Stochastic optimization algorithms 2024
Home problems, set 2

General instructions. READ CAREFULLY!

Problem set 2 consists of four parts. Problems 2.1 and 2.2 are mandatory, the others voluntary (but check the requirements for the various grades on the web page). After solving the problems, collect your programs *and* your report (see below) in *one* zip file which, when extracted, makes the report available and also generates one main folder for each problem in the assignment (with subfolders if necessary).

You should provide a *single* report in the form of a PDF file (Note: *Only* this format is accepted). In the case of *analytical* problems, make sure to include *all the relevant steps of the calculation* in your report, so that the calculations can be followed. Providing only the answer is *not* sufficient. For analytical problems, whenever possible use symbolic calculations as far as possible, and introduce numerical values only when needed. You should write the report on a computer, preferably using LaTeX (see `www.miktex.org`). Scanned, handwritten pages are *not* allowed.

In *all* problems requiring programming, use Matlab. The *complete* Matlab program for the problem in question (i.e. all source files) must be handed in, collected in the folder(s) for the problem in question. Do not include unnecessary files, e.g. unused Matlab files. It should *not* be necessary to edit the programs, move files etc. Furthermore, when writing Matlab programs, you should make sure to follow the coding standard (available on the web page). You may hardcode *parameters* in your .m-files (as is indeed done in some of the skeleton files, e.g. `RunAntSystem.m`). Programs that do not function, or require editing to function, or deviate significantly from the coding standard, or otherwise do not follow the requirements given above, will result in a deduction of points.

The maximum number of points for problem set 2 is 15. Incorrect problems will be returned for correction *only* in cases where the mandatory requirements have not been met, so please make sure to check your solutions and programs carefully before submitting the (single) compressed file (.zip or .7z) in Canvas.

You may, of course, discuss the problems with other students. However, each student *must* hand in his or her *own* solution. Note that a plagiarism check will be carried out, in which both your report and your code are checked against reports and code from other students. In obvious cases of plagiarism, points will be deducted from all students involved.

NOTE: Don't forget to write your name *and* civic registration number on the front page of the report! Make sure to keep copies of the files that you hand in! Good luck!

(Strict) deadline: 20241016, 23.59.59

Problem 2.1, 3p, The traveling salesman problem (TSP) (mandatory)

The traveling salesman problem (TSP) has many applications in, for example, network routing and placement of components on circuit boards. In this problem, you will solve the TSP in a case where the cost function is simply taken as the length of the path.

Following the description in Chapter 4, implement an Ant system (AS) algorithm (Algorithm 4.1), for solving the problem of finding the shortest possible path between the cities whose coordinates are given in the file `LoadCityLocations.m` (available on the web page). This file contains a 50×2 matrix with the coordinates (x_i, y_i) for city i, $i = 1, \ldots, 50$. You should use the sample code (for plotting paths) available on the web page in the file `TSPGraphics.zip`. Whenever a new best path is found (and *only* when that happens), it should be immediately visualized in the plot window, to keep the user informed about the progress of the program. Thus, it is *not* sufficient just to plot the final path at the end of a run.

Any generated path should, of course, fulfil the criteria for a valid TSP path: Every city (node) should be visited once during the tour which should then end with a return to the starting city (i.e. as described in the course book).

For the implementation, you *should* use the Matlab script `RunAntSystem.m` (available on the web page) as the starting point, and you should write the functions that are mentioned in that file (see the comments that include the words "To do"). The inputs to the functions *should* be *exactly* as specified in `RunAntSystem.m`. Note that, except for changing parameter settings and removing comments (so that the various functions are actually called), you should *not* modify `RunAntSystem.m`. (Exception: You may add code for storing the best paths found during a run, if you wish). The most complex function to write is the `GeneratePath` function that, taking τ_{ij}, η_{ij}, α and β as inputs, should return the path (i.e. a list of node indices) for *one* ant (i.e. *not* all the paths at once). It is a good idea (for clarity) to use at least one additional function called by `GeneratePath`, for example `GetNode`, that returns the next node in the path, given the tabu list, the pheromone levels (τ_{ij}), the visibility matrix (η_{ij}), α, and β as inputs. Note also that the `GetVisibility` function should return the matrix η_{ij}, *not* η_{ij}^{β}. Similarly, the `pheromoneLevel` matrix should, of course, store τ_{ij}, *not* τ_{ij}^{α}. When updating pheromone levels, follow the equations in Algorithm 4.1 exactly.

Notes:
(1) The pheromone levels will decay quite fast on those edges that are rarely traversed. Thus, to avoid numerical problems that might occur as a result of that decay, you may include a lower bound on the pheromone levels: If $\tau_{ij} < 10^{-15}$ (for any ij) then set $\tau_{ij} = 10^{-15}$ for that edge.
(2) Do *not* symmetrize τ_{ij}. The AS algorithm says nothing about symmetry of τ_{ij}. In this particular case it is true at $\eta_{ij} = \eta_{ji}$ but that is not the case for all problems. Your algorithm should be applicable to the general case.

Once you have completed the implementation of the AS algorithm, run the program to find a path that is as short as possible. You should store (only) the best path found, in a file named `BestResultFound.m`, consisting of a single line of the form

```
bestPath = [4 7 1 39 50 41 3 ...  etc.
```

You can generate this file whichever way you like (it need not be *saved* by your program, you may generate the file by hand, if you prefer, using the contents of the workspace when the run has been completed), but make sure that the saved vector is directly readable using Matlab. In particular, `BestResultFound.m` should not contain any instances of the `>` character. Note also that the indices *should* be in the interval $[1, 50]$, *not* $[0, 49]$, and that the path should contain exactly 50 elements (the return to the start city is implied, but the start city should *not* be appended at the end of the path). The length of the path will be tested using the vector that you provide, assuming that city indices have been enumerated from 1 to 50. For full points on the problem, the length of the best path (obtained with your implementation of the algorithm) should be shorter than 100 length units.

What to hand in You should hand in your complete Matlab program (all files, including `RunAntSystem.m`), as well as the file `BestResultFound.m` containing the `bestPath` vector, with 50 elements. In your report include a section *Problem2.1* where you specify the length of your shortest path and also provide a plot of the best path (i.e. the graphical output of the program).

Maximum number of points for this problem: 3p.
Maximum number of points if the problem must be returned for correction: 1p

Problem 2.2, 3p, Particle swarm optimization (mandatory)

In this problem, you will implement and use particle swarm optimization (PSO), which is a stochastic optimization method based on the properties of swarms such as, for example, bird flocks.

Start by implementing a standard PSO algorithm (as described in Chapter 5) in Matlab in a Matlab script called `RunPSO.m`. Note: The standard PSO algorithm *should* include the (varying) inertia weight, see p. 128 and Eq. (5.20) in the course book! Remember to follow the coding standard and to place separate Matlab functions in separate files. Next, use the `contour` command in Matlab to determine the number of minima of the function

$$f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2, \tag{1}$$

over the range $(x_1, x_2) \in [-5, 5]$. Hint: The function varies quite strongly with x_1 and x_2. In order to make sure that you identify all the local minima from the contour plot, you may therefore instead plot the function $\log(a + f(x_1, x_2))$, where a is a small positive constant (e.g. 0.01).

Next, use your PSO (`RunPSO.m`) to find the location of *all* the local minima (as well as the corresponding function values) of the function $f(x_1, x_2)$. Since the PSO is stochastic, it will generally find different minima in different runs. Thus, you will need to run the PSO a number of times in order to find all the minima.

What to hand in You should hand in your complete Matlab program (all files). In your report, include a section *Problem2.2* where you should provide the contour plot described above (with the minima clearly identified in the plot) as well as a table with (x_1, x_2, f) for the minima that you have found with your PSO.

Maximum number of points for this problem: 3p.
Maximum number of points if the problem must be returned for correction: 1p

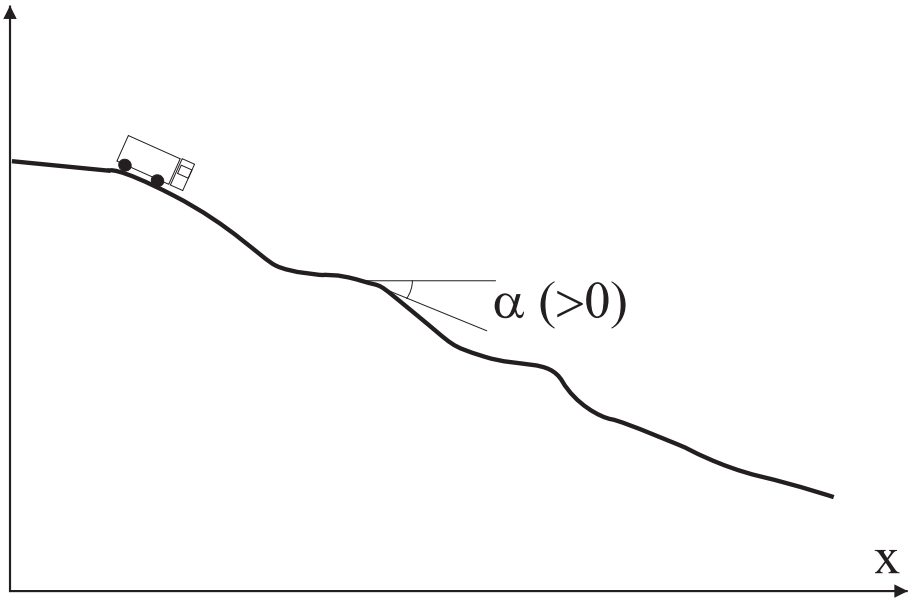

Figure 1: *A schematic illustration of a slope. The variable x measures the horizontal distance from the origin, and $\alpha(x)$ is the instantaneous slope angle at x. Note that α is always positive for a downhill slope.*

Problem 2.3, 5p, Optimization of braking systems (voluntary)

Here, you will develop an intelligent system for braking heavy-duty trucks during complicated descents (see pp. 83-86 in the course book). Such trucks are equipped with several braking systems, including the ordinary pedal brakes (also called foundation brakes), engine brakes, retarder etc. In order for the (foundation) brakes to work properly, the brake discs must never be overheated. We will use a strongly simplified truck model, which will now be described.

Truck model

The truck is described by the following equations. The acceleration ($\dot{v}$) is given by

$$M\dot{v} = F_{\mathrm{g}} - F_{\mathrm{b}} - F_{\mathrm{eb}}, \tag{2}$$

where M is the mass of the truck and F_{g} is the component of the force of gravity in the instantaneous direction of motion, i.e. $Mg\sin\alpha$, where $\alpha(>0)$ is the instantaneous downhill slope angle (see Fig. 1). F_{b} is the force from the foundation brakes, and F_{eb} the force from the engine brakes. The braking force F_{b} is given by

$$F_{\mathrm{b}} = \begin{cases} \frac{Mg}{20} P_{\mathrm{p}} & \text{if } T_{\mathrm{b}} < T_{\max} - 100, \\ \frac{Mg}{20} P_{\mathrm{p}} \mathrm{e}^{-(T_{\mathrm{b}} - (T_{\max} - 100))/100} & \text{otherwise} \end{cases}, \tag{3}$$

where g is the constant of gravity (use SI units!). $P_{\mathrm{p}} \in [0, 1]$ denotes the pressure (applied by the driver) on the brake pedals. The brake temperature ΔT_{b} (relative to the ambient temperature) is assumed to vary according to

$$\frac{\mathrm{d}\Delta T_{\mathrm{b}}}{\mathrm{d}t} = \begin{cases} -\frac{\Delta T_{\mathrm{b}}}{\tau} & \text{if } P_{\mathrm{p}} < 0.01, \\ C_{\mathrm{h}} P_{\mathrm{p}} & \text{otherwise} \end{cases}, \tag{4}$$

where τ and C_h are constants. Note that the temperature in the brakes never can fall below the ambient temperature T_amb. Thus, the *actual* brake temperature equals $T_b = T_\mathrm{amb} + \Delta T_\mathrm{b}$.

Furthermore, it is assumed that the truck has 10 gears, and that the force from the engine brakes F_eb is given by

$$F_\mathrm{eb} = \begin{cases} 7.0C_\mathrm{b} & \text{gear 1} \\ 5.0C_\mathrm{b} & \text{gear 2} \\ 4.0C_\mathrm{b} & \text{gear 3} \\ 3.0C_\mathrm{b} & \text{gear 4} \\ 2.5C_\mathrm{b} & \text{gear 5} \\ 2.0C_\mathrm{b} & \text{gear 6} \\ 1.6C_\mathrm{b} & \text{gear 7} \\ 1.4C_\mathrm{b} & \text{gear 8} \\ 1.2C_\mathrm{b} & \text{gear 9} \\ C_\mathrm{b} & \text{gear 10} \end{cases}, \tag{5}$$

where C_b is a constant.

Using the model just described, we will now implement a GA to optimize (the weights of) a feedforward neural network (FFNN) (with a single hidden layer, with N_h hidden neurons) for handling the braking system. There are four main steps of the implementation: (1) writing the decoding and encoding steps for converting a chromosome into an FFNN, and vice versa; (2) implementing the truck model; (3) implementing the GA for optimizing the FFNNs; and (4) writing a test program for re-evaluating optimized FFNNs. Now follow the steps below to complete the implementation.

Step 1: Encoding and decoding Write program code for encoding an FFNN into a chromosome and for decoding the chromosome (to get the same FFNN back). Here, you *must* use the template files `EncodeNetwork.m` and `DecodeChromosome.m` (available on the Canvas web page). You may *not* change the interfaces (inputs and outputs) in those methods. Note that an FFNN with a single hidden layer is defined by two matrices, $w^{I\to H}$ and $w^{H\to O}$. $w^{I\to H}$ encodes the weights between the n_i inputs and the hidden layer with n_h neurons. This matrix has n_h rows and $n_\mathrm{i}+1$ columns (where the +1 comes from the bias term). $w^{H\to O}$ encodes the weights between hidden layer and the output layer (with n_o neurons). This matrix has n_o rows and $n_\mathrm{h}+1$ columns (where the +1 comes from the bias term).

Then, use the program `RunEncodingDecodingTest.m` (also available in Canvas) to make sure that your encoding function and your decoding function work as they should, i.e. such that the decoding is the inverse of the encoding, meaning that the original matrices are obtained after carrying out both steps. Your functions should be able to handle any positive, integer values of n_i, n_h, and n_o.

Note: Having a correct decoding function is essential for solving this problem. Thus, when correcting, we will first check the encoding-decoding test (using your implementations of those two functions) over a set of different matrices, corresponding to different values of n_i, n_h, and n_o. *If there are errors in that part, the remaining parts of the solution will not be considered.*

Step 2: Truck model Write a program that implements the truck model described above. The program should be able to determine $v(t)$ by integrating Eq. (2) for any downhill slope (specified by $\alpha(x)$), see the file `GetSlopeAngle.m` which is available on the Canvas web page. Discretize the differential equations using first-order differences, e.g.

$$\dot{v} \approx \frac{v(t+\Delta t) - v(t)}{\Delta t}, \tag{6}$$

where Δt is the time step length, which should be smaller than the smallest relevant time scale in the problem. In this case Δt should not exceed 0.25 s, and even smaller values may be better. You can test your program by setting P_p to a constant value, for example, or by setting it to 1 when the speed exceeds a certain value (you do not need to hand in these tests).

Step 3: Optimization program Implement a GA (`RunFFNNOptimization.m`) where the individuals consist of FFNNs (decoded from the chromosomes as described above), and the evaluation consists of running a truck (using the model described above) over a set of slopes (for a specification of the fitness measure; see below). You should implement the the FFNNs (and their evaluation) yourself, i.e. you should not use any pre-existing neural network packages. For the neurons, use the logistic sigmoid activation function (Eq. (A6) in the course book) with a constant parameter c (chosen, once and for all, somewhere in the range [1,3]).

The FFNN should have three inputs, namely $v/v_{\max}$, $\alpha/\alpha_{\max}$ (see below), and $T_b/T_{\max}$, where $v_{\max}$, $\alpha_{\max}$, and $T_{\max}$ are constants, and two outputs, namely P_p and Δ_{gear}. Δ_{gear} denotes the gear choice, i.e. whether to increase or decrease the gear (by a single step, in both cases), or to leave the gear unchanged. The Δ_{gear} output signal can be encoded in various ways - you may select any suitable encoding. Note that Δ_{gear} is the *desired* gear change. However, it is not possible to change gears several times per second: The number of gear changes should be limited to *one* every two seconds (i.e. after changing gears, no further change can be made until at least two seconds later, regardless of the output from the neural network. Thus, you must take this into account when evaluating an FFNN.

The optimization criterion is that the truck should manage to drive as fast as possible down a given slope, *without* ever violating the constraints regarding speed (should not exceed $v_{\max}$ or fall below $v_{\min}$) and brake temperature (should not exceed $T_{\max}$). If any constraint is violated, the evaluation (on that particular slope) should be terminated. A possible fitness measure for slope i is therefore

$$F_i = \overline{v}_i d_i, \tag{7}$$

where $\overline{v}_i$ is the average speed over the evaluation (before termination), and d_i is the distance travelled before termination. Thus, if the truck manages to cover the whole slope, then $F_i = \overline{v}_i L$, where L is the length of the slope (see below). Note that other fitness measures are possible (and might work better). You may choose a suitable fitness measure yourself. Your program should measure the fitness over both the training set and the validation set (specified below). The fitness value F for a given network over a given set (e.g. training) can be taken as either the average $\overline{F}$ over the slopes in the set or as the *worst* evaluation, i.e. $F = \min F_i$.

In order to train your network, generate three sets of slopes: A training set (10 slopes) a validation set (5 slopes) and a test set (5 slopes) and store them in the file `GetSlopeAngle.m`,

using the format and interface specified in there. In other words, edit this file to generate your slopes. Make all slopes $L = 1000$ m long (horizontal distance; see Fig. 1). α (> 0) should vary with the horizontal distance x (i.e. it should *not* be constant), and should never exceed $\alpha_{\max}$, see below. Note that the function `GetSlopeAngle.m` should give the slope angle in *degrees*.

Step 4: Test program As the final implemenation step, you should write a program called `RunTest.m` that allows a user quickly to rerun your best network (i.e. the one with smallest validation error) on an arbitrary slope. This program should automatically load the chromosome corresponding to the best network, decode the chromosome, and run the network (once) on a given slope from the `GetSlopeAngle.m` file (e.g. the first slope in the test set), without the user having to specify any inputs or otherwise modify the code. Moreover, when the truck *reaches the end* of the slope (or violates the constraints so that the simulation is stopped) this test program *should* generate a set of (sub-)plots (in one plot window) showing (i) the slope angle (α), (ii) the brake pedal pressure, (iii) the gear, (iv) the speed, and (v) the brake temperature, as functions of the horizontal distance travelled (x). The plots should be shown once the truck reaches the end, i.e. not during the run.

Running the program When you have completed the four steps of the implementation, run the GA to train the network, using the method of holdout validation, described in Appendix C.2 in the book, i.e. measure both F^{tr} and F^{val}, use F^{tr} as feedback to the GA and F^{val} to determine when to stop the training. Make sure to store the best chromosome found, in a file called `BestChromosome.m`, for use in the `RunTest.m` program. Set the parameters according to $T_{\max} = 750$ (K), $M = 20000$ (kg), $\tau = 30$ (s), $C_{\text{h}} = 40$ (K/s), $T_{\text{amb}} = 283$ (K), $C_{\text{b}} = 3000$ (N), $v_{\max} = 25$ (m/s), $v_{\min} = 1$ (m/s), $\alpha_{\max} = 10$ (degrees). Assume that the truck starts at $x = 0$, with speed 20 m/s and with gear 7 in place. In the starting position, set $T_{\text{b}} = 500$ K. You may chose any suitable value (around 3-10, say) for the number of hidden neurons (n_{h}).

What to hand in You should hand in the complete Matlab programs, namely your implementations of `EncodeNetwork.m` and `DecodeChromosome.m`, the complete GA program, `RunFFNNOptimization.m` with all relevant Matlab functions (including your version of `GetSlopeAngle.m`) placed in separate files, as usual, so that one can directly rerun the optimization), and the `RunTest.m` program. You should also hand in the best chromosome found (i.e. the file `BestChromosome.m`). In your report, include a section *Problem2.3* where you describe your implementation (including your choice of fitness measure). You should also plot the results from your best GA run, i.e. the maximum fitness value in the population, as a function of the number of evaluated generations, obtained on the training and validation sets.

Your network will be tested on a few test slopes generated as described above, and your score will be based on the performance of the system on these test slopes. It is therefore important to make sure that your network is capable of handling previously unseen test slopes (within reason: The test slopes will have similar slope variation as in the `GetSlopeAngle.m` file on the Canvas page). The implementation of the programs is worth 3p, and the remaining 2p are awarded depending on the performance over our test slopes.

Population 包含 N个个体（这里一个个体就是一个染色体 因为 g只有 一个变量 x） = N个拟合函数
Chromosome = 拟合函数，BestChromosome 就是一个候选的 g(x).

Problem 2.4, 4p, Function fitting using LGP (voluntary)

Consider a case in which a data series has been generated from a function of the form

$$g(x) = \frac{a_0 + a_1 x + a_2 x^2 + \ldots + a_p x^p}{b_0 + b_1 x + b_2 x^2 + \ldots + b_q x^q}. \tag{8}$$

The values of p and q, as well as the constants a_i and b_i are unknown, and should be inferred from the data series using LGP. Start by writing a general LGP program (called `RunLGP.m`), with M variable registers, N constant registers, and the operator set $\{+, -, \times, /\}$. The program should evolve linear chromosomes using tournament selection, two-point crossover (see pp. 76-77 in the book), and mutations where, as usual, the various functions should be placed in separate Matlab files. Make sure that you store the best chromosome found (in a file called `BestChromosome.m`), which you will need later; see below.

The structure of the chromosomes should be of the kind illustrated in Fig. 3.19 in the book, i.e. such that each instruction is defined by four genes. The first gene (in a given instruction) should encode the operator and the second gene should encode the destination register. The third and fourth genes should encode the two operands. Note that crossover should occur between instructions, as shown in Fig. 3.21 in the book. The data set, consisting of values of the function $y = g(x)$ for various values of x, is contained in the file `LoadFunctionData.m`, available on the course web page.

The evaluation of a chromosome should be done as follows: For each data point (x_k, y_k), place the value of x_k in the first variable register (r_1), and set the contents of the other variable registers to 0. Next, execute the sequence of instructions contained in the chromosome, and take the final value contained in r_1 as the output, i.e. as the estimate $\hat{y}_k$ of y_k. When all data points have been considered in this way, form the total error as

$$e = \sqrt{\frac{1}{K} \sum_{k=1}^{K} (\hat{y}_k - y_k)^2}, \tag{9}$$

where K is the number of data points. Finally, set the fitness value as $f = 1/e$. Note that the data are (unrealistically) noise-free. Thus, there is no risk of overfitting in this problem. You may use as many variable registers and constant registers as you like (but you only need a few of each). Note that the values of the constant registers should be set once and for all, before the evaluation of chromosomes begins.

Run your program, and try to determine the function $g(x)$. In addition, you *should* write a separate test program (`TestLGPChromosome.m`) that loads the best LGP chromosome (from `BestChromosome.m`), then decodes and evaluates this chromosome, and then, finally, plots both the original data (y_k, $k = 1, \ldots, K$) and your best fit (i.e. the output $\hat{y}_k$, $k = 1, \ldots, K$ obtained from running the best LGP chromosome). Morever, this program *should* print out both your best estimate of the function $g(x)$ (which must be computed from the chromosome when the test program runs, it cannot be hardcoded) and the error (see above) as text output. Thus, you must write a Matlab function (for use in the test program only) that takes a chromosome as

input and outputs your estimate of the function $g(x)$ in the form given above. It is *not* allowed to use the Symbolic Math Toolbox in your LGP program (`RunLGP.m`), but you may use it in order to simplify your best function in the test program (`TestLGPChromosome.m`). Note that the simplification can also be done via string operations.

Hint: In order to cope with variable-length chromosomes, you may wish to use the `struct` concept in Matlab, see e.g. the help files in Matlab (`>> help struct` etc.) and the simple example available on the web page.

If you wish, you may also use some of the more advanced EA operators in this problem, such as, for instance, varying mutation rates (to implement that, you can simply increase the mutation rate if there is no progress for a few generations, and then reduce it again when a new best solution is found. There are other ways as well.)

In this problem, the chromosomes may grow to be very long (at least temporarily, until the algorithm begins zooming in on the correct solution). To avoid having to deal with (and decode) very large chromosomes, you should set an upper limit on the chromosome length (m) at $m = 400$, i.e., 100 instructions (since each instruction requires four genes). In crossover, if the length of either (or both) of the generated LGP chromosomes exceeds this limit, then discard those two chromosomes and repeat the selection and crossover procedures. In other words, for any given generation, keep running selection and crossover until you have generated N chromosomes (N = the population size) such that no chromosome is longer than the length limit. Then apply mutation as usual, etc. In addition to the hard limit ($m = 400$), you can, if you want, introduce a soft limit, for example by applying a multiplicative penalty (reducing the fitness values) for individuals with, say, $m > 300$ or so.

What to hand in You should hand in the complete Matlab programs, i.e. `RunLGP.m` (with all necessary matlab functions, placed in separate files) *and* the `TestLGPChromosome.m` program. You should also hand in the best chromosome, in the file `BestChromosome.m`. *Solutions that do not include the test program and the best chromosome, will not be considered.* In the report, include a section *Problem2.4*, where you should describe your LGP program a bit (especially the use of any special operators to avoid premature convergence). You should also specify how many registers (variable and constant) were used as well as the the values chosen for the constant registers. Moreover, you should also provide (in the report) the output generated by your test program, namely (1) function that you have found (obtained from the best chromosome found by your LGP program), *in the same form as in Eq. (8) above* (but with numerical values for all constants); (2) the two plots showing the data points ($y_k,\ k = 1, \ldots, K$) and your best fit ($\hat{y}_k,\ k = 1, \ldots, K$); and (3) the error e, computed as described above.

The implementation of the programs is worth 2p, and the remaining 2p are awarded depending on the quality of the fit. For full points, the error (described above) should be less than 0.01 (Ideally, of course, you should provide the exact function!). If the error exceeds 0.30, no points are given for the performance of the program.