
void keep_the_best(void);
void elitist(void);
void select(void);
void crossover(void);
void Xover(int,int);
void swap(double *, double *);
void mutate(void);
void report(void);
```

```
/***************************************************************/
/* Main function: Each generation involves selecting the best */
/* members, performing crossover & mutation and then          */
/* evaluating the resulting population, until the terminating */
/* condition is satisfied                                      */
/***************************************************************/

void main(void)
{
int i;

if ((galog = fopen("galog.txt","w"))==NULL)
        {
        exit(1);
        }
generation = 0;

fprintf(galog, "\n generation  best  average  standard \n");
fprintf(galog, " number      value fitness  deviation \n");

initialize();
evaluate();
keep_the_best();
while(generation<MAXGENS)
        {
        generation++;
        select();
        crossover();
        mutate();
        report();
        evaluate();
        elitist();
        }
fprintf(galog,"\n\n Simulation completed\n");
fprintf(galog,"\n Best member: \n");

for (i = 0; i < NVARS; i++)
    {
    fprintf (galog,"\n var(%d) = %3.3f",i,population[POPSIZE].gene[i]);
    }
fprintf(galog,"\n\n Best fitness = %3.3f",population[POPSIZE].fitness);
fclose(galog);
printf("Success\n");
}
/***************************************************************/
```

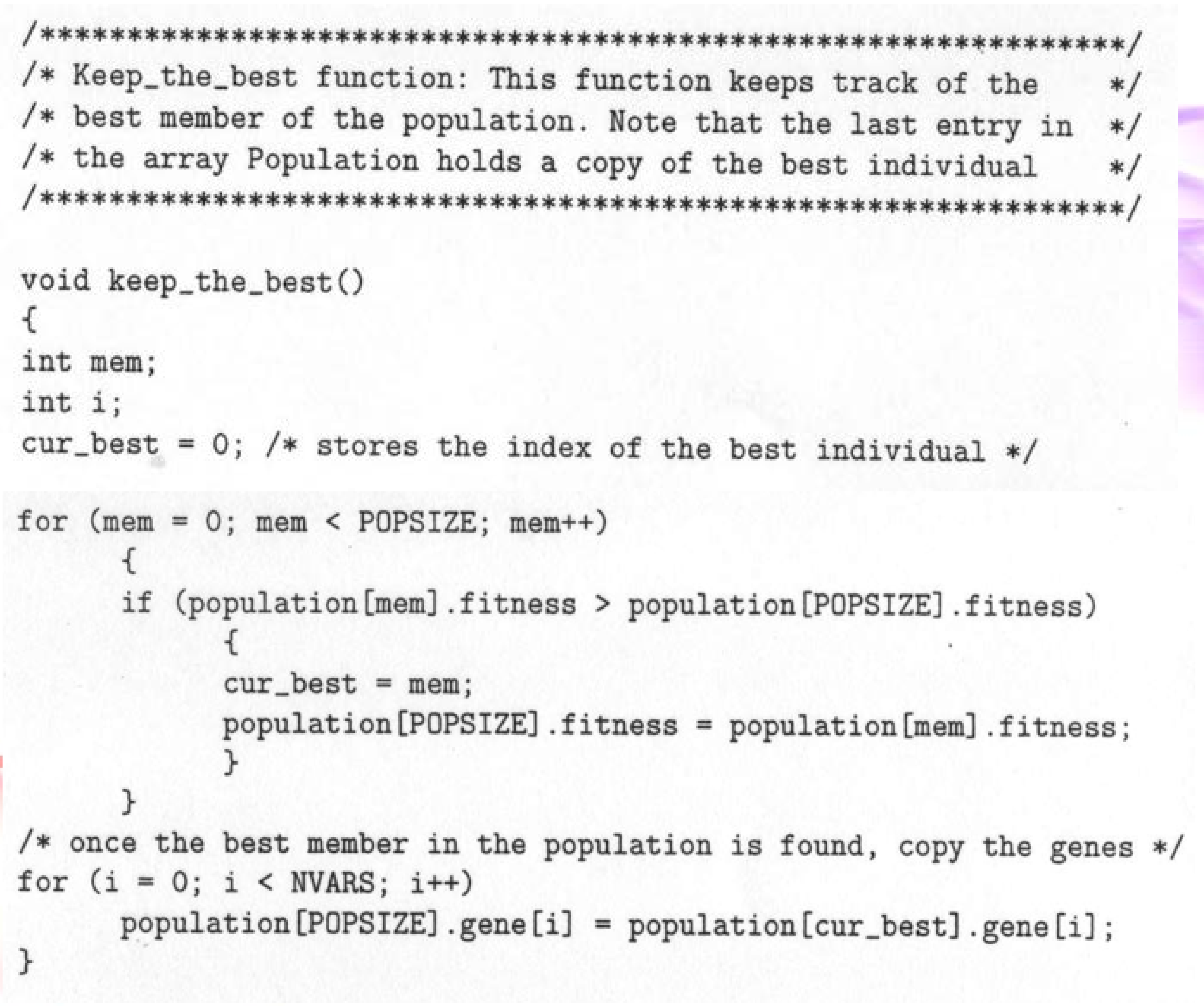
```
/***************************************************************/
/* Initialization function: Initializes the values of genes    */
/* within the variables bounds. It also initializes (to zero)  */
/* all fitness values for each member of the population. It    */
/* reads upper and lower bounds of each variable from the      */
/* input file 'gadata.txt'. It randomly generates values       */
/* between these bounds for each gene of each genotype in the  */
/* population. The format of the input file 'gadata.txt' is    */
/* var1_lower_bound var1_upper bound                           */
/* var2_lower_bound var2_upper bound ...                       */
/***************************************************************/

void initialize(void)
{
FILE *infile;
int i, j;
double lbound, ubound;

if ((infile = fopen("gadata.txt","r"))==NULL)
      {
      fprintf(galog,"\nCannot open input file!\n");
      exit(1);
      }

 /* initialize variables within the bounds */

 for (i = 0; i < NVARS; i++)
      {
      fscanf(infile, "%lf",&lbound);
      fscanf(infile, "%lf",&ubound);

      for (j = 0; j < POPSIZE; j++)
           {
           population[j].fitness = 0;
           population[j].rfitness = 0;
           population[j].cfitness = 0;
           population[j].lower[i] = lbound;
           population[j].upper[i]= ubound;
           population[j].gene[i] = randval(population[j].lower[i],
                                   population[j].upper[i]);
           }
      }

  fclose(infile);
  }
```



```
/*********************************************************************/
/* Random value generator: Generates a value within bounds */
/*********************************************************************/

double randval(double low, double high)
{
double val;
val = ((double)(rand()%1000)/1000.0)*(high - low) + low;
return(val);
}
```

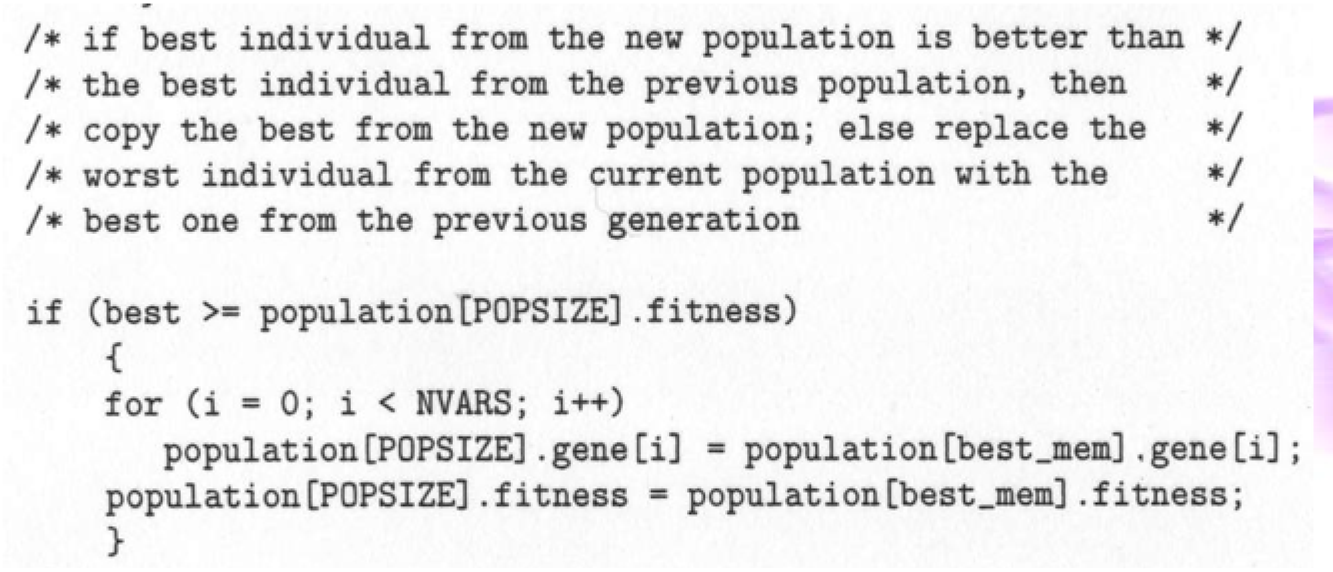


```
/*********************************************************************/
/* Evaluation function: This takes a user defined function.  */
/* Each time this is changed, the code has to be recompiled. */
/* The current function is:  x[1]^2-x[1]*x[2]+x[3]          */
/*********************************************************************/

void evaluate(void)
{
int mem;
int i;
double x[NVARS+1];

for (mem = 0; mem < POPSIZE; mem++)
      {
      for (i = 0; i < NVARS; i++)
            x[i+1] = population[mem].gene[i];

      population[mem].fitness = (x[1]*x[1]) - (x[1]*x[2]) + x[3];
      }
}
```

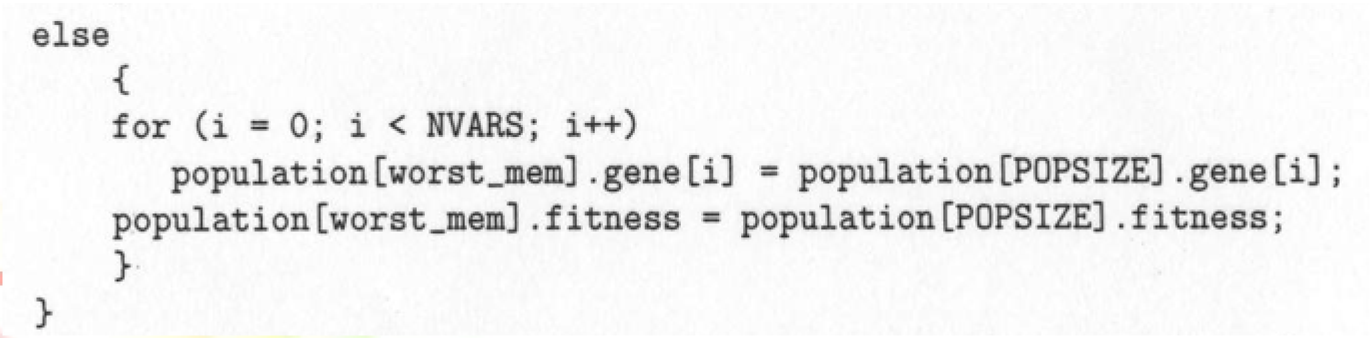


```
/**************************************************************/
/* Keep_the_best function: This function keeps track of the   */
/* best member of the population. Note that the last entry in */
/* the array Population holds a copy of the best individual   */
/**************************************************************/

void keep_the_best()
{
int mem;
int i;
cur_best = 0; /* stores the index of the best individual */

for (mem = 0; mem < POPSIZE; mem++)
      {
      if (population[mem].fitness > population[POPSIZE].fitness)
            {
            cur_best = mem;
            population[POPSIZE].fitness = population[mem].fitness;
            }
      }
/* once the best member in the population is found, copy the genes */
for (i = 0; i < NVARS; i++)
      population[POPSIZE].gene[i] = population[cur_best].gene[i];
}
```

```
/*********************************************************************/
/* Elitist function: The best member of the previous generation */
/* is stored as the last in the array. If the best member of    */
/* the current generation is worse then the best member of the  */
/* previous generation, the latter one would replace the worst  */
/* member of the current population                             */
/*********************************************************************/

void elitist()
{
int i;
double best, worst;              /* best and worst fitness values */
int best_mem, worst_mem; /* indexes of the best and worst member */

best = population[0].fitness;
worst = population[0].fitness;
for (i = 0; i < POPSIZE - 1; ++i)
      {
      if(population[i].fitness > population[i+1].fitness)
            {
            if (population[i].fitness> = best)
                  {
                  best = population[i].fitness;
                  best_mem = i;
                  }
            if (population[i+1].fitness <= worst)
                  {
                  worst = population[i+1].fitness;
                  worst_mem = i + 1;
                  }
            }
      else
            {
            if (population[i].fitness < = worst)
                  {
                  worst = population[i].fitness;
                  worst_mem = i;
                  }
            if (population[i+1].fitness >= best)
                  {
                  best = population[i+1].fitness;
                  best_mem = i + 1;
                  }
            }
      }
```

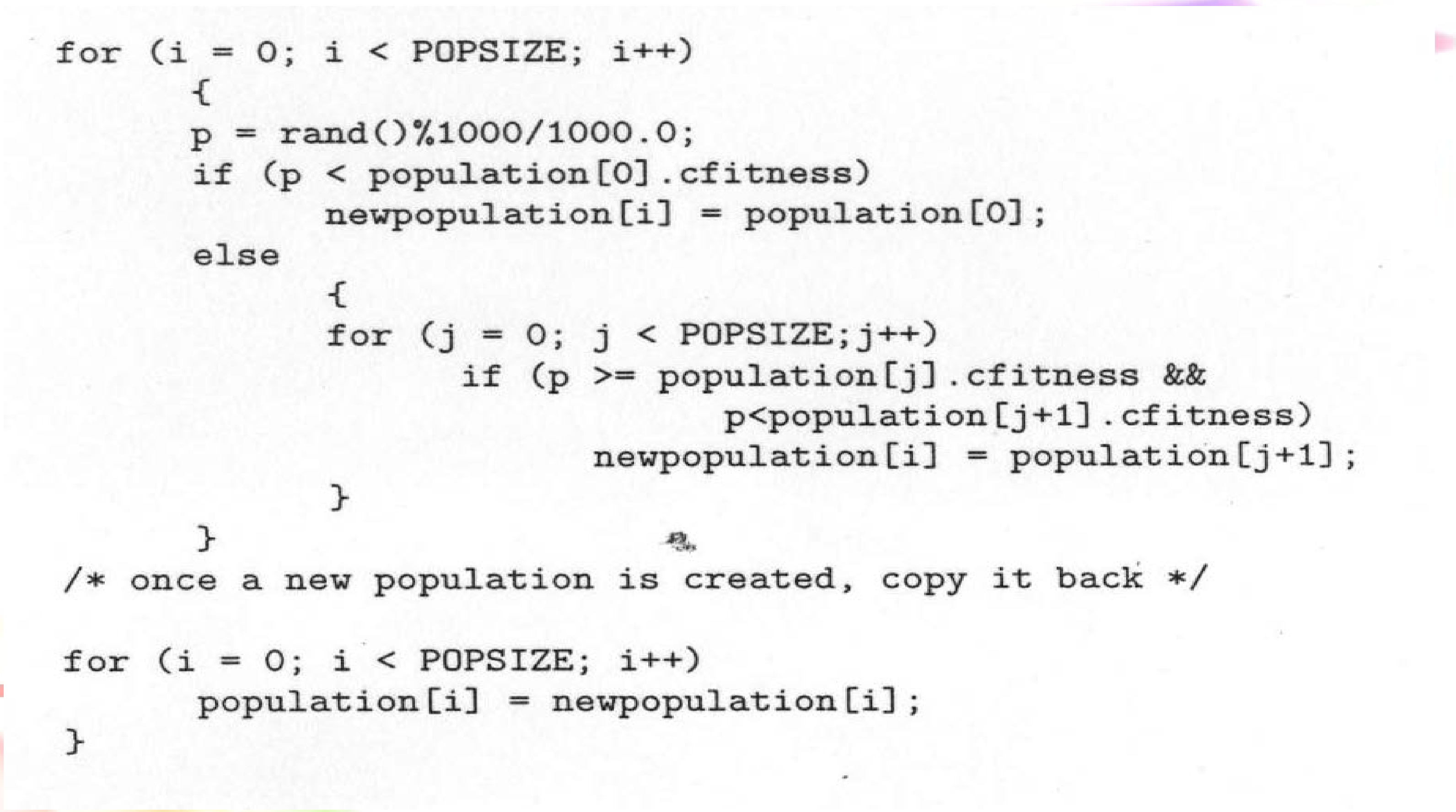


```
/* if best individual from the new population is better than */
/* the best individual from the previous population, then    */
/* copy the best from the new population; else replace the   */
/* worst individual from the current population with the     */
/* best one from the previous generation                     */

if (best >= population[POPSIZE].fitness)
    {
    for (i = 0; i < NVARS; i++)
       population[POPSIZE].gene[i] = population[best_mem].gene[i];
    population[POPSIZE].fitness = population[best_mem].fitness;
    }
```



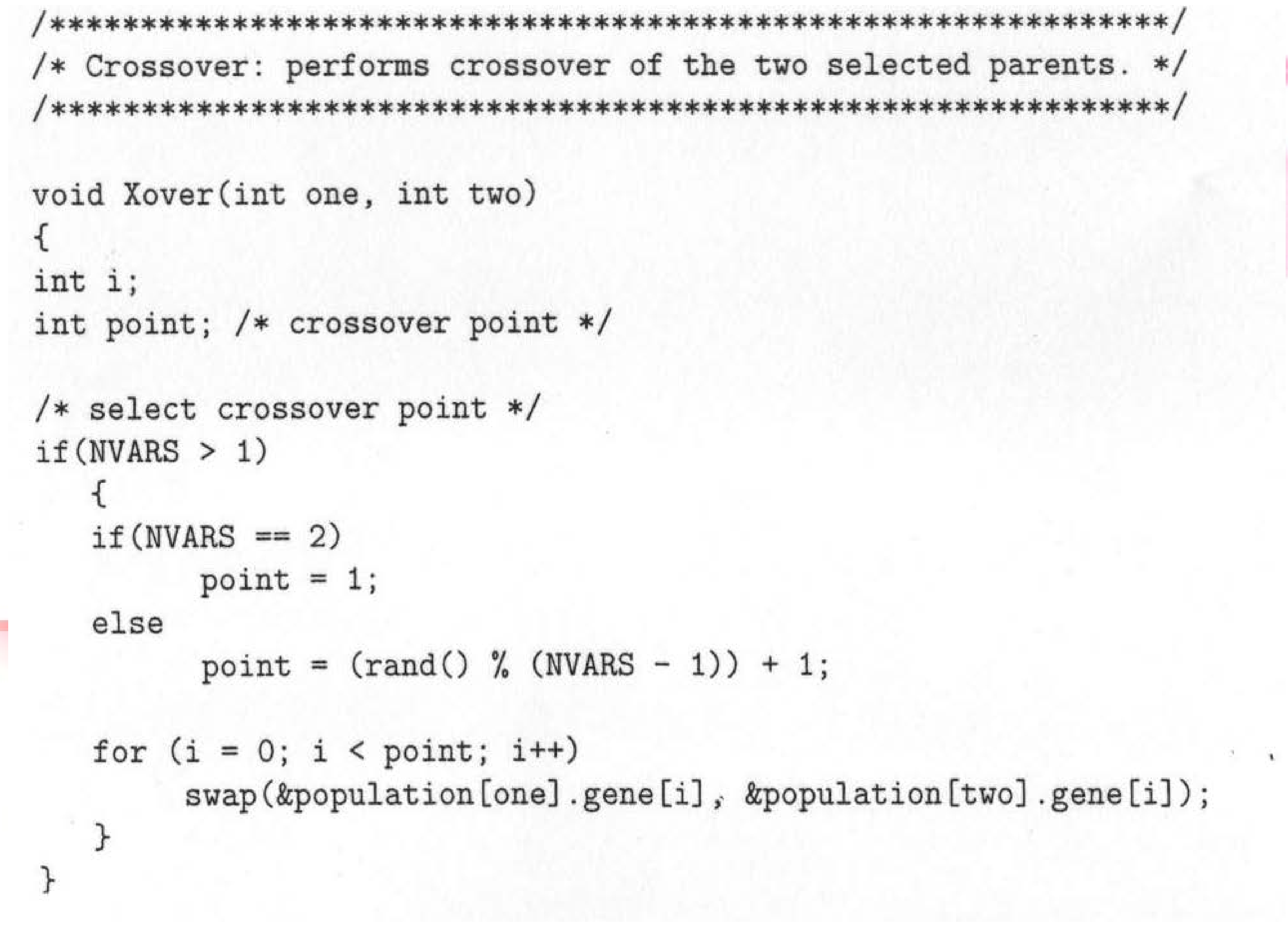
```
else
    {
    for (i = 0; i < NVARS; i++)
       population[worst_mem].gene[i] = population[POPSIZE].gene[i];
    population[worst_mem].fitness = population[POPSIZE].fitness;
    }
}
```

```
/*************************************************************/
/* Selection function: Standard proportional selection for    */
/* maximization problems incorporating elitist model - makes  */
/* sure that the best member survives                         */
/*************************************************************/

void select(void)
{
int mem, i, j, k;
double sum = 0;
double p;
/* find total fitness of the population */
for (mem = 0; mem < POPSIZE; mem++)
        {
        sum += population[mem].fitness;
        }

/* calculate relative fitness */
for (mem = 0; mem < POPSIZE; mem++)
        {
        population[mem].rfitness =  population[mem].fitness/sum;
        }
population[0].cfitness = population[0].rfitness;

/* calculate cumulative fitness */
for (mem = 1; mem < POPSIZE; mem++)
        {
        population[mem].cfitness =  population[mem-1].cfitness +
                                    population[mem].rfitness;
        }

/* finally select survivors using cumulative fitness. */
```

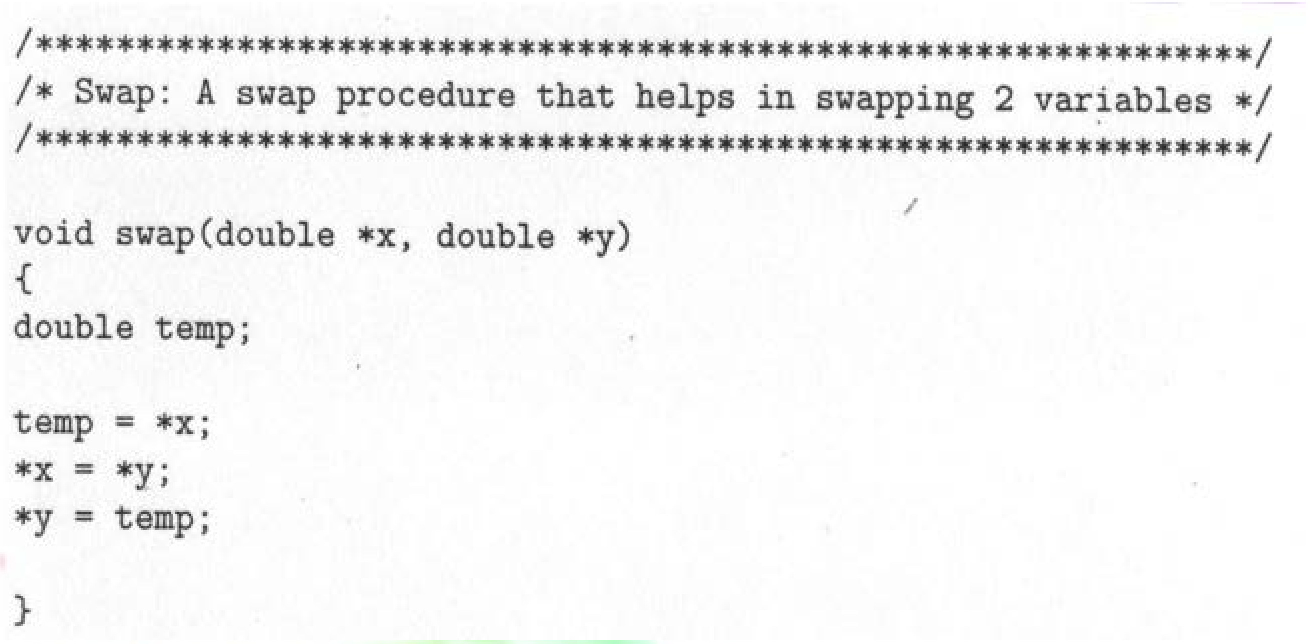


```
for (i = 0; i < POPSIZE; i++)
      {
      p = rand()%1000/1000.0;
      if (p < population[0].cfitness)
            newpopulation[i] = population[0];
      else
            {
            for (j = 0; j < POPSIZE;j++)
                  if (p >= population[j].cfitness &&
                          p<population[j+1].cfitness)
                        newpopulation[i] = population[j+1];
            }
      }
/* once a new population is created, copy it back */

for (i = 0; i < POPSIZE; i++)
      population[i] = newpopulation[i];
}
```

```
/***************************************************************/
/* Crossover selection: selects two parents that take part in  */
/* the crossover. Implements a single point crossover          */
/***************************************************************/

void crossover(void)
{
int i, mem, one;
int first  =  0; /* count of the number of members chosen */
double x;

for (mem = 0; mem < POPSIZE; ++mem)
      {
      x = rand()%1000/1000.0;
      if (x < PXOVER)
            {
            ++first;
            if (first % 2 == 0)
                  Xover(one, mem);
            else
                  one = mem;
            }
      }
}
```

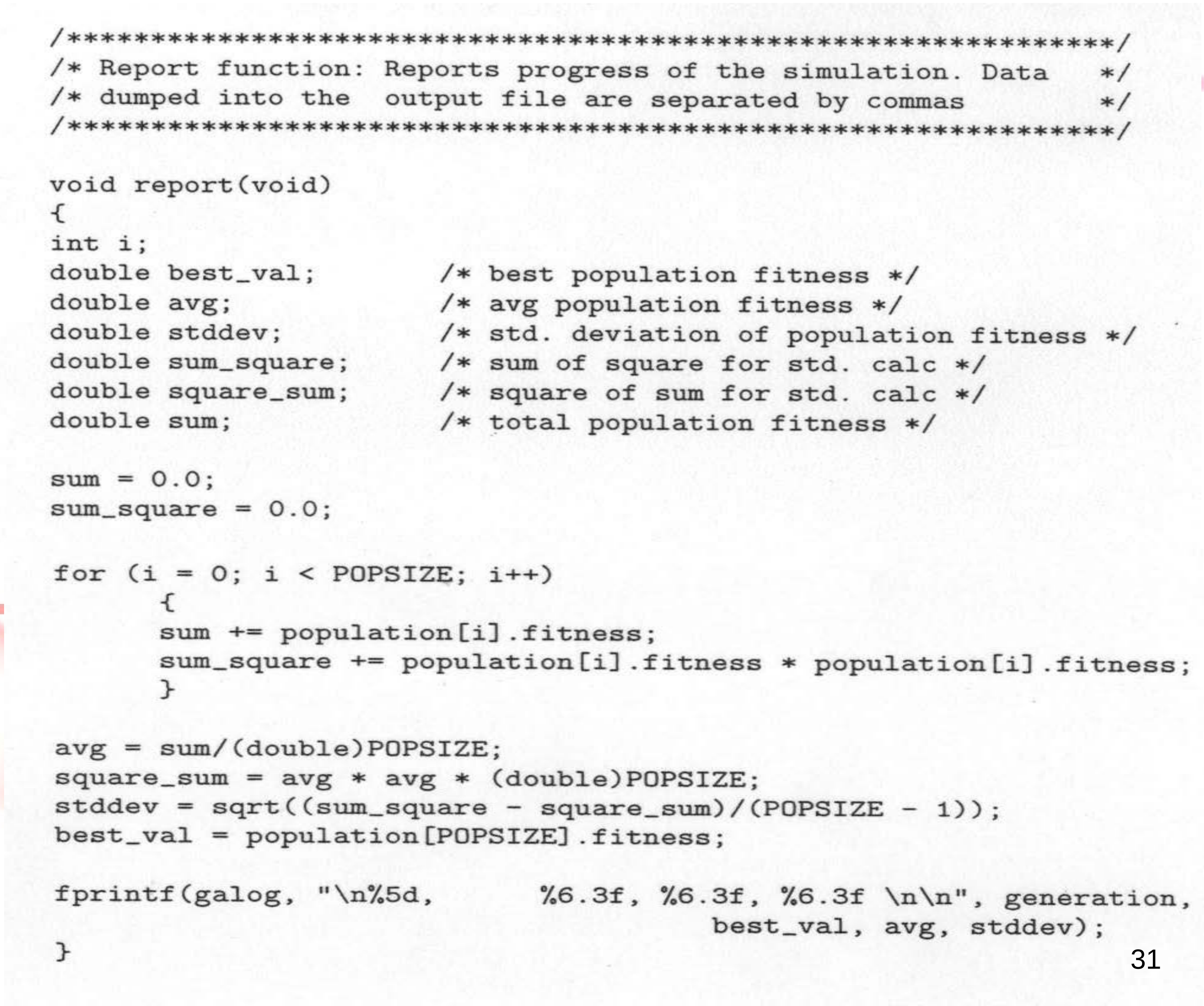


```
/*****************************************************************/
/* Crossover: performs crossover of the two selected parents. */
/*****************************************************************/

void Xover(int one, int two)
{
int i;
int point; /* crossover point */

/* select crossover point */
if(NVARS > 1)
   {
   if(NVARS == 2)
         point = 1;
   else
         point = (rand() % (NVARS - 1)) + 1;

   for (i = 0; i < point; i++)
        swap(&population[one].gene[i], &population[two].gene[i]);
   }
}
```



```
/********************************************************************/
/* Swap: A swap procedure that helps in swapping 2 variables */
/********************************************************************/

void swap(double *x, double *y)
{
double temp;

temp = *x;
*x = *y;
*y = temp;

}
```

```
/***************************************************************/
/* Mutation: Random uniform mutation. A variable selected for */
/* mutation is replaced by a random value between lower and   */
/* upper bounds of this variable                              */
/***************************************************************/

void mutate(void)
{
int i, j;
double lbound, hbound;
double x;

for (i = 0; i < POPSIZE; i++)
      for (j = 0; j < NVARS; j++)
            {
            x = rand()%1000/1000.0;
            if (x < PMUTATION)
                  {
                  /* find the bounds on the variable to be mutated */
                  lbound = population[i].lower[j];
                  hbound = population[i].upper[j];
                  population[i].gene[j] = randval(lbound, hbound);
                  }
            }
}
```



```
/*****************************************************************/
/* Report function: Reports progress of the simulation. Data    */
/* dumped into the  output file are separated by commas          */
/*****************************************************************/

void report(void)
{
int i;
double best_val;            /* best population fitness */
double avg;                 /* avg population fitness */
double stddev;              /* std. deviation of population fitness */
double sum_square;          /* sum of square for std. calc */
double square_sum;          /* square of sum for std. calc */
double sum;                 /* total population fitness */

sum = 0.0;
sum_square = 0.0;

for (i = 0; i < POPSIZE; i++)
      {
      sum += population[i].fitness;
      sum_square += population[i].fitness * population[i].fitness;
      }

avg = sum/(double)POPSIZE;
square_sum = avg * avg * (double)POPSIZE;
stddev = sqrt((sum_square - square_sum)/(POPSIZE - 1));
best_val = population[POPSIZE].fitness;

fprintf(galog, "\n%5d,      %6.3f, %6.3f, %6.3f \n\n", generation,
                                          best_val, avg, stddev);
}
```