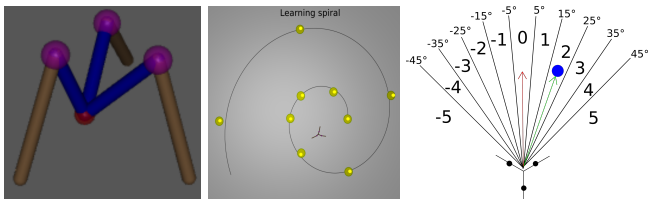
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2018
http://excel.fit.vutbr.cz

Evolution of programs controlling simple robot model

Jakub Fajkus*



Abstract
The aim of this work is to utilize Evolutionary Algorithms for finding computer programs in order to control simple robot models. A model is placed in the physical simulation where it is supposed to move along given specified reference points. The evolution is given a task to evolve a program that will result in a robot moving along a specified trajectory. The program that controls the model consists of application specific instructions and its design is inspired by Linear Genetic Programming. The program has an information about a direction to the next reference point.

Keywords: Evolutionary computation — Linear Genetic Programming — Robotics

Supplementary Material: *N/A*

*xfajku06@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

In the area of robotics, it is very important to be able to create prototypes quickly and cheaply. For this purpose, it is beneficial to use computer models and simulation when designing the structure of the robot. There are many ways to control the robot depending on the given requirements. It can be, for example, controlled by instructions of an imperative language [1], finite state machines [2], classical control theory [3] or artificial neural networks [4] [5]. For a fast prototyping, we can use Evolutionary Algorithms to evolve the robot control.

The problem to solve in this work is defined as follows. We have a model of a simple robot and we want to evolve a program that is based on instructions of an imperative language. The robot is supposed to move in a simulated environment and visit the predefined reference points. The reference points, specified by the designer, define a trajectory the robot should follow. The program that controls the model is being evolved using Evolutionary Algorithms and then evaluated in the Mujoco simulator [6].

There is a number of works using Evolutionary Algorithms to evolve controllers based on artificial neural networks [7] [8]. There are also works that are using Genetic Programming [9]. The most similar work to this one is the work of Wolff and Wahde [1]. They are using Linear Genetic Programming to evolve a bipedal locomotion of a humanoid model.

The research conducted in this paper is primarily motivated by [1] where a successful evolution of control algorithms was proposed for a specific humanoid model. In this paper, we present an extended concept which allows evolutionary design of controllers for various creatures with a different number of legs. The goal is to evolve a program using a very reduced introduction. set in which several sub-programs can be evolved automatically and simultaneously in order to perform specific motions of the model. Our approach will be evaluated using several trajectories the model should

follow in a simulated environment. The detail of the proposed system will be described in Section 3.2.

Here, we will describe the robot's model structure (Figure 1).

A following robot model will be considered in the work. The robot model has three legs (brown), all connected via joints with the core (blue). The joints are located under the purple spheres. All of the joints are actuated and their range is limited within 50°. The core of the robot contains a head (red), that is used for calculating the distance between the robot and the reference points. The purple spheres are used for collision detection – if they collide with the ground, the simulation is terminated.

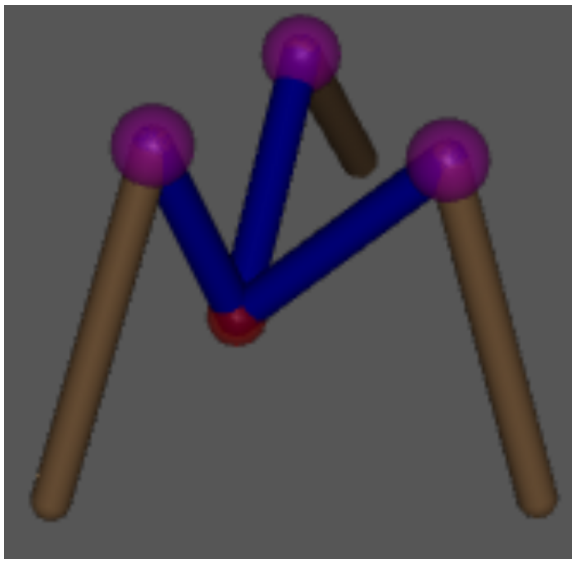


Figure 1. Detail of the three leg robot.

2. Linear Genetic Programming

Here we will discuss a brief introduction to Linear Genetic Programming (LGP), based on [10]. LGP is a special variant of the Genetic Programming [11].

Genetic Programming (GP) solves a modelling problem. That means we know a set of inputs to a system and the outputs it should produce. But we do not have the system that will compute the inputs and tell us the output. When solving a control problem, we are looking for such a function, that will lead to the desired behavior.

In GP we are evolving computer program P that represents a function: $f : I^n \rightarrow O^m$, where I^n denotes the n-dimensional input data and O^m denotes the m-dimensional output data.

The genotype space G in GP includes all programs that can be composed of elements of a given programming language L. The programming language L is defined by an instruction set and a terminal set. An interpreter translates the genotype representation into the phenotype, i.e. the behavior of the robot. The phenotype is then executed and its fitness is evaluated using a software physical simulator.

The original GP uses trees that correspond to expressions from a functional programming language. The nodes of the tree represent functions, while leaves represent input values or constants.

The Linear Genetic Programming is a variant where the programs are composed of a sequence of instructions from an imperative language or a machine code. Each program has available a predefined set of registers that can hold constants as well as results of instructions. Those registers are often divided into groups: input registers, output registers, and calculation registers. The instructions are composed of an operation and one or more operands. The operands may be registers or constants. As the program is being executed it modifies the values of the registers, reads the inputs and writes the outputs.

3. Evolution of Robot Controller using LGP

In order to evolve a robot controller, the LGP concept was combined with an evolutionary algorithm and the Mujoco simulator [6] was used to evaluate the candidate programs using a built-in interpreter. A scene is prepared in the simulator consisting of a set of reference points defining a spiral (Figure 2). The robot model is given a task (using the evolved LGP programs) to move from point to point at given order and to get as close as possible to each of them. This way the robot moves on a predefined trajectory.

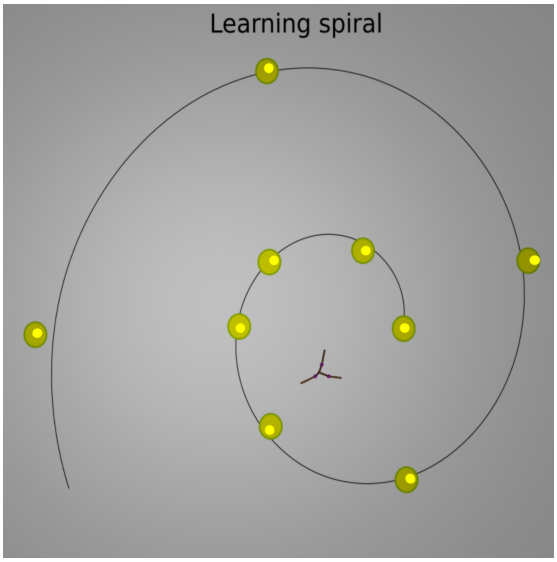


Figure 2. Top view at the scene in the simulator. We can see a spiral on which are placed the reference points (yellow circles).

3.1 LGP-Based Robot Controller

The programs (which are the subject of evolution using LGP) are composed of an application-specific instruction of the form `COPY ARG1 ARG2`, where `COPY` is the instruction name, `ARG1` is source register and `ARG2` is the destination register. Each register has its unique identification called *index* which is used in an instruction's argument. As the program is being executed, it copies values from input or constant registers into the output registers.

The interpreter has 11 constant registers (indices 0–10) holding integer values from -5 to 5, 2 input registers (indices 11–12) and 3 output registers (indices 13–15). The values of the input registers depend on the direction to the next reference point as follows.

The space around the robot is divided into circular sectors (Figure 3). The borders between the sectors are defined by an angle from the robot's heading vector. Each sector has a value of integer between -5 and 5. The value of the input registers depends on the value of the sector in which the next reference point is located. At the Figure 3 we can see a situation, where the robot (at the bottom) is heading to the top and the next reference (blue) is located in the sector with a value of 2. This number is inserted into the first input register. Next, the second register is filled with a value, that has an inverted sign (-2 in this case).

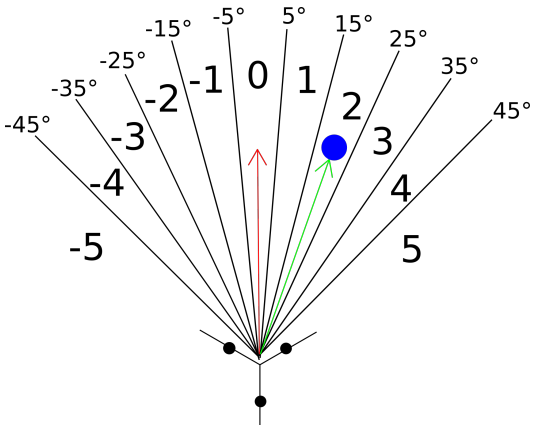


Figure 3. Calculation of the input register values.

Each output register is connected with exactly one robot's joint and its value is converted into a force that is applied in the joint.

For the purposes of this work, a concept of subprograms has been introduced which works as follows.

The individual's genotype (representing the whole program) is split into 3 smaller subprograms - *init*, *event* and *main*.

The *main* subprogram is being run in an endless loop during the simulation. Only one instruction is executed at a time. Period of 0.3 seconds between instructions is used.

The *init* subprogram is executed at the start of the simulation and its purpose is to set the initial rotation of all robot's joints. All instructions of the init program are executed in a single time. After that, the main subprogram is paused for one second to give the robot's joints time to finish the rotation.

The *event* subprogram is run each time the robot gets close enough to some of the reference points (but only once for each point). Its purpose is to change the robot's joints rotation as a preparation to move to the next reference point. The event subprogram is executed in a single time and the main subprogram is paused as well as in the *init* subprogram.

A schema of the interpreter is in Figure 4. Each 0.3 seconds an instruction from the *main* subprogram is executed. The first argument is an input or constant register index, the other one is the output register index. A value from constant or input register is then copied into the output register.

Each of the interpreter's output registers is mapped to exactly one robot's joint. Each time an instruction is executed, the values from the output registers are read. The values are then used as a control signal to the simulator for each joint in the model. A value (control signal) holds two pieces of information. The sign of the value is used to determine the direction of the force applied (clockwise, counterclockwise). The absolute value of the number is used to determine the magnitude of the force. Based on those signals and the model parameters the simulator calculates a force that will be applied in the joint.

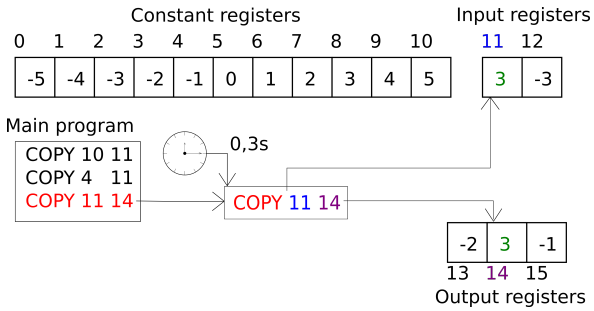


Figure 4. Schematic view of the interpreter. The situation depicted here is as follows. The instruction to be executed has arguments with values of 11 and 14. A value from the input register with index 11 is copied into the output register with index 14.

3.2 Evolution of robot controllers

In order to design the LGP-based programs to control the robot, a simple steady-state Genetic Algorithm (GA) was implemented – see Figure 5. The parameters of the GA are as follows.

- Population size is set to 1000.

- A steady-state population model is used (the better half of the population is kept, the other one is replaced).
- The genotype length is fixed to 36 instructions.
- A custom genetic crossover based on the uniform crossover is used (Figure 7).
- The probability of crossover is 80%.
- The mutation is set with 100% probability and changes only one gene.
- Parent selection is implemented by a tournament of size 2.
- The evolution is terminated after 300 generations.

```
//the T has a size of k
set time t = 0;
initialize all chromosomes in P(t) with random alleles;
do {
  calculate fitness of individuals in P(t);
  do {
    select two individuals from the parent population P(t);
    create two offspring using a crossover operator;
    apply a random mutation to the offspring;
    insert the offspring into temporary population T;
  } until(T has size of k);
  replace k worst individuals in P(t) with T
  t = t + 1;
} until (termination criterion is met);
```

Figure 5. Steady-state algorithm

Each chromosome in the GA represents a single program (a candidate controller for the robot). The structure of the chromosome is illustrated in Figure 6. The chromosome has fixed length (36 genes). It is split into three smaller subprograms (init, event, main) that have fixed lengths, too. Each gene represents a single instruction. The instruction is a triplet and consists of the instruction name and two numbers.

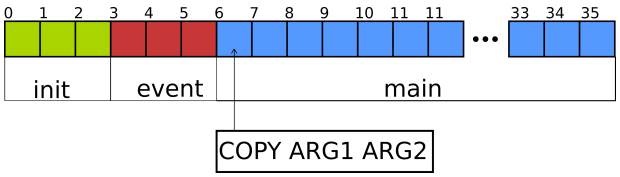


Figure 6. The structure of the genotype.

The GA applies both the crossover and mutation operators which work as follows.

The crossover operates on the level of subprograms. It is based on the uniform crossover with $p = 0.5$ but instead of swapping the genes, it swaps the whole subprograms (Figure 7).

The mutation operates on the level of genes (instructions). There is an 80% chance of mutating the instruction values. There is a 20% chance of replacing the instruction with a new, randomly generated one. When mutating the instruction values, there is an equal chance to mutate the first or the second argument. When a value is mutated, it is replaced by a new value from the specified range of values.

The fitness evaluation of the candidate programs is performed as follows. For each program, a new simulation is run. During the simulation, the program is being executed and it controls the robot's model (as explained in the previous section) as well as the minimal distance D_i between each of the reference points and the robot is being recorded. $D_i = \min F(R, P_i)$, where F denotes the distance function, R is the robot position, P_i is a reference point position and $i = 1, \cdots N$, where N is the number of reference points.

When the simulation run is finished, the fitness value is calculated. For each reference point a score is calculated as follows:

$$S_i = \begin{cases} t - D_i & \text{if } D_i \leq t \\ 0 & \text{else} \end{cases}$$

The parameter t is a threshold – value that determines the minimal distance, at which the score is calculated. The fitness f is then calculated as a sum of scores for all reference points:

$$f = \sum_{i=1}^{N} S_i$$

The threshold parameter was set to 40, based on the dimensions of the simulation and the robot itself. The maximum fitness in the simulation can be calculated as follows:

$$f_{max} = t \cdot N$$

Given 9 reference points on the spiral and the threshold parameter set to 40, the maximum fitness is equal to 360. This way, the evolution maximizes the score and thus minimizes the distance between the robot and each of the reference points.

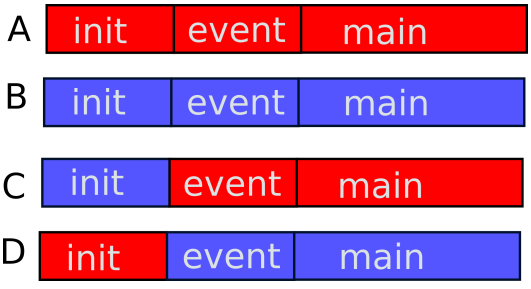


Figure 7. Here, the crossover of parents A and B resulted in offspring C and D. In this case, only the init subprograms were swapped.

4. Results

The evolution was run 20 times. The typical progress of fitness values can be seen in Figure 9.

The maximum fitness possible in the simulation was set to 360, although the simulation time (600s) was not enough to gain the maximum fitness value. The fitness of the best solution the GA was able to find was equal to 255.9. After all the evolutionary runs, the best solution was evaluated over longer simulation time and its fitness was equal to 330.9, which is 92% of the maximum fitness value possible.

The trajectory of the best solution can be seen in Figure 9.

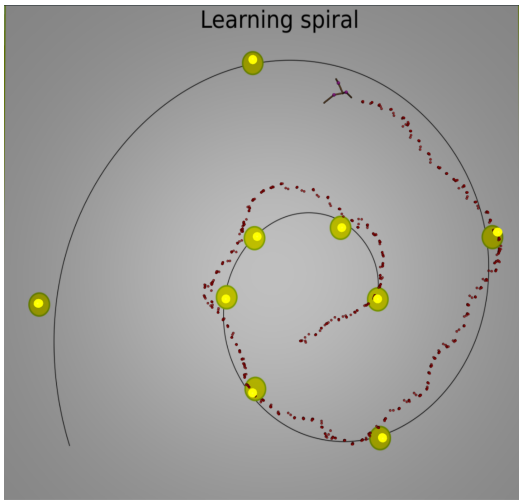


Figure 8. Trajectory of the best solution

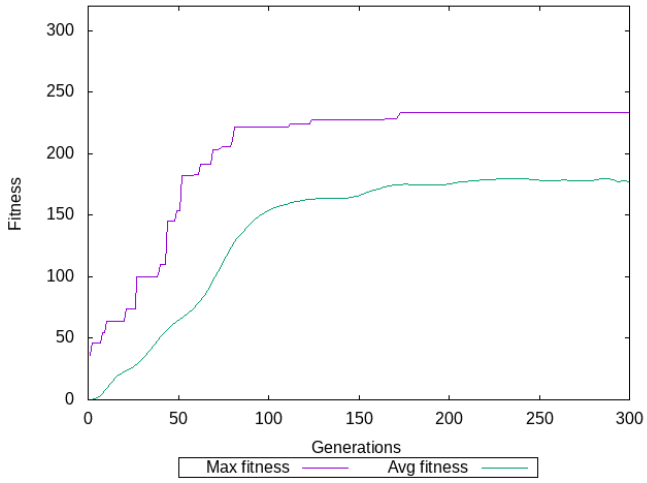


Figure 9. A typical progress of fitness values in an evolutionary run.

5. Conclusions

This work showed an usage of Evolutionary Algorithms for finding a program for a robot control.

The results show that the evolution successfully found a relatively good solution to the problem. The simulation time was limited and the robot could not finish it in given amount of time. Despite that, when given enough time, the best solution was able to finish the simulation successfully. That is, in other words, the robot continued following the part of the spiral trajectory that it never saw before.

The future work will aim to evolve programs with an emphasis on the ability to follow before unseen trajectories.

Acknowledgements

I would like to thank my supervisor Michal Bidlo for his time and valuable advices.

References

[1] K. Wolff and M. Wahde. Evolution of biped locomotion using linear genetic programming, 10 2007.

[2] J. K. Hodgins. Three-dimensional human running. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 4, pages 3271–3276 vol.4, Apr 1996.

[3] T. Mita, T. Yamaguchi, T. Kashiwase, and T. Kawase. Realization of a high speed biped using modern control theory. *International Journal of Control*, 40(1):107–119, 1984.

[4] T. Reil and P. Husbands. Evolution of central pattern generators for bipedal walking in a real-time physics environment. *IEEE Transactions on Evolutionary Computation*, 6(2):159–168, Apr 2002.

[5] F. L. Lewis, A. Yesildirek, and K. Liu. Multilayer neural-net robot controller with guaranteed tracking performance. *IEEE Transactions on Neural Networks*, 7(2):388–399, March 1996.

[6] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, Oct 2012.

[7] R. D. Beer and J. C. Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1(1):91–122, 1992.

[8] S. S. Farooq and K. J. Kim. Evolution of neural controllers for simulated and real quadruped robots. In *2013 Second International Conference on Robot, Vision and Signal Processing*, pages 295–298, Dec 2013.

[9] J. Macedo, L. Marques, and E. Costa. Robotic odour search: Evolving a robot's brain with genetic programming. In *2017 IEEE International Conference on Autonomous Robot Systems*

and Competitions (ICARSC), pages 91–97, April 2017.

[10] M. F. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[11] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.