Chalmers University of Technology

FFR105 - Stochastic Optimization Algorithms

Problem set 2

Jacopo Credi
(910216-T396)

October 16, 2015

Problem 2.1 - The travelling salesman problem (TSP)

2.1(a)

In a TSP problem with N cities, a path can be constructed by picking the first city among N possibilities, the second among $N-1$ possibilities, and so on, until only 1 city is left. Therefore, a *valid* path can be constructed in $N!$ possible ways.

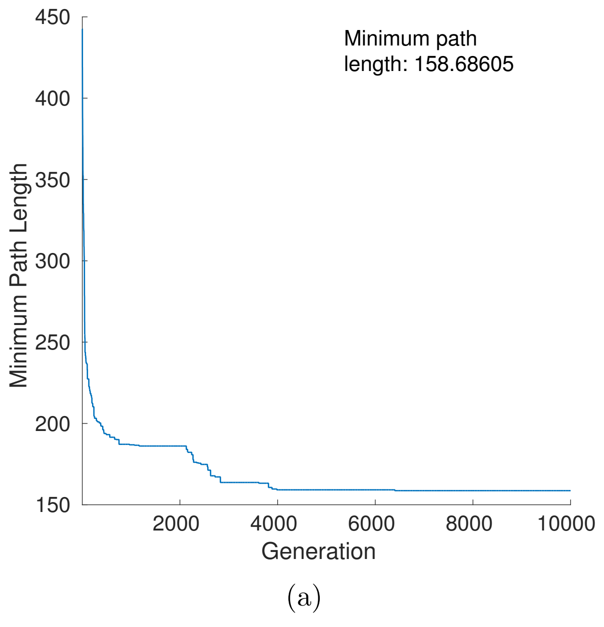
However, since paths are closed (i.e. they return to the original city), the starting node of the path does not matter, as long as the same cities are visited in the same order, thus we must divide the number of *valid* paths by a factor N to obtain the number of *distinct* paths. Furthermore, since paths going through a given sequence of cities in opposite order are also equivalent, this number must be also divided by two. Hence, the number of *distinct* paths in a TSP with N cities is

$$\frac{N!}{2N} = \frac{(N-1)!}{2} .$$

2.1(b)

A GA was implemented in MATLAB (see script `GA21b.mat` in folder `2.1/2.1b`) to solve the TSP problem with $N = 50$ cities whose locations were provided. Paths are encoded using permutation encoding and population is initialised randomly, generating M chromosomes containing random valid paths. The fitness of a chromosome is taken as the inverse of the corresponding path length. Selection is carried out using tournament selection, crossover is not used and mutations are implemented as swap mutations.

Several test runs were carried out to find a set of parameters suitable for the problem. Figure 1 shows the best result of a few long runs (10000 generations) of `GA21b.mat` with 100 individuals, tournament size $= 5$, $p_{\text{tour}} = 0.8$, $p_{\text{mut}} = 2/N$, and inserting one copy of the best path in the next generation. The length of the best path found with this algorithm was 158.69.



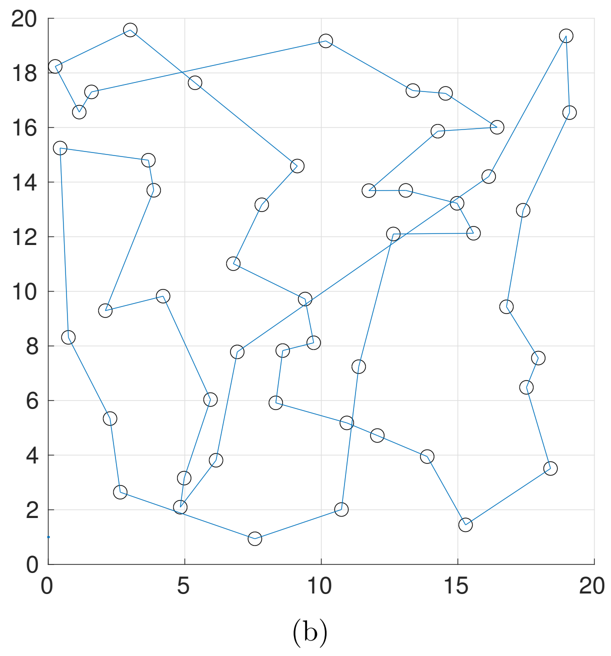


Figure 1: Best result found by `GA21b.mat` (GA for TSP with random initial population). Left panel: minimum path length vs. generations. Right panel: best path found (length $\simeq$ 158.69).

2.1(c)

An Ant System (AS) algorithm was implemented by completing the provided MATLAB script `AntSystem.m` and then applied to the same TSP as above.

After experimenting with several parameter sets, it was found that using a larger number of ants than N (number of cities) could help avoiding premature convergence, as more pheromone is deposited in each iteration. The path of length 122.44 shown in Figure 2 was obtained by running `AntSystem.m` with 100 ants and parameters $\alpha = 2,\ \beta = 1,\ \rho = 0.5$, after 107 iterations. The AS algorithm was thus able to find a better solution than the GA. The chromosome corresponding to this path is saved as a vector in file `bestPath21.m` (in main problem folder `2.1`), which can be run as a script and loads a vector called `bestPath` in the workspace.

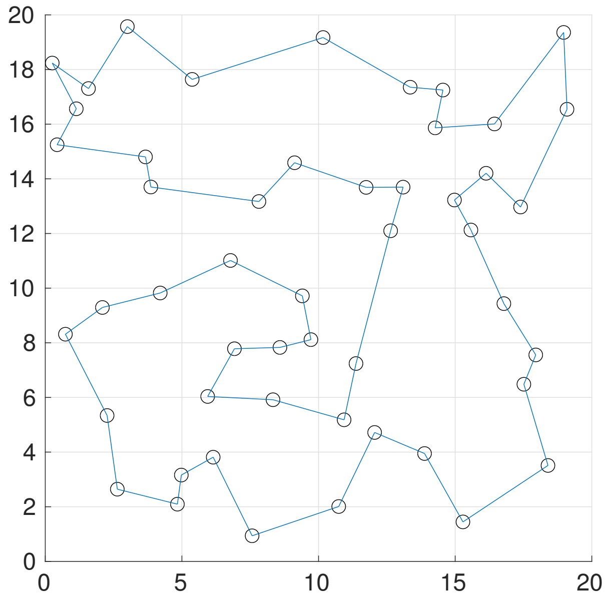


Figure 2: Best path found with Ant System algorithm. Length $\simeq$ 122.44.

2.1(d)

To allow a more fair comparison between GA and AS, the initialisation of the GA population was modified, so that starting path are nearest-neighbour paths (generated by starting from a random city) with a few random swap mutations (see function `InitializePopulation21d`).

Figure 2 shows the best result of a set of long runs (10000 generations) of the GA with 100 individuals and the same parameters as in 2.1(b), starting from a population of nearest-neighbour paths with 3 swap mutations each. The length of the best path found by `GA21d.mat` was 130.76. Thus, as expected, this implementation yields better results that the one in 2.1(b). However, AS implementation in 2.1(c) remains far superior for solving this combinatorial optimisation problem.

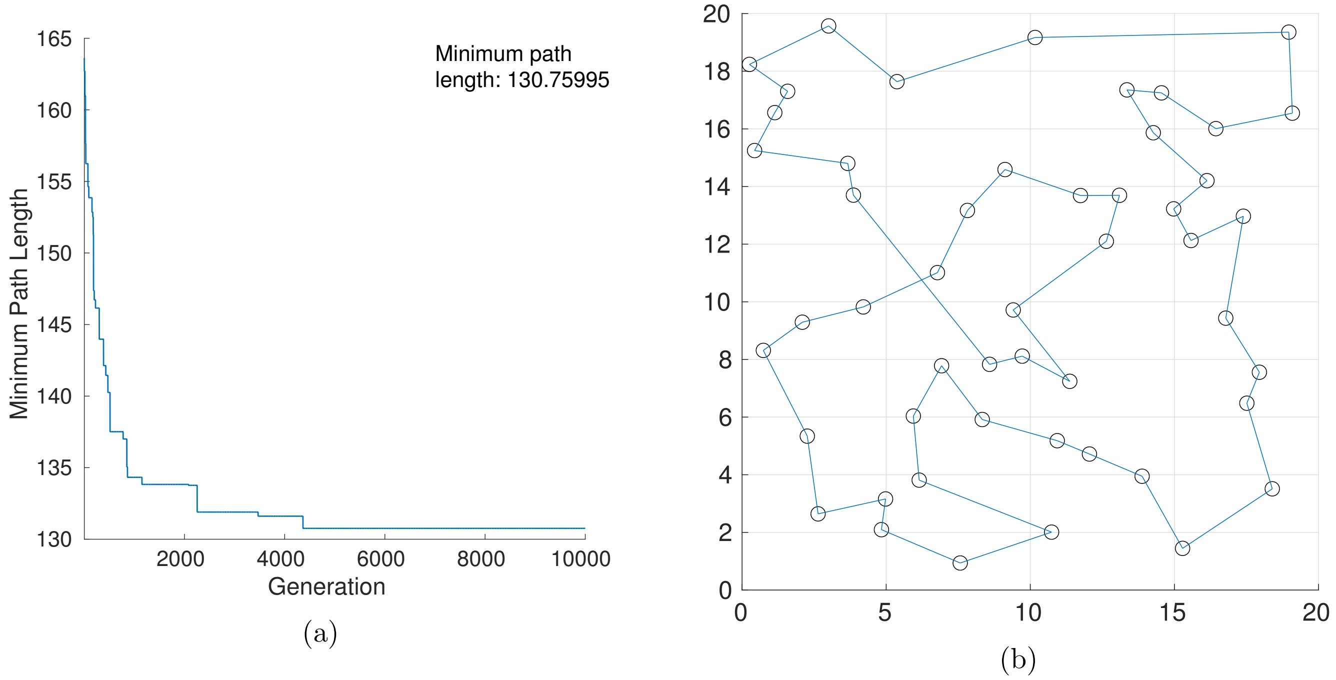


Figure 3: Best result found by `GA21d.mat` (GA for TSP with initial population of nearest-neighbour paths with 3 swap mutations). Left panel: minimum path length vs. generations. Right panel: best path found (length $\simeq$ 130.76).

Problem 2.2 - Particle swarm optimisation

2.2(a)

MATLAB script `PSO22a` in folder `2.2/2.2a` contains an implementation of a standard PSO algorithm for function minimisation, with inertia weights and no craziness operator.

The program was used to find the minimum of the function

$$f(x,y) = 1 + (-13 + x - y^3 + 5y^2 - 2y)^2 + (-29 + x + y^3 + y^2 - 14y)^2 ,$$

with x and y real values in $[-10, 10]$.

After a few runs to explore the parameter space, the parameter set in Table 1 was used to find the minimum

$$(x^*, y^*)^T = (5,\ 4)^T , \qquad \text{with} \qquad f(x^*, y^*) = 1 .$$

Swarm size	Number of iterations	α	δt	c_1	c_2	Maximum speed	Starting inertia weight	Inertia weight decrease rate	Minimum inertia weight
20	1000	1	1	2	2	20	1.4	0.99	0.3

Table 1: PSO parameters used in `PSO22a`.

2.2(b)

The PSO was then modified to handle integer programming (see script `PSO22b` in folder `2.2/2.2b`). When the objective function is evaluated, particle positions are temporarily rounded to the nearest integer values, whereas floating point values are using in all other stages of the computation.

The script was then used to find the minimum of the function

$$f(\boldsymbol{x}) = -(15\ 27\ 36\ 18\ 12)\ \boldsymbol{x} + \boldsymbol{x}^T \begin{pmatrix} 35 & -20 & -10 & 32 & -10 \\ -20 & 40 & -6 & -31 & 32 \\ -10 & -6 & 11 & -6 & -10 \\ 32 & -31 & -6 & 38 & -20 \\ -10 & 32 & -10 & -20 & 31 \end{pmatrix} \boldsymbol{x}\ ,$$

where $\boldsymbol{x} = (x_1, x_2, x_3, x_4, x_5)^T$, and $x_i \in \{-30, -29, \ldots, 29, 30\} \in \mathbf{Z}$.

Using parameters in Table 2, two (possibly global) minima were found:

$$\boldsymbol{x}^*_{(1)} = (0, 11, 22, 16, 6)^T\ , \quad \text{and} \quad \boldsymbol{x}^*_{(2)} = (0, 12, 23, 17, 6)^T\ , \qquad \text{both with } f(\boldsymbol{x}^*_{(i)}) = -737\ .$$

Swarm size	Number of iterations	α	δt	c_1	c_2	Maximum speed	Starting inertia weight	Inertia weight decrease rate	Minimum inertia weight
50	2000	1	1	2	2	60	1.4	0.99	0.3

Table 2: PSO parameters used in `PSO22b` (integer programming).

Problem 2.3 - Optimisation of braking systems

An intelligent braking system for trucks descending slopes was developed by evolving populations of multilayer FFNN with three inputs (normalised velocity $v/v_{\max}$, slope $\alpha/\alpha_{\max}$ and foundation brakes temperature $T/T_{\max}$) and two outputs (brake pedal pressure P_p and requested gear change Δ_{gear}).

GA parameters and other parameters of the truck basic model can be specified at the beginning of the main script `BrakingSystemOptimisation.m` (in folder `2.3/GA_Training`). The script generates a population of chromosomes representing (3-N_h-2)-perceptrons, where the number N_h of hidden units can be modified by the user (hard-coded default value $N_h = 8$), with genes sampled uniformly at random in [0,1] (real number encoding). Specifically, chromosomes are (2×1) MATLAB cells containing two weight matrices (input-to-hidden and hidden-to-output), and each such matrix also includes neuron biases in an extra column.

At the evaluation step, each chromosome is decoded into the corresponding FFNN with rescaled weights and biases (default range [-10, 10]) and such network is then used as a braking control system, running a series of simulations on different slopes. Network outputs at each time-step are used to control the truck pedal pressure and gear. The gear, in particular, is increased by one if $\Delta_{\text{gear}} > 0.7$, decreased by one if $\Delta_{\text{gear}} < 0.3$ and otherwise left unchanged. Moreover, the gear value is only modified if no gear changes have occurred in the last 2 seconds.

At the end of each simulation, the network fitness on the slope i is calculated as

$$F_i = \sqrt{\bar{v}_i \; d_i} \; ,$$

where $\bar{v}_i$ is the average speed over the simulation (before termination), and d_i is the travelled distance. Each simulation is stopped if any of the following constraints is violated:

- $T > T_{\max}$ (maximum brakes temperature exceeded);
- $v > v_{\max}$ (maximum speed exceeded);
- $v < 0$ (truck stopped);
- $d > d_{\max}$ (slope completed, where by default $d_{\max} = 1000$ m).

Simulations are performed on a set of 10 training slopes and on a set of 5 validation slopes[1]. At the end of each simulation set, the FFNN fitness is taken as the mean of the fitness values F_i. Only the fitness value computed on the training set is used as feedback for the GA, whereas the validation fitness is only used for determining when to stop the training process. The reference function for the evaluation step is `EvaluateNetwork.m`, which basically integrates the simple equation of motion of the truck described in the task. Model parameters are harcoded in the function, and were set to the suggested values, using SI units. The default time-step length Δt, instead, is 0.1 s.

At the selection step, tournament selection is used (default parameters: size = 3 and $p_{\text{tour}} = 0.8$). Crossover operator is not used, as neural networks perform distributed computation and cross-breeding two good networks does not necessarily produce a good offspring network (in fact, it usually does not).

Mutation operator works as follows: if a gene has to be mutated, i.e. a drawn random number is lesser than p_{mut} (default value $1/n_{\text{genes}} = 1/(6N_h + 2) = 0.02$), then another random number is drawn. If this number is lesser than p_{creep} (default value = 0.3), then a uniform creep mutation with rate C_r (default value = 0.2) is applied to the gene, making sure it does not fall below 0 or rise above 1. Otherwise, the mutation is full-range, and the gene is set to a random value in [0,1].

At each generation, the network with the highest training fitness is copied unchanged in the next population (more than one copy can be replicated by modifying parameter `elitismCopies`).

Several training runs were carried out, with various parameters sets. A typical run is shown in Figure 4. Very long simulations were not possible due to a lack of time, therefore the overfitting phenomenon was not observed. Thus, in order to select the best network, in the final generation the networks with the highest fitness on the training and validation sets were compared by testing them on the (previously unseen) test set. The network with the highest fitness on the test set was taken as the best network *of each run*. A set of 5 best networks were obtained with this procedure. The *overall best* network was then chosen by comparing the fitness values of these 5 networks on the test set. The best o these 5 FFNNs was able to complete all 5 slopes in the test set without violating any temperature or speed constraint.

The *best overall* network is hard-coded in the script `TestProgram.m`, in folder `2.3/Test_Program`. This script allows to rerun the network on an arbitrary slope, which must be specified in the beginning of the script as an anonymous MATLAB function, as in

```
alpha = @(x) 4 + sin(x/50) - 2*cos(sqrt(3)*x/50);
```

[1]Training, validation and test slopes are generated using the provided slopes as templates and are hardcoded in function `GetSlopeAngle.m`

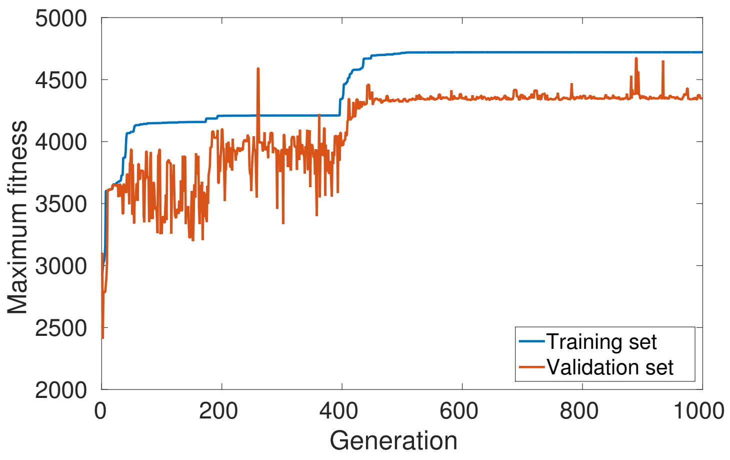


Figure 4: Maximum fitness in the population in a typical training run. Training set in blue, validation set in red.

The (horizontal) length of the slope can also be specified at the beginning of the script, by modifying variable `slopeLength` (default value = 1000). Other parameters can also be modified, but this can affect the performance of the network, as the training process was carried out with the hard-coded default parameters. At the end of the run, the script plots some graphical outputs showing the FFNN performance as an intelligent braking system. Figure 5 shows the performance of the best network found tested on the slope above (taken from the test set) extended to 10000 m. As we can see, the truck is able to complete the (rather lengthy) run without brake overheating or exceeding the maximum speed.

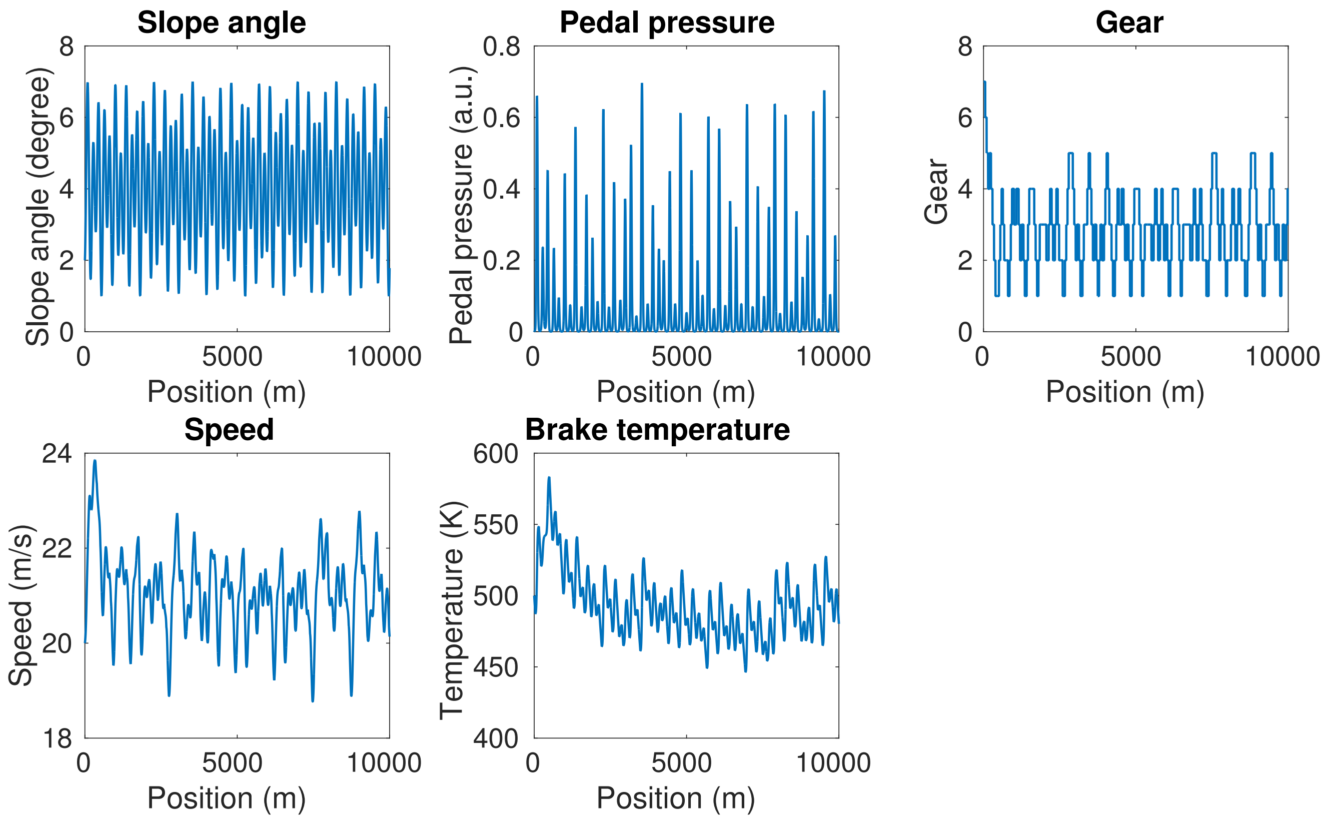


Figure 5: Performance of the FFNN hard-coded in script `TestProgram.m` on a slope picked from the test set.

Problem 4.4 - Function fitting using LGP

A LGP program was implemented in MATLAB, to evolve functions fitting a given data set (see script `LGP24.m` in folder `2.4/LGP`).

First, a population of chromosomes representing sets of instructions is initialised. Each chromosome is a $[n \times 4]$-matrix, with n being the number of instructions encoded in the chromosome. Each matrix entry is an integer value picked uniformly at random in its appropriate range, i.e. [1, 4] for the operator in column 1, [1, M] for the destination in column 2, [1, $M + N$] for the operands in column 3 and 4, where M is the number of variable registers and N is the number of constant registers. Only basic operators $\{+, -, \times, /\}$ are used in the program, and division is protected with constant $c_{\max}$.

Given a data set of K values (x, y), the program places a value x_k in the first variable register r_1, whereas all other variable registers are set to 0, then the instructions encoded in the chromosome are evaluated and the final value of r_1 is used as the output, i.e. the function estimate $\hat{y}_k$ of y_k. This is repeated for each $k = 1, \ldots, K$. The chromosome fitness is then taken as the inverse of the error defined as

$$e = \sqrt{\frac{1}{K} \sum_{k=1}^{K} (\hat{y}_k - y_k)^2} \; .$$

The algorithm then proceeds with tournament selection, two-point crossover and mutation. Given two selected chromosomes, two-point crossover is carried out by cutting each of chromosome into 3 sub-matrices and then swapping the middle sub-matrices. Mutation, instead, is carried out by checking each gene for mutation and, in case of success, by drawing an integer value uniformly at random in the corresponding range of the gene (see above-mentioned ranges).

The program was used to find the curve which best fits the provided data set. Several runs were carried out with different parameters, in order to find a suitable parameter set for the problem. Finally, a few long runs were performed with the parameters in Table 3, finding a function yielding an error of 0.0169. The best (though overly complicated) fitting function found is

$$f(x) = \frac{-6\,(-x^{17} + 7x^{16} - 44x^{15} + 184x^{14} - 668x^{13} + 1936x^{12} +}{6x^{18} - 37x^{17} + 231x^{16} - 900x^{15} + 3192x^{14} - 8748x^{13} + 21696x^{12} +} \cdots$$

$$\cdots \frac{-4910x^{11} + 10369x^{10} - 19175x^{9} + 29414x^{8} - 38832x^{7} + 40044x^{6} +}{-43830x^{11} + 81045x^{10} - 123363x^{9} + 176566x^{8} - 206592x^{7} + 244500x^{6} +} \cdots$$

$$\cdots \frac{-32814x^{5} + 13302x^{4} + 3294x^{3} - 18387x^{2} + 14094x - 10935)}{-228150x^{5} + 250542x^{4} - 182682x^{3} + 188325x^{2} - 79218x + 64881} \; .$$

This function is hard-coded in script `TestFit.m` (in folder `2.4/Test_Fit`), which can be run standalone to produce the plot in Figure 6, showing the original data together with the best fitting curve found with LGP. The program also computes the fitting error and outputs it on the console.

Population size	100
Number of generations	5000
Tournament size	5
p_{tour}	0.75
p_{cross}	0.2
p_{mut}	0.01
Number of variable registers M	3
Number of constant registers N	3
Constant registers values	(1, 3, -1)
Initial number of instructions	$5 \sim 25$
$c_{\max}$	10^{10}

Table 3: Parameters used in LGP long runs.

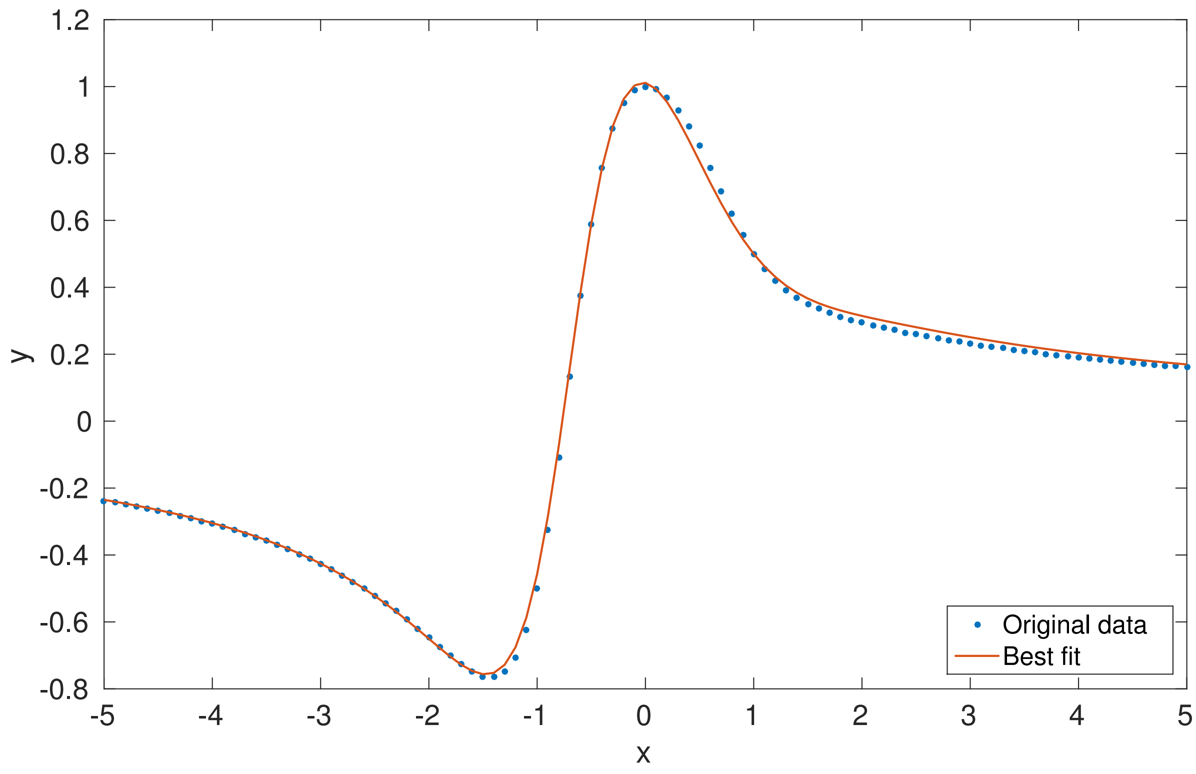


Figure 6: Graphical output of program `TestFit.m`. Blue dots: original data points. Red curve: best fitting function found with program `LGP24.m`