Evolutionary Computation in Muscle Action Simulations

Colby Blair
Computer Science Undergraduate
University of Idaho Computer Science Department

December 14th, 2011

Abstract

In biomechanics, there are many interactions between the body and brain. Some interactions can be quantified as electrical signals to motor nerves of muscles. The result is a particular musclular reaction due to an electrical signal. The input is an electrical nerve signal, and the output is a measured velocity at a joint. The overall result to this sytem of inputs and output is actions like walking.

In this project, a Genetic Algorithm (GA) is used to replace the Simulated Annealing (SA) optimization that Dr. Craig McGowan uses for his research in biomechanics. His research takes EMG signals as inputs, and tries to predict joint velocities as outputs. This report shows that the GA is much slower, but maintains better solutions when it is set up with the right parameters. This report will also recommend further research opportunities for better results and performance.

Contents

List of Figures

1 Introduction

Walking requires a lot of complex tasks to make locomotion possible [1]. The inputs to muscle motor nerves and the outputs of joint motion are one way of measuring all these complex tasks. The data collected as inputs and outputs fits nicely into Evolutionary Computation (EC). EC uses many measured inputs and expected outputs, and evolves a system that will produce those expected outputs on its own.

The EC program will do this with some margin of error, of course, but the error of the solution should be small. The EC solution will then be useful when there are a set of inputs that the output is not already know, like in a simulation. In this report, we will use a **Genetic Program (GA)** as it fits the multiple input to one output model well. The expected output is used to evaluate the GA individuals using a fitness function. This fitness function calls the muscle simulation function with paramaters help by each GA individual. These parameters are changed in a scheme discribed next, and the individuals' fitnesses depend on how well their parameters do in the simulation.

1.1 Hypothesis

As noted above, the GA is set up to represent SA as closely as possible, at least at first. Our hypothesis is that the GA will find better solutions than the SA, while having the advantage of trying more diverse solution that would could cause worse solutions for an SA. The GA will be able to better search the domain, without as many problem with local optimums. These advantages may come at the cost of process time, but are hypothesized to not be significant.

2 Experiment Description

2.1 Test Problems

In Dr. Craig McGowan's research, a **Simulated Annealing (SA)** algorithm is used. SA's are superior for some statistical problem over conventional algorithms [2], like optimizing muscle excitation patterns. SA's are modeled after the method used to cool and harden material like steel. The start with a high 'temperature', which in optimization resembles the cost / error level of solution parameters. The temperature is decreased by picking random neigbor solutions that are better or worse within a certain tolerance. This tolerance is relative to the temperature, and as time passes and the temperature decreases, the tolerance becomes less.

The temperature is measured on how well muscle excitation pattern solutions simulate walking. Walking is judged on a fitness function, described in section 2.3.

2.2 Genetic Algorithm

To start, we wanted to start with a GA that was as similar to an SA baseline as possible. The GA could then be varied, trying different methods with increasing variation. This first approach was to use a GA with a **generational** method of regeneration, a **tournament selection** method for crossover, a **flip or two point crossover** method, and a modified **uniform mutation** for newly generated individuals.

A Genetic Algorithm (GA) creates a **population** of individuals, each of which have attributes to try to solve a problem. Their correctness will be measure with a **fitness function**. The best of these individuals are found with **selection**, their attributes are combined using **crossover**, and a child is produced from this. Each child is then slightly **mutated**, or its attributes changed. This results in new individuals in the population, and the whole process is started again.

Please note that in this report, **max fitness is the worst**, and **min fitness is the best**. This is non-intuitive, because usually $fitness = \frac{1}{cost}$. But this project used $fitness = cost$ for technical reasons.

The **population** for all the GA methods in this report is represented as a list of vectors. The **initial population** has random values set in the attributes vector (x) for every individual (see Figure 4):

$$P = i_1, i_2, ...i_j$$

where
i_n is a vector of floats or integers ($= x_1, x_2, ...x_y$)
$j = 10$
$y = 96$
$x_n = [B_L, B_U]$
B_L is the lower bound, $= -5.0$
B_U is the upper bound, $= 100.00$

Figure 1: The representation of the population

2.2.1 Selection

For **tournament selection** (Figure 2), the two individuals with the smallest fitnesses (best) will be selected from a random sample of the entire population. The sample size is represented by k. As k gets bigger, the best individuals of a population are more likely preserved, while k getting smaller means more variety in the population.

$$selection(P) = \quad i_1, i_2, ...i_k$$

where
P is the entire population
i_k is a random individual
k is the sample size, specified at run time

Figure 2: The selection function

Generally, k being larger leads to conversion on local optimums that the population may be surrounding, while k being lower allows individuals that may lead to some other optimum to remain.

2.2.2 Crossover

The **crossover** function will take two individuals i_a, i_b, and return a child individual, which has a mix of attributes (x_n values) from each parent.

One crossover method we may use is **two point crossover**:

$crossover(i_a, i_b) =$

$x_{i_a}[1:n], x_{i_b}[n+1:m], x_{i_a}[m+1:length(x_{i_a})]$ x values for the new child individual
where
$n =$ a random integer from $1...m-1$
$m = n+1$ the length of x_{i_a} or xi_b (same)

Figure 3: Two point crossover

Another crossover method may be **flip crossover**, where n attributes are swapped between two different individuals:

2.2.3 Uniform Mutation

Uniform mutation takes k values in an individual and sets them to new random values:

$$\begin{aligned}
mutate(x_i) \quad &= x_i'[a_1, a_2, ...a_n] \\
&\text{where} \\
&k \text{ attributes } a_1, a_2... \text{ are mutated by setting them to } f(a_x) \\
&f(a_x) = a_x + ((a_x + m) * scale) \\
&m = \text{random value between } B_L \text{ and } B_U \\
&B_L \text{ is the lower bound, } = -5.0 \\
&B_U \text{ is the upper bound, } = 100.00 \\
&k = 1 \\
&scale = \tfrac{1}{800}
\end{aligned}$$

Figure 4: Uniform mutation

2.2.4 Generational Reproduction

Generation reproduction of the population happens every cycle. It consists of selection, crossover, and mutation mentioned in the sections previously.The flow of converging on a solution is as follows:

- while user doesn't quit the process
  - calculate current fitnesses
  - select 2 individuals (parents) with minimum (best) fitnesses from a random subset of the entire population
  - create new child from the 2 parents using crossover
  - mutate the new child using uniform mutation
  - replace the the max fitness individual (worst), with the new child

2.3 Fitness Function

The fitness function is the cost function provided by Dr. McGowan's research:

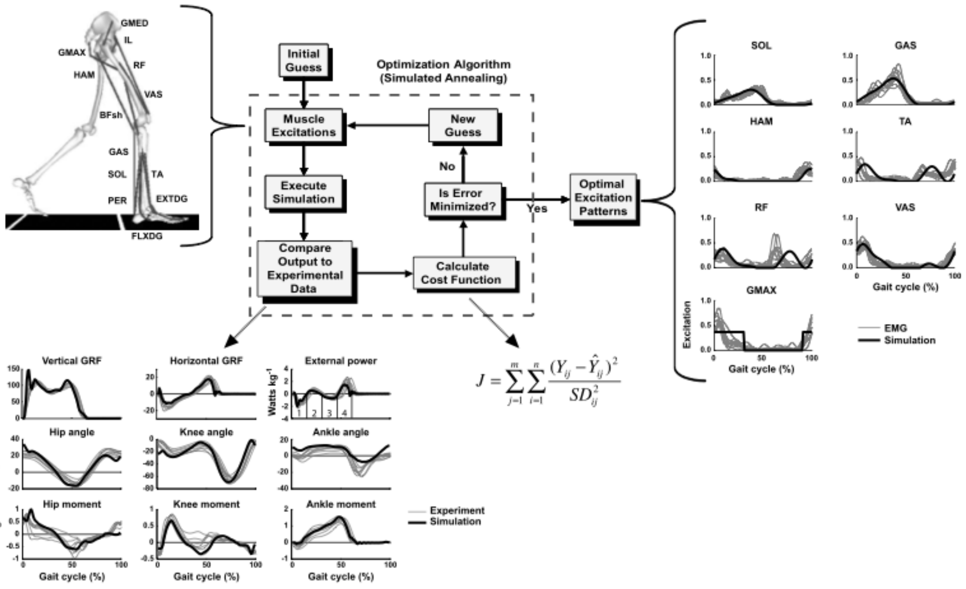

Figure 5: Fitness function overview [3]

2.4 Parameters

The parameter POP_SIZE is the population size. It is set to 10 for all experiments.

ATTR_SIZE is how many attributes each individual has. It is set to 96 for all experiments.

GA_LBOUND is the lower bound for random mutation values. It is set to -5.000 for all experiments.

GA_UBOUND is the upper bound for random mutation values. It is set to 100.00 for all experiments.

GA_DIFF_SCALE is the *scale* value in Figure 4. It is set to 800.000 for all experiments.

K_MUT is the denominator to equation $k = \frac{populationsize}{K\_MUT}$, where k is in Figure 4. It is set to 96 for all experiments.

K_SELECT is the denominator to the equation $k = \frac{populationsize}{K\_SELECT}$, k belonging to Figure 2. It is set to 1 for all experiments.

K_FLIP_XOVER is the number of points to flip in flip crossover.

3 Results

3.1 Simulated Annealing

This section lists some SA optimizations for comparison.

3.1.1 SA Run 1

Min:	127030005662992
Max:	1636549083561452
Standard Deviation:	217723255965676

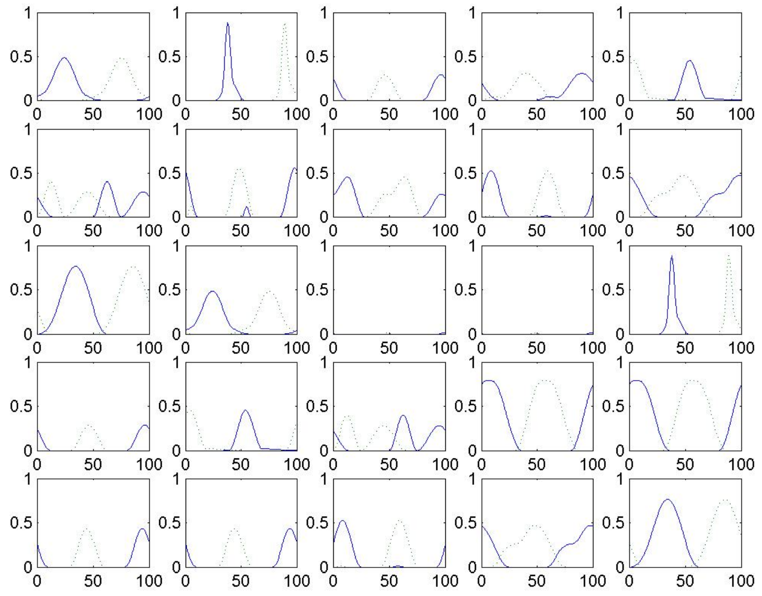

Figure 6: SA muscle excitation patterns

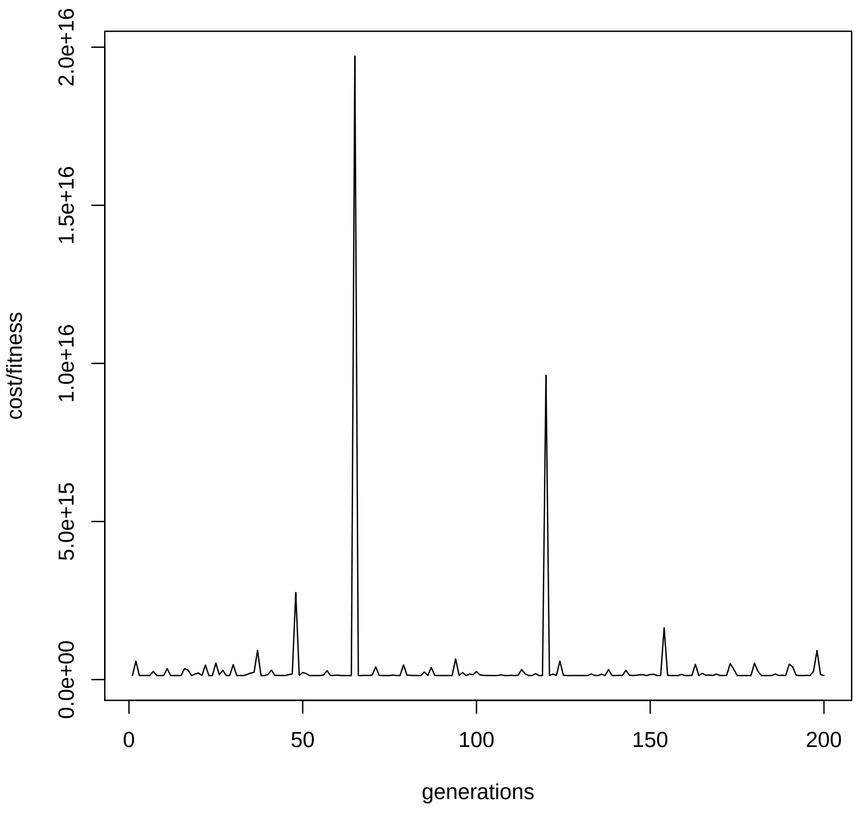

Figure 7: SA fitness over generations

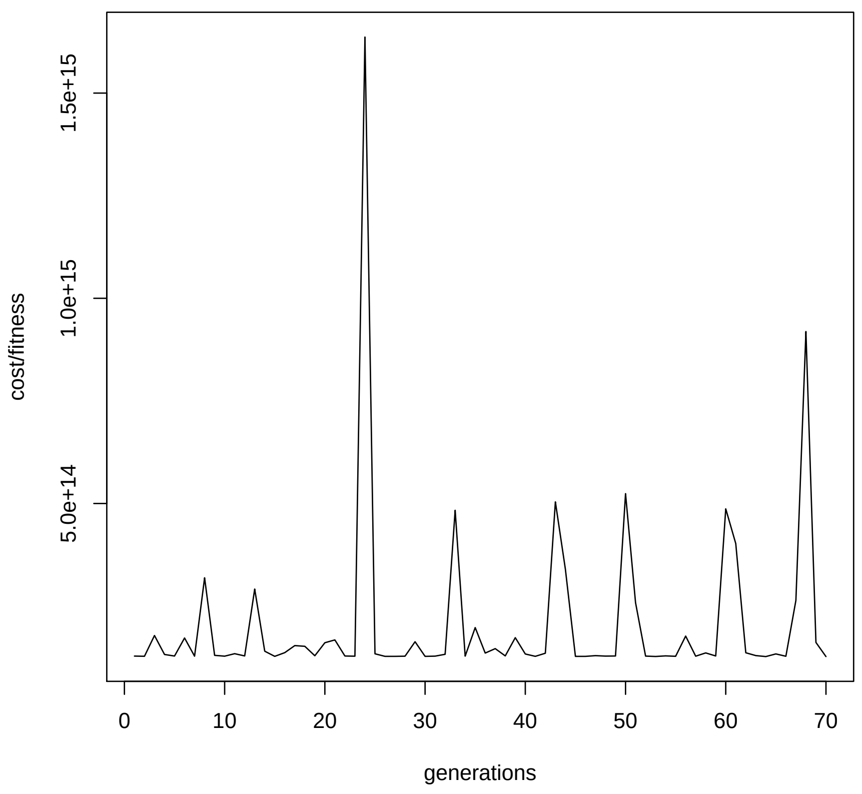

Figure 8: Same SA fitness over last 70 generations

3.2 Genetic Algorithm with Mutation Only

3.2.1 GA Run 1

Min:	127156943557514
Max:	127354767955212
Standard Deviation:	35435274465.6039

Parameters for ga:

POP_SIZE:	10
ATTR_SIZE:	96
GA_LBOUND:	-5.000
GA_UBOUND:	100.00
GA_DIFF_SCALE:	800.000
K_MUT:	96
K_SELECT:	1

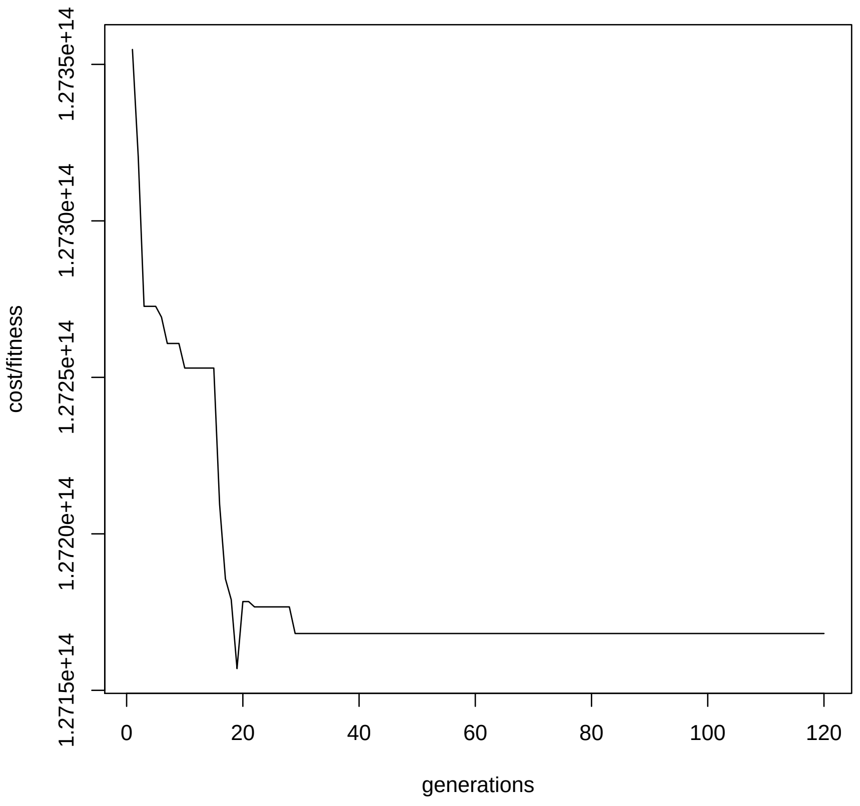

Figure 9: GA fitness, uniform mutation only

3.2.2 GA Run 2

Min:	126393050706189
Max:	126471924413236
Standard Deviation:	24842647883.601

Parameters for ga:

POP_SIZE:	10
ATTR_SIZE:	96
GA_LBOUND:	-5.000
GA_UBOUND:	100.00
GA_DIFF_SCALE:	800.000
K_MUT:	96
K_SELECT:	1

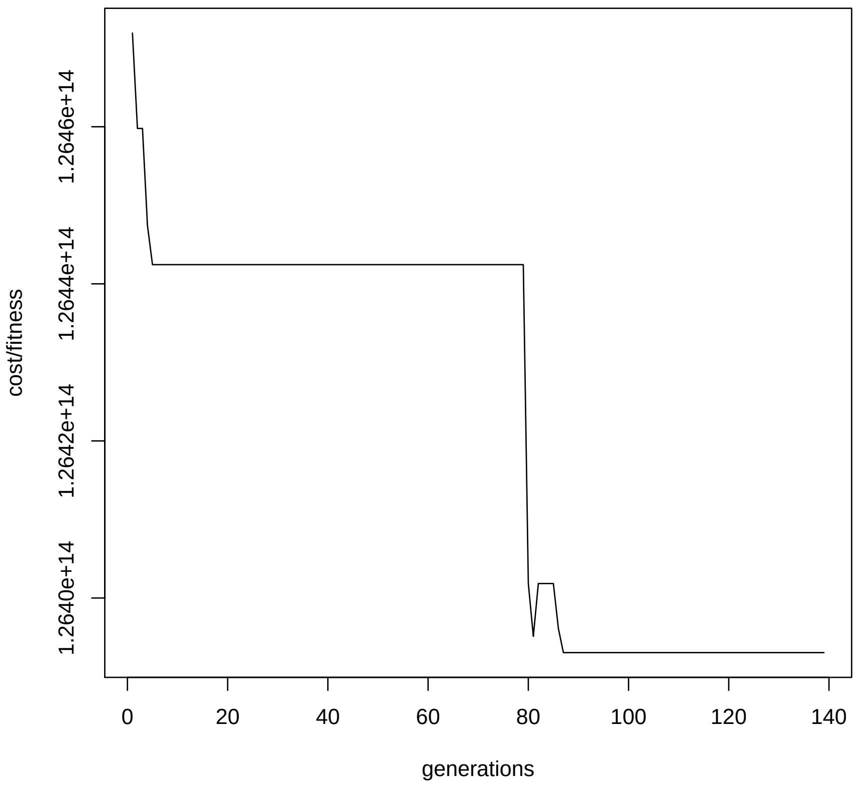

Figure 10: Another GA fitness, uniform mutation only

3.3 Genetic Algorithm with Uniform Mutation and Flip Crossover

3.3.1 GA Run 3

Min:	126286311965193
Max:	126903624988318
Standard Deviation:	226416086588.11

Parameters for ga:

POP_SIZE:	10
ATTR_SIZE:	96
GA_LBOUND:	-5.000
GA_UBOUND:	100.00
GA_DIFF_SCALE:	800.000
K_MUT:	96
K_SELECT:	1
K_FLIP_XOVER:	1

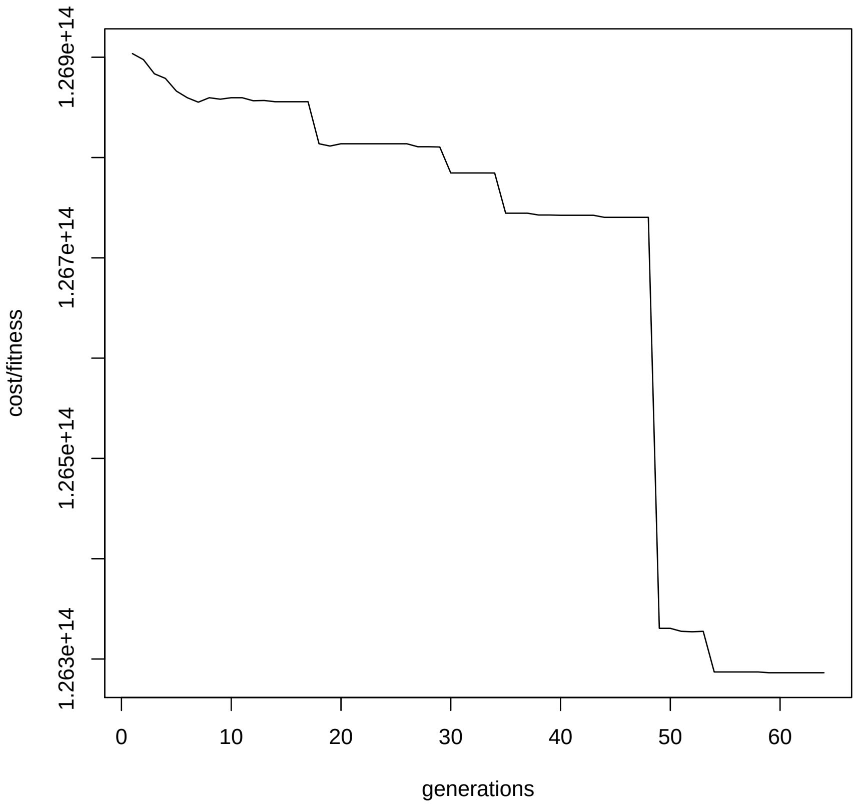

Figure 11: GA fitness, uniform mutation and one point flip crossover

3.3.2 GA Run 4

Min:	126295965034485
Max:	126900977814057
Standard Deviation:	250328896067.985

Parameters for ga:

POP_SIZE:	10
ATTR_SIZE:	96
GA_LBOUND:	-5.000
GA_UBOUND:	100.00
GA_DIFF_SCALE:	800.000
K_MUT:	96
K_SELECT:	1
K_FLIP_XOVER:	3

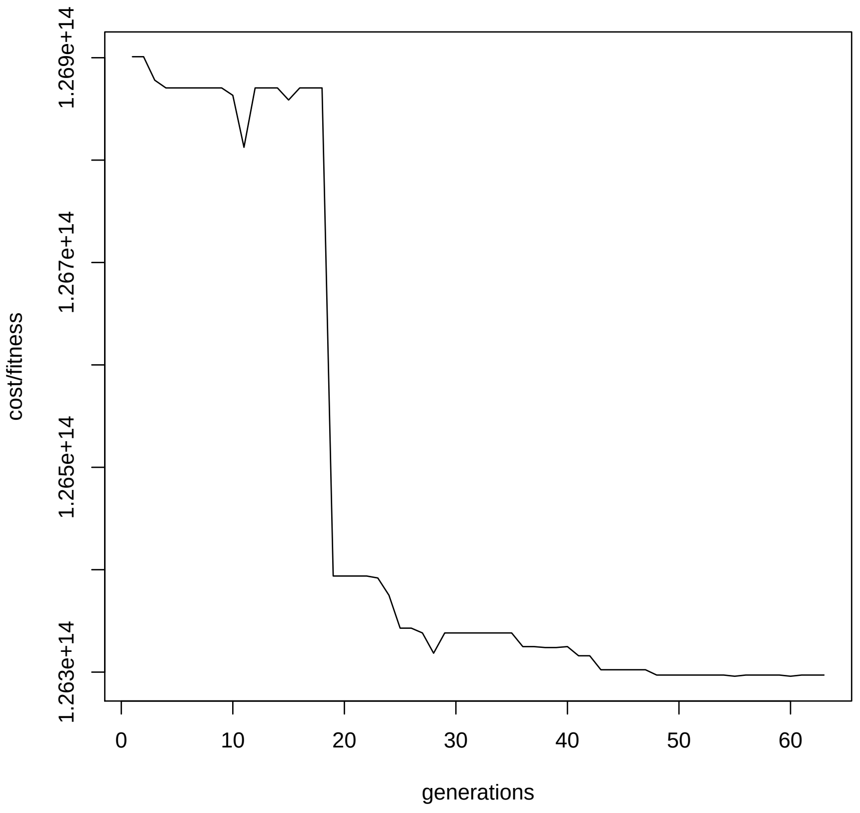

Figure 12: GA fitness, uniform mutation and three point flip crossover

3.4 Algorithm Comparison

Optimization	Min	Max	Standard Deviation
SA	127030005662992	1636549083561452	217723255965676
GA Run 1	127156943557514	127354767955212	35435274465.6039
GA Run 2	126393050706189	126471924413236	24842647883.601
GA Run 3	126286311965193	126903624988318	226416086588.11
GA Run 4	126295965034485	126900977814057	250328896067.985

4 Performance

The overhead of both the SA and the GA are almost nothing ($\leq$ .01 second per iteration / generation). Especially compared to the cost / fitness function, which has to run the simulation every time to get results. The main disadvantage of the GA over the SA is that for every iteration, it has to run the simulation function once for every individual in the population. For this project, the population size was 10, so the GA was approximately 10 times as slow as the SA.

The good thing about the slowdown, is that it happens in data independent region of the process; each fitness iteration for each individual can be calculated without needing to share data with other iterations. This meets the criteria for true **task parallelism**, or running each fitness function simultaneously.

4.1 Enhancements with Parallelism

To understand parallelism and the gains from it, we must consider Amdahl's Law:

$$A = \frac{1}{(1-P)+\frac{P}{N}}$$

where

P is the portion of the program that can be 'parallelized'
N processors or workers
$(1-P)$ is the sequential / serial portion (cannot be parallelized)

Figure 13: Amdah's Law. [4]

P informally is the portion of the program that can be split up into sections, each of which can be worked on simultaniously. The name for this process is **paralellization**. Informally, Ahmdahl's Law shows that the higher P, or portion of work that can be split up, the more beneficial adding more CPU's (N) is. Too many CPU's, and the benefit decreases.

For parallelism over many CPU's on a network, an API like OpenMPI can be used. This creates an additional element, and that is the time cost of moving data over the network:

$$A = \frac{1}{((1-P)+(T_P*N))+\frac{P}{N}}$$

where

P is the portion of the program that can be 'parallelized'
N processors or workers
$(1-P)$ is the sequential / serial portion (cannot be parallelized)
$(T_P * N)$ is the overhead

Figure 14: Amdah's Law (Figure 13) with Overhead.

Adding more CPU's actual slows parallel processes down at a certain point. For many algorithms, P can only be a fairly small size. The best value for N will then only be 1. If P is not very big to begin with, than any N greater than 1 will cause a slowdown instead of speedup. Thus, even though some programs can be parallelized, if $N > 1$ causes a slowdown, they shouldn't.

For the muscle excitation GA, we luckily have a fairly high $P = 13.10$ seconds, and a low $T_P = .08$ seconds. P is the measured time for each simulation to run, and T_P is an approximation of how long the data takes to travel across an MPI network. The resulting graph for speed is below:

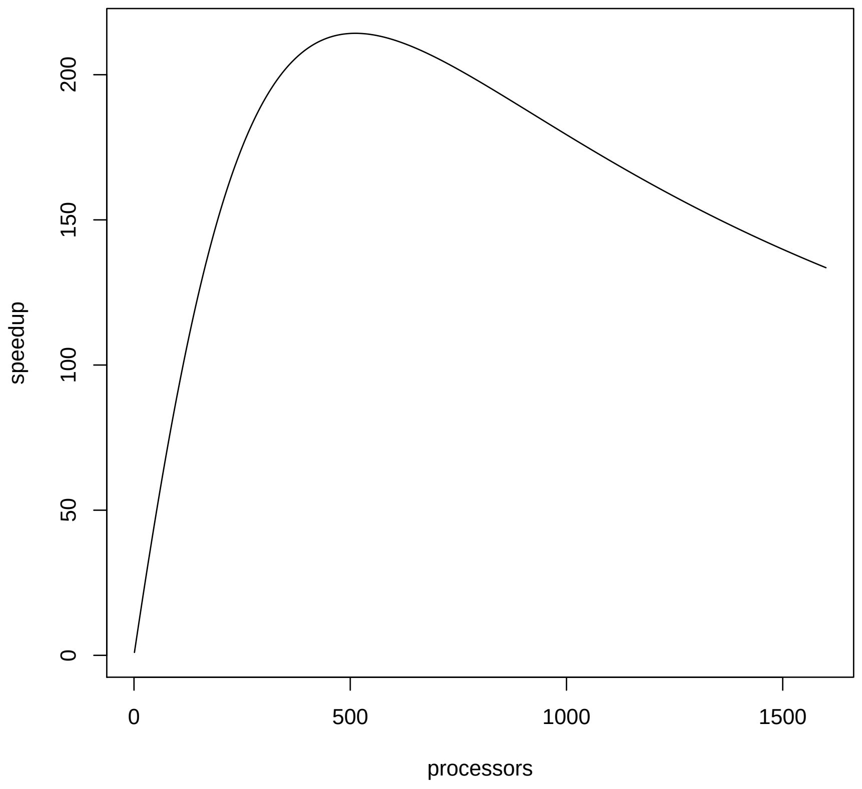

Figure 15: Proposed speedup using OpenMPI and Amdahl's Law with Overhead (Figure 14)

In figure 15, observe that the max speedup is 214.264919941776 with 512 CPU's. Although this is the most speedup, significantly less will give more speedup for the extra cost, and a small to medium size computing cluster (50-100+ CPU's) would be very beneficial.

5 Conclusion

In conclusion, the GA outperformed the SA a little bit at the cost of a lot of extra compute time. This may be still desirable, if the GA is in fact exploring a global optimum. It is probable, however, that there is a very sudden global optimum somewhere, and this GA doesn't maintain enough diversity to find it.

The project started with trying to imitate an SA using a GA, and then added more variability with mutation and crossover rates. What was quite suprising was how sensitive the existing solution were to mutation and crossover. The solutions the GA started with were recent bests found by the SA, and the GA originally changed them so much that fitness / cost went to infinity. This is why this project used such conservative changes, scaling down mutations 800 times (Figure 4) and even leaving out crossover (Section 3.1.1). When we did use crossover, it was very conservative flip crossover, like flipping only 1, 2, or 3 attributes between individuals.

This conservation could be because the GA is already stuck in a local optimum, and we have pressured it hard to not stray outside. It may also be that the fitness function is a little inaccurate.

In this report, it was hypothesized that the GA would find a better solution through diversity. Instead, this GA found a better solution within the search space that the SA was probably already searching. To get the GA to try more diverse solutions, it may be necessary to start the simulations without initial solutions already found by the SA. We also hypothesized not much performance difference between the SA and GA, but once we found out the simulation time, this was definitely not true.

This GA found better solutions in our limited testing, but could probably be better. Implementing the performance enhancements suggested may be the only way to try larger populations, and to try more parameters for the GA.

Finally, it should be said that these parameters for the GA could easily be controlled by another GA, or meta-GA. Even a GA for the fitness function may be a good idea, as Dr. McGowan's current research edits this by hand. There is a lot of opportunity for improvement, and hopefully some of the things found here will be useful in future research.

6 Source

These file have .cpp postfixes, but were originally written in C, and take a C style. The development environment was Visual Studio C++, and with a better understanding of it, these files will go back to being pure C.

6.1 ga_main.h

```
//#ifndef _GA_MAIN_H
//#define _GA_MAIN_H

#define USE_GA 1//comment out if you want to use SA instead

void ga_main(int , double *, double **, double **, double **,double **);
int ga_test();

//#endif
```

6.2 ga_main.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <limits>

using namespace std;

//taken from SA stuff to run simulation
#include "SA.h"
//#include "Look.h"
#include "SAstructs.h"

#include "structs.h"
#include "system_params.h"

extern "C" int submain(double **,double **,double **,double**);
extern "C" int submain_ga(double **,double **,double **,double**);

extern "C" OptStruct mdata; //new global data structure

//end SA stuff

//place holders for the muscle data structures, for use in fitness
// function
double **XR_GLOBAL;
double **XL_GLOBAL;
double **YR_GLOBAL;
double **YL_GLOBAL;
int N_MUS_GLOBAL;
int N_EMG_STEP_GLOBAL;
int N_PARAMS_GLOBAL;

//ga stuff
#include "ga_i.h"
#include "ga_pop.h"

double GA_UBOUND;
double GA_LBOUND;
double GA_DIFF_SCALE;

//mutation selection, % to mutate = k = nattr / K_MUT
extern int K_MUT;

//how many points flip crossover will do
extern int K_FLIP_XOVER;

//tournament selection, % to select for subpool for tournament selection
// % = k = (pop)size / K_SELECT
extern int K_SELECT;


double simple_fitness(struct ga_i *p)
{
        double sum = 0.0;
        int i;
        for(i = 0; i < p->nattr; i++)
        {
                sum += pow(p->attr[i], 2);
        }

        if(sum < 0.0)
        {
                fprintf(stderr,
                        "ERROR: negative fitness value. Exiting.\n");
                ga_i_print_attr(p);
                for(i = 0; i < p->nattr; i++)
                {
                        printf(" %f\n", p->attr[i]);
                }
                exit(1);
        }

        return(sum);
}


double muscle_fitness(struct ga_i *p)
{
        int i = 0;
        int j = 0;
        double *x = p->attr;

        //update / init the data
        update_group_excitation_patterns(XR_GLOBAL,YR_GLOBAL,XL_GLOBAL,
                                                                        YL_GLOBAL,x
        change_init(x,N_PARAMS_GLOBAL); //initial velocities

        double **XRa, **XLa ;//abscissa adjusted
        XRa=new double *[N_MUS_GLOBAL];
        XLa=new double *[N_MUS_GLOBAL];
        for(i=0;i<N_mus;i++){
                XRa[i]=new double[N_emg_step]; if(XRa[i]==NULL) exit(0);
                XLa[i]=new double[N_emg_step]; if(XLa[i]==NULL) exit(0);
        }

        for(j=0;j<N_MUS_GLOBAL;j++)
        {
                for(i=0;i<N_EMG_STEP_GLOBAL;i++)
                {
                        XRa[j][i]=XR_GLOBAL[j][i]*MAX_X/100.0;
                        XLa[j][i]=XL_GLOBAL[j][i]*MAX_X/100.0;
                }
        }//adjust abscissa

        //run the sim each iteration
        int BAD = submain_ga(XRa,YR_GLOBAL,XLa,YL_GLOBAL);
        //save_motion(); //write motion file even walking was not completed

        double retval = get_cost();

        //if it didn't walk, put highest cost possible
        if(BAD == 1)
        {
                        printf(" Didn't walk.\n");
                        retval = numeric_limits<float>::infinity( );
        }

        printf(" Cost/fitness: %f\n", retval);
        return(retval);
}


double simple_random(double current_val)
{
        int precision = 2; //the most decimal values allowed
        int scale = pow((double)10, precision);

        //double lbound = -5.12;
        //double ubound = 5.12;
        double lbound = GA_LBOUND; // = -5.0;
        double ubound = GA_UBOUND; //100.0;

        //get a random value in the range using ints
        int range = (ubound * scale) - (lbound * scale);
        int irand = rand() % range;

        //the new random value is a random value in the range, scaled
        // down to the original precision
        double drand = ubound - ( ((double)irand) / scale );

        //return(drand);

        //get the difference between the rand value and the current value
        double diff = current_val - drand; //can be neg or pos

        //scale down the difference
        double diff_scale = GA_DIFF_SCALE;
        diff = diff / diff_scale;

        //return the scaled down change of current value
        double retval = current_val - diff;

        //gaurantee max precision
        retval = retval * (10 * precision);
        retval = ceil(retval);
        retval = retval / (10 * precision);

        //if we go beyond the bounds, limit
        if(retval < lbound)
        {
                retval = lbound + (lbound / scale);
        }
        else if (retval > ubound)
        {
                retval = ubound - (ubound / scale);
        }

        return(retval);
}

int ga_test()
{
        int i;

        /* initialize random seed: */
        srand ( time(NULL) );

        double d1[] = { \
                        -5.0000, -4.0000, -3.0000, -2.0000, -1.0000, \
                        0.00000, 1.0000, 2.0000, 3.0000, 4.0000, \
                        -5.0000, -4.0000, -3.0000, -2.0000, -1.0000, \
                        0.00000, 1.0000, 2.0000, 3.0000, 4.0000, \
                        -5.0000, -4.0000, -3.0000, -2.0000, -1.0000, \
                        0.00000, 1.0000, 2.0000, 3.0000, 4.0000
                        };
        struct ga_pop *p = ga_pop_init(500, d1, 30, simple_fitness,
                                        simple_random);

        i = 0;
        int bored = 0;
        double min_fit;
        while(!bored)
        {
                min_fit = ga_pop_generational(&p);
                //min_fit = ga_pop_steady_state(&p);
                int mini = ga_pop_get_min_fitness_index(p);
                printf("Generation: %d, min fitness [%d]: %f\n",
                                                i, mini, min_fit);
                ga_i_print_attr(p->iarray[mini]);

                i++;

                if(min_fit == 0.0 || i > 100000)
                {
                        bored = 1;
```

```
                    }
            }

            return(0);

}


void ga_main(int N_params, double *x, double **XR, double **YR, double **XL,
                                            double **YL)
{
            int i;
            int j;
            int ngen = 0;
            int pop_size = 10;
            const int nattr = 96;
            int bored = 0;
            double min_fit;

            // ga setting setup
            GA_LBOUND = -5.000;
            GA_UBOUND = 100.00;
            GA_DIFF_SCALE = 800.000;
            K_MUT = 96;
            K_SELECT = 1; // select best from all population
            K_FLIP_XOVER = 3;

            //set the global muscle data structure so they can be accessed
            // by the fitness function
            XR_GLOBAL = XR;
            XL_GLOBAL = XL;
            YR_GLOBAL = YR;
            YL_GLOBAL = YL;
            N_MUS_GLOBAL = N_mus;
            N_EMG_STEP_GLOBAL = N_emg_step;
            N_PARAMS_GLOBAL = N_params;

            /*
            GA_LBOUND = -5.12;
            GA_UBOUND = 5.12;
            K_MUT = 5;
            GA_DIFF_SCALE = 10.000;
            ga_test();
            return;
            */
```

```
printf("GA: %d individuals, %d attributes\n", pop_size, nattr);

//x is what we will optimize. copy in nattr of x, to be sure of size
double d1[nattr];
for(i = 0; i < nattr; i++)
{
        d1[i] = x[i];
}

FILE *fp;
char *fname = "ga_r_data.dat";
fp = fopen(fname, "w");
fprintf(fp, "min_fit\n");
fclose(fp);

//init ga pop
// pop size = N_mus, nattr = N_emg_step
struct ga_pop *p = ga_pop_init(pop_size, d1, nattr, muscle_fitness,
                                simple_random);

ngen = 0;
while(bored == 0)
{
        //do our ga
        min_fit = ga_pop_generational(&p);
        //min_fit = ga_pop_steady_state(&p);
        int mini = ga_pop_get_min_fitness_index(p);
        printf("Generation: %d, min fitness [%d]: %f\n",
                                        ngen, mini, min_fit);
        //ga_i_print_attr(p->iarray[mini]);
        ga_pop_print_individuals(p);

        //set x to the best / min fitness (min fitness is min cost)
        x = p->iarray[mini]->attr;

        //write out the min to an R data file
        fp = fopen(fname, "a");
        fprintf(fp, "[%d] %f\n", ngen + 1, min_fit);
        fclose(fp);

        ngen++;

        if(min_fit <= 0.0 || ngen > 100000)
        {
```

```
                        bored = 1;
                }
        }
}
```

6.3 ga_i.h

```
#ifndef _GA_I_H
#define _GA_I_H

//individuals' attribute vector size
#define GA_MAX_ATTR_SIZE 500

//genetic algorithm individual
struct ga_i
{
        double attr[GA_MAX_ATTR_SIZE];
        int nattr;
        double (*random)(double); //how to create random values for attrs
        double fitness_cache;   //so we don't have to call a fitness
                                                        // function so much
};


struct ga_i *ga_i_init(double *, int);
void ga_i_copy(struct ga_i**, struct ga_i**);
int ga_i_del(struct ga_i**);

void ga_i_2p_crossover(struct ga_i **, struct ga_i **);
void ga_i_flip_crossover(struct ga_i **, struct ga_i **);
void ga_i_uniform_mutation(struct ga_i**);

void ga_i_print_attr(struct ga_i*);
#endif
```

6.4 ga_i.cpp

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

#include "ga_i.h"

//mutation selection, % to mutate = k = nattr / K_MUT
int K_MUT = 1;
```

```
//how many points flip crossover will do
int K_FLIP_XOVER = 0;

struct ga_i *ga_i_init(double *darray, int nattr)
{
        if(nattr > GA_MAX_ATTR_SIZE)
        {
                fprintf(stderr,
                        "ERROR: attributes of %d too big, max size is %d. Exiting.\
                exit(1);
        }

        struct ga_i *retval = (struct ga_i*)malloc(sizeof(struct ga_i));

        //Init attr
        retval->nattr = nattr;

        int i;
        for(i = 0; i < nattr; i++)
        {
                if(darray == NULL)
                {
                        retval->attr[i] = 0;
                }
                else
                {
                        retval->attr[i] = darray[i];
                }
        }

        return(retval);
}


void ga_i_copy(struct ga_i **to, struct ga_i **from)
{
        if((*from) == NULL)
        {
                fprintf(stderr,
                        "ERROR: copying individual, 'from' is null. Exiting.\n");
                exit(1);
        }

        //init memory
        (*to) = (struct ga_i*)malloc(sizeof(struct ga_i));
        //set attrs
        int i;
        for(i = 0; i < (*from)->nattr; i++)
        {
                (*to)->attr[i] = (*from)->attr[i];
        }

        //copy nattr
        (*to)->nattr = (*from)->nattr;

        //set the random function pointer
        (*to)->random = (*from)->random;
}


int ga_i_del(struct ga_i **p)
{
        free((*p));
        (*p) = NULL;

        return(0); //no way to check free
}


void ga_i_2p_crossover(struct ga_i **i1, struct ga_i **i2)
{
        //Sanity checks
        //nattrs equal
        if((*i1)->nattr != (*i2)->nattr)
        {
                fprintf(stderr,
                "ERROR: individuals have different nattr during 2p crossover. Exiti
                exit(1);
        }

        //if nattrs == 0, random point gen will infinite loop
        if((*i1)->nattr <= 0)
        {
                fprintf(stderr,
                        "ERROR: zero / negative nattr values during 2p crossover. I
                exit(1);
        }

        /* initialize random seed: */
        //srand ( time(NULL) );

        //Crossover points
        int p1 = 0;
        int p2 = 0;
        while(p1 == p2)
        {
                //+1 to ignore rand() == 0. each are decremented again below
                p1 = rand() % (*i1)->nattr + 1;
                p2 = rand() % (*i1)->nattr + 1;
        }

        //if p1 > p2, swap so p1 is the first index / point
        if(p1 > p2)
        {
                int temp = p1;
                p1 = p2;
                p2 = temp;
        }

        //decrement each index by 1 for the 0..n-1 array indexing
        p1--;
        p2--;

        //Middle segment crossover
        int i;
        for(i = p1; i < p2; i++)
        {
                double old_i1_attr_val = (*i1)->attr[i];
                (*i1)->attr[i] = (*i2)->attr[i];
                (*i2)->attr[i] = old_i1_attr_val;
        }
}


void ga_i_flip_crossover(struct ga_i **i1, struct ga_i **i2)
{
        //Sanity checks
        //nattrs equal
        if((*i1)->nattr != (*i2)->nattr)
        {
                fprintf(stderr,
                "ERROR: individuals have different nattr during 2p crossover. Exiti
                exit(1);
        }
        //if nattrs == 0, random point gen will infinite loop
        if((*i1)->nattr <= 0)
        {
                fprintf(stderr,
                        "ERROR: zero / negative nattr values during 2p crossover. E
                exit(1);
        }

        //pick k random points
        int i;
        for(i = 0; i < K_FLIP_XOVER; i++)
        {
                //flip a random point between the two individuals
                int point = rand() % (*i1)->nattr;
                double temp = (*i1)->attr[point]; //stash i1's random point value
                (*i1)->attr[point] = (*i2)->attr[point];
                (*i2)->attr[point] = temp;
        }
}


void ga_i_uniform_mutation(struct ga_i **p)
{
        if((*p) == NULL)
        {
                fprintf(stderr,
                        "ERROR: trying to mutate a null individual. Exiting.\n");
                exit(1);
        }

        int i;
        int k = (*p)->nattr / K_MUT; //k = % of all the attributes
        if(k <= 0)
        {
                fprintf(stderr,
                        "ERROR: Mutation k = %d, too small for nattr %d and K_MUT %
                        k, (*p)->nattr, K_MUT);
                exit(1);
        }

        for(i = 0; i < k; i++)
        {
                //select a random attribute
                int j = rand() % (*p)->nattr;
                //get a random value
                double drand = (*p)->random((*p)->attr[j]);

                //make the mutation
                (*p)->attr[j] = drand;
        }
}


void ga_i_print_attr(struct ga_i *p)
{
        int i;
        for(i = 0; i < p->nattr; i++)
        {
                printf("%10f", p->attr[i]);
                if( ((i+1) % 5) == 0 )
                {
                        printf("\n");
                }
        }
        printf("\n");
}
```

6.5 ga_pop.h

```
#ifndef _GA_POP_H
#define _GA_POP_H

#include <stdlib.h>

#include "ga_i.h"

#define GA_MAX_POP_SIZE 10000

//Elitism - comment out if not wanted
#define GA_ELITISM


struct ga_pop
{
        struct ga_i *iarray[GA_MAX_POP_SIZE]; //individual array
        int ni; //n individuals
        double (*fitness)(struct ga_i*); //the fitness function
};
```

```
struct ga_pop *ga_pop_init(int, double *, int,
                                double (* fitness)(struct ga_i*),
                                double (* random)(double));
void ga_pop_del(struct ga_pop**);

int ga_pop_get_min_fitness_index(struct ga_pop *);
void ga_pop_get_max_fitnesses(struct ga_pop *, int*, int*);
void ga_pop_recalc_fitness_cache(struct ga_pop **);

double ga_pop_steady_state(struct ga_pop **);
double ga_pop_generational(struct ga_pop **);

void ga_pop_print_individuals(struct ga_pop*);
#endif
```

6.6 ga_pop.cpp

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

#include "ga_i.h"
#include "ga_pop.h"

//tournament selection, % to select for subpool for tournament selection
// % = k = (pop)size / K_SELECT
int K_SELECT = 1;

using namespace std;


struct ga_pop *ga_pop_init(int size, double *darray,
                                int nattr,
                                double (* fitness)(struct ga_i*),
                                double (* random)(double) )
{
        int i;
        ga_pop *retval = (ga_pop*)malloc(sizeof(ga_pop));

        //Required values
        if(fitness == NULL)
        {
                fprintf(stderr,
                        "ERROR: you must register a fitness function. Exiting.\n");
                exit(1);
        }
        if(random == NULL)
        {
                fprintf(stderr,
                        "ERROR: you must register a random function. Exiting.\n");
                exit(1);
        }
        if(size > GA_MAX_POP_SIZE)
        {
                fprintf(stderr,
                        "ERROR: size of %d too big, max size is %d. Exiting.\n", G
                exit(1);
        }

        //Init inidividuals
        retval->ni = size;

        for(i = 0; i < size; i++)
        {
                retval->iarray[i] = ga_i_init(darray, nattr);

                //Init the random function pointer
                retval->iarray[i]->random = random;
        }

        //Init the fitness function pointer
        retval->fitness = fitness;

        //Init the fitness caches
        ga_pop_recalc_fitness_cache(&retval);

        return(retval);
}


void ga_pop_del(ga_pop **p)
{
        int i;
        for(i = 0; i < (*p)->ni; i++)
        {
                ga_i_del(&(*p)->iarray[i]);
        }

        free((*p));
        (*p) = NULL;
```

```
}


//TODO: cleanup
void ga_pop_tournament_selection(int indices[2], ga_pop *p)
{
        /* initialize random seed: */
        //srand ( time(NULL) );

        int i;
        int j;
        //k = % of all the attributes
        int k = p->ni / K_SELECT;
        //int subset_indices[k];
        vector<int> subset_indices(k);

        if(k <= 1)
        {
                fprintf(stderr,
                        "ERROR: Selection k = %d, too small for (pop)size %d and K_
                        k, p->ni, K_SELECT);
                exit(1);
        }

        //create a subset of indices for min1 and min2 selection
        for(i = 0; i < k; i++)
        {
                j = rand() % p->ni;
                subset_indices[i] = j;
        }

        //Select min1
        j = subset_indices[0];
        indices[0] = j;
        //double min1 = p->fitness(p->iarray[j]);
        double min1 = p->iarray[j]->fitness_cache;
        for(i = 1; i < k; i++)
        {
                j = subset_indices[i];
                //if(p->fitness(p->iarray[j]) < min1)
                if(p->iarray[j]->fitness_cache < min1)
                {
                        //min1 = p->fitness(p->iarray[j]);
                        min1 = p->iarray[j]->fitness_cache;
                        indices[0] = j;
                }
        }

        //Select min2
        i = 0;
        j = subset_indices[i];
        //if min1's index is 0, then set min2's index to 1
        //double min2 = p->fitness(p->iarray[j]);
        double min2 = p->iarray[j]->fitness_cache;
        indices[1] = j;
        if(j == indices[0])
        {
                i = 1;
                j = subset_indices[i];
                //min2 = p->fitness(p->iarray[j]);
                min2 = p->iarray[j]->fitness_cache;
                indices[1] = j;
        }
        for(; i < k; i++)
        {
                j = subset_indices[i];
                if(p->iarray[j]->fitness_cache < min2
                        && j != indices[0])
                {
                        min2 = p->iarray[j]->fitness_cache;
                        indices[1] = j;
                }
        }
}


int ga_pop_get_min_fitness_index(struct ga_pop *p)
{
        if(p == NULL)
        {
                fprintf(stderr,
                        "ERROR: trying to get min fit index of null pop. Exiting.\r
                exit(1);
        }

        double min = p->iarray[0]->fitness_cache;
        int mini = 0;
        int i;
        for(i = 0; i < p->ni; i++)
        {
                if(p->iarray[i]->fitness_cache < min)
                {
                        min = p->iarray[i]->fitness_cache;
                        mini = i;
                }
        }

        return(mini);
}


void ga_pop_recalc_fitness_cache(struct ga_pop **p)
{
        int i;

        for(i = 0; i < (*p)->ni; i++)
        {
                (*p)->iarray[i]->fitness_cache = (*p)->fitness((*p)->iarray[i]);
        }
}


double ga_pop_get_min_fitness(struct ga_pop *p)
{
        if(p == NULL)
        {
                fprintf(stderr,
                        "ERROR: trying to get min fit of null pop. Exiting.\n");
                exit(1);
        }


        int i = ga_pop_get_min_fitness_index(p);
        double fit = p->iarray[i]->fitness_cache;

        return(fit);
}


void ga_pop_get_max_fitnesses(struct ga_pop *p, int *max1, int *max2)
{
        int i;

        if(p->ni < 2)
        {
```

```
        fprintf(stderr,
                "ERROR: pop size of %d is too small. Exiting\n", p->ni);
        exit(1);
}

//Init indexes and values
(*max1) = 0;
(*max2) = 1;
double max1_d = p->iarray[(*max1)]->fitness_cache;
double max2_d = p->iarray[(*max2)]->fitness_cache;

//switch if max2 greater
if(max2_d > max1_d)
{
        (*max1) = 1;
        (*max2) = 0;
        max1_d = p->iarray[(*max1)]->fitness_cache;
        max2_d = p->iarray[(*max2)]->fitness_cache;
}

//find max1
for(i = 0; i < p->ni; i++)
{
        if(p->iarray[i]->fitness_cache > max1_d)
        {
                (*max1) = i;
                max1_d = p->iarray[(*max1)]->fitness_cache;
        }
}

//find max2
for(i = 0; i < p->ni; i++)
{
        if(p->iarray[i]->fitness_cache > max2_d
                && i != (*max1) )
        {
                (*max2) = i;
                max2_d = p->iarray[(*max2)]->fitness_cache;
        }
}

//this shouldn't happen, but if it does, lets kill it
if((*max1) == (*max2))
{
        fprintf(stderr,
                    "ERROR: max's are the same. Exitting\n");
                exit(1);
        }
}


//returns minimum fitness
double ga_pop_steady_state(struct ga_pop **p)
{
        int indices[2];
        ga_pop_tournament_selection(indices, (*p));

        int i1 = indices[0];
        int i2 = indices[1];

        #ifdef GA_ELITISM
        //preserve the best
        struct ga_i *i1p;
        struct ga_i *i2p;
        ga_i_copy(&i1p, &((*p)->iarray[i1]));
        ga_i_copy(&i2p, &((*p)->iarray[i2]));
        #endif

        //Crossover
        //ga_i_2p_crossover(&((*p)->iarray[i1]), &((*p)->iarray[i2]));

        //Mutate
        ga_i_uniform_mutation(&(*p)->iarray[i1]);
        ga_i_uniform_mutation(&(*p)->iarray[i2]);

        //Recalculate fitness
        // we're steady state, so we only need to recalce the
        // 2 xover/mutated
        //ga_pop_recalc_fitness_cache(&(*p));
        (*p)->iarray[i1]->fitness_cache = (*p)->fitness((*p)->iarray[i1]);
        (*p)->iarray[i2]->fitness_cache = (*p)->fitness((*p)->iarray[i2]);

        //Elitism
        #ifdef GA_ELITISM
        int maxi1;
        int maxi2;
        //find max's (worst)
        ga_pop_get_max_fitnesses((*p), &maxi1, &maxi2);
        //delete them
        if(i1 != maxi1)
        {
                ga_i_del(&((*p)->iarray[maxi1]));
                ga_i_copy(&((*p)->iarray[maxi1]), &i1p);
        }
        if(i2 != maxi2)
        {
                ga_i_del(&((*p)->iarray[maxi2]));
                ga_i_copy(&((*p)->iarray[maxi2]), &i2p);
        }

        ga_i_del(&i1p);
        ga_i_del(&i2p);
        #endif

        double pop_min_fit = ga_pop_get_min_fitness((*p));

        return(pop_min_fit);
}


//returns minimum fitness
double ga_pop_generational(struct ga_pop **p)
{
        int i = 0;

        int indices[2];
        ga_pop_tournament_selection(indices, (*p));

        int i1 = indices[0];
        int i2 = indices[1];

        //Elitism
        #ifdef GA_ELITISM
        //preserve the best
        struct ga_i *i1p;
        struct ga_i *i2p;
        ga_i_copy(&i1p, &((*p)->iarray[i1]));
        ga_i_copy(&i2p, &((*p)->iarray[i2]));
        #endif

        //Crossover
        //ga_i_2p_crossover(&((*p)->iarray[i1]), &((*p)->iarray[i2]));
        ga_i_flip_crossover(&((*p)->iarray[i1]), &((*p)->iarray[i2]));

        //Mutate
```

```
//ga_i_uniform_mutation(&(*p)->iarray[i1]);
//ga_i_uniform_mutation(&(*p)->iarray[i2]);

//Regenerate all
for(i = 0; i < (*p)->ni; i++)
{
        if(i == i1)
        {
                continue;
        }

        //Regenerate each
        ga_i_del(&((*p)->iarray[i]));
        (*p)->iarray[i] = NULL;
        ga_i_copy( &((*p)->iarray[i]), &((*p)->iarray[i1]));

        //Mutate
        ga_i_uniform_mutation(&(*p)->iarray[i]);
}

ga_pop_recalc_fitness_cache(&(*p));

//Elitism
#ifdef GA_ELITISM
int maxi1;
int maxi2;
//find max's (worst)
ga_pop_get_max_fitnesses((*p), &maxi1, &maxi2);
//delete them
if(i1 != maxi1)
{
        ga_i_del(&((*p)->iarray[maxi1]));
        ga_i_copy(&((*p)->iarray[maxi1]), &i1p);
}
if(i2 != maxi2)
{
        ga_i_del(&((*p)->iarray[maxi2]));
        ga_i_copy(&((*p)->iarray[maxi2]), &i2p);
}

ga_i_del(&i1p);
ga_i_del(&i2p);
#endif

double pop_min_fit = ga_pop_get_min_fitness((*p));
```

```
        return(pop_min_fit);
}



void ga_pop_print_individuals(struct ga_pop *p)
{
        int i;
        for(i = 0; i < p->ni; i++)
        {
                printf("%d: %f\n", i, p->iarray[i]->fitness_cache);
                //Yikes, TMI
                //ga_i_print_attr(p->iarray[i]);
        }
}
```

7 Bibliography

References Cited

[1] Neptune, R.R.; McGowan, C.P. ”Muscle contributions to whole-body sagittal plane angular momentum during walking” *Journal of Biomechanics, 2011* . 44 612 : Print.

[2] Goffe, L.G; Ferrier, G.D.; Rogers, J. ”Global optimization of statistical function with simulated annealing” *Journal of Econometrics, 1994* . 60 65-99 : Print.

[3] Neptune, R.R.; McGowan, C.P.; Kautz, S.A. ”Forward Dynamics Simulations Provide Insight Into Muscle Mechanical Work During Human Locomotion” *Exerc. Sport Sci. Rev., 2009* . Vol. 37 No. 4 203-210 : Print

[4] Amdahl, Gene (1967). ”Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities”. *AFIPS Conference Proceedings* (30): 483485. Print.