Lab 4:
Evolutionary Computation and Swarm Intelligence

Introduction

Goals

The goal of this assignment is to learn about two population methods: genetic algorithms, and particle swarm optimization. These are both methods, that work by using differently sized populations of solutions to explore large parts of the search space in parallel. Although they can be used to solve similar or identical problems, the approach of each is quite different. You will use both GA and PSO to solve several optimization problems that are challenging for gradient descent methods.

Preparations

Read through these instructions carefully in order to understand what is expected of you. Read the relevant sections in the course book.

Report

Hand in a report according to the instructions found on the course lab page. The report should contain answers to all questions, and all requested plots. Observe that some questions are marked with ★. These questions are of particular importance and should be answered *carefully.*

Files

You will need the files `gasolver.zip`, `psopt.zip` and `gtop.zip` found on the student portal.

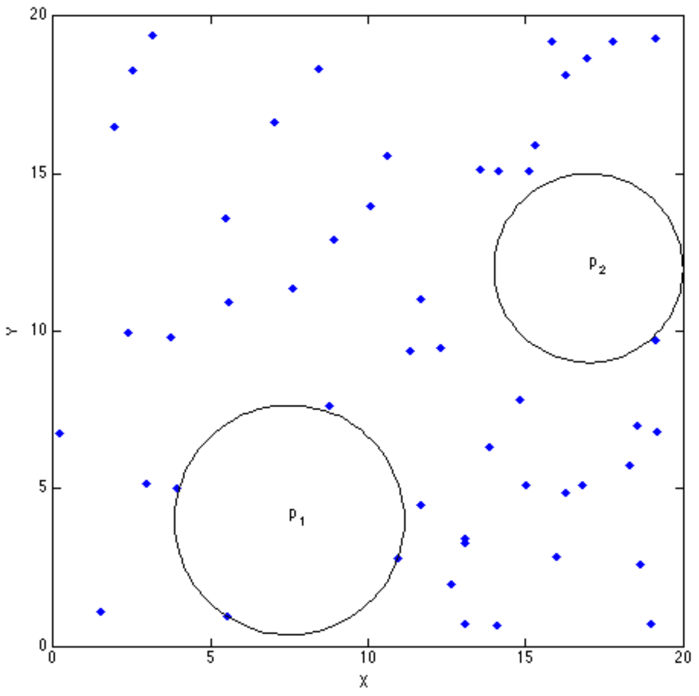

Figure 1: Two points, p_1 and p_2, with circles representing the distance to the nearest "star".

Task 1: Genetic Algorithms

For a random scattering of points, which we'll call "stars", in a bounded area, we wish to find the largest circle that can be drawn inside those same bounds that does not enclose any stars (see Figure 1). Solving this problem with a genetic algorithm requires deciding how to encode as a genome information sufficient to represent any solution. In this case, the only information we need is the center point of the circle, so our genome of point p_i will look like (x_i, y_i), representing the Cartesian coordinates.

Question 1: *Think about what each of the genetic operators means for this simplistic genome. Geometrically speaking, what would a 1-point crossover look like in this case? What about mutation?*

For the fitness of a genome, we'll use the radius of the largest circle that can be drawn at the coordinate it represents. Since the circle cannot contain any of the stars, that radius is the distance to the nearest star:

$$circle_{obj}(p_i) = \min_{s \in stars} \left\{ \sqrt{(x_i - x_s)^2 + (y_i - y_s)^2} \right\}$$

Question 2: *Would you expect more improvement in this problem to result from crossovers or mutations? Why? Is that what you would normally expect?*

Unpack the file `gasolver.zip` into a directory you can access. Locate the file `lab4.mat` in the MATLAB file browser, and double click it to add the contents of several variables to you workspace.

Now, you can set up the circle problem and run the GA solver like this:

```
star = GAparams
star.objParams.star = star1;
[best, fit, stat] = GAsolver(2, [0 20 ; 0 20], ...
    'circle', 50, 100, star);
```

GAsolver has three required parameters, and several optional parameters:

- The length of the chromosome (in this case, 2, for our two coordinates).
- An $n \times 2$ matrix of lower and upper bounds for each dimension. If there is 1 column, every position in the chromosome gets the same bounds; alternately, you may specify bounds for each position independently.
- A MATLAB function to calculate the fitness of the population. The function name is formed by prepending `ga_fit_` to the string you pass here; so, in this case, we are using the function `ga_fit_circle`, which you can find in your workspace. This function will be passed matrices holding the population (one genome per row) and the bounds of the variables, and an object of other parameters.
- The size of the population (default is 500).
- The number of generations (default is 1000).
- A structure of parameters for the solver. The `GAparams` struct has default values you can start with, and Table 1 gives brief explanations.

The function returns three variables:

- The chromosomes of the best individuals found. Each time a new global best is located, it is added to the top of this matrix, so `best(1,:)` is always the best individual found.
- The fitness values associated with the best individuals (`fit(1)` is the best fitness encountered).
- A structure of various statistics of the run. This is used to plot performance after the fact.

Table 1: Parameters for GAsolver. Default values shown in bold.

`param.`	values	
`geneType`	**‘float’**	
	‘binary’	$0, 1$
`crossover.func`	**‘1point’**	
	‘npoint’	set `crossover.n` for number of points
	‘uniform’	
	‘arithmetic’	(float) weighted average of n parents
	‘blend’	(float) stochastic weighted average of 2 parents
	‘geometric’	(float) negative values result in imaginary offspring!
	‘linear’	(float) 3 offspring on a line through parents, keep best 2 (requires extra fitness evals, but more exploration)
	‘none’	no crossover operator
`crossover.weight`	$[0-1, \dots]$	vector of parent weights (size sets # of parents)
`crossover.prob`	$0-1$	chance that given parents produce an offspring (default is .9)
`mutate.prob`	$0-1$	default is 0.005
`mutate.step`	$0-$	standard deviation of distribution for step size (used in some floating point mutation ops)
`select.func`	**‘proportional’**	selection based on actual fitness values
	‘rank’	selection based on ranking of fitness values
	‘tournament’	pick best from pool of n parents
	‘random’	ignore fitness, pick at random
`select.sampling`	**‘roulette’**	chance of selection proportional to fitness
	‘stochastic’	number of offspring proportional to fitness [**?**, section 8.5.3]
`select.size`	**2** $-$	(tournament) size of pool (default = 2)
`select.pressure`	$1-$ **2**	(rank) selection pressure (default = 2)
`scalingFunc`	**‘none’**	use raw fitness values
	‘sigma’	scale fitness based on variance (per generation)
`scalingCoeff`	$0-$ **1**	less selection pressure at high values
`replace.func`	**‘all’**	generational GA
	‘worst’	replace worst individuals of current gen
	‘random’	offspring replace random individuals
`replace.elitist`	**true**, false	fittest chromosome always survives
`visual.active`	Boolean	visualize the objective function
`visual.func`	string	name of the visualization function
`visual.step`	> 0	number of generations between updating the visualization
`verbose`	**true**, false	provide feedback in every generation
`stop.func`	**‘generation’**	stop at maximum generation count
	‘improvement’	stop when best fitness stops improving
	‘absolute’	stop when a (known) optimum is reached
`stop.window`	default: 100	generations to wait for fitness improvement
`stop.direction`	‘max’,‘min’	whether to maximize or minimize the objective
`stop.value`	default: 1	known best fitness value
`stop.error`	default: .005	how close to get to optimum before stopping

Run GAsolver now. You should get a list of the minimum, average and maximum fitness in each generation; once all generations are complete, you will also see the genome of the individual with the highest fitness.

To get a better idea of what the solver is doing, we'll supply it with a visualization function that will plot some information from intermediate generations. We'll do this by adding the name of the visualizer function to the parameter structure:

```
star.visual.active = 1;
star.visual.func = 'circle';
[best, fit, stat] = GAsolver(2, [0 20 ; 0 20], ...
    'circle', 50, 100, star);
```

Run GAsolver again. As the solver runs, the red circles represent the coordinates of the population of the current generation, whereas the bold red circle shows the objective value of the best individual. Run it several more times and observe how the spread of the population over the solution space changes as the population evolves. Odds are that it won't perform terribly well (a few of the default parameters are not particularly good for this problem). To this point, you've only been running for 100 generations, so one possibility is to give the solver more time to improve the solution; however, extra generations are unlikely to help in this case. To see why, we'll take a look at a plot of a measure of the diversity of the population over time.

```
ga_plot_diversity(stat);
```

Plot 1: *Include a plot showing the change in diversity of the population by generation.*

Question 3: *What are the possible sources of population diversity when using genetic algorithms? How are those sources of diversity reflected in the shape of the diversity curve in your plot?*

Question 4: *Why is population diversity important?*

To try and maintain diversity a little longer, we'll modify some of the parameters to the algorithm. You can get an overview of the current settings by typing:

```
ga_show_parameters(star);
```

One thing that can affect diversity is the choice of a selection heuristic. The default here, which you have been using up to now, is *proportional* selection, but, as you no doubt recall from lecture, *rank* based selection is often a better choice for avoiding premature convergence.

Question 5: *What's the difference between rank and proportional selection? Why does that make rank based selection better at avoiding premature convergence?*

In GAsolver, choices like selection heuristics, crossover and mutation operations, and replacement strategies are all implemented as MATLAB functions. To select a different function, you supply its name as a parameter, like this:

```
star.select.func = 'rank';
```

Rank selection also makes use of the selection pressure parameter, a number between 1 and 2 which affects the likelihood of more fit individuals being selected: at a value of 2, high fitness individuals are much more likely to be selected than low fitness individuals, while at 1 there is a much smaller difference in the probabilities.

Plot 2: *For each pressure value of 2, 1.75, 1.5, 1.25 and 1 provide one plot of the population diversity over 100 generations using rank based selection.*

Question 6: *What selection pressure resulted in the most promising looking diversity curve? Run the algorithm a few more times using that pressure setting. What was the best fitness value and position you were able to locate?*

The replacement strategy determines which individuals from the old population and the new pool of offspring survive into the next generation. The default here is a *generational*, meaning that the offspring completely replace the old population, and *elitist*, meaning that the best individual encountered so far is always preserved unchanged.

To observe the effect elitism has on the algorithm, make 5 runs using the current settings, and make a record of which generation the best individual was created in for each trial. Also record whether the best individual was made by crossover or mutation. You can also plot the fitness of the population at each generation like this:

```
ga_plot_fitness(stat);
```

Repeat this experiment for 5 more runs, after deactivating elitism:

```
star.replace.elitist = false;
```

Plot 3: *Include fitness plots for one elitist trial, and one non-elitist trial.*

Question 7: *What effect does an elitist strategy have on the progress of fitness values?*

Question 8: *Report the generation the best individual was born in, and whether it was created by crossover or mutation, for both the elitist and non-elitist trial. What differences, if any, do you observe between the two methods?*

Now try the solver on two other star maps. `star2` has a global optimum in an area of low density which also has a couple of strong local optima; in `star3` the global optimum is found in the middle of an otherwise high-density section. Pass each of these maps to the solver by setting `star.objParams.star` equal to the new map, and then running the solver again, using the settings that have resulted in the best performance so far.

Question 9: *What settings did you use? How well did the solver perform on the more difficult maps? Explain any difference in performance you observed.*

Task 2: Minimizing a Function

In this task you will use genetic algorithms to minimize Ackley's function, a widely used test function for global minimization. Generalized to n dimensions, the function is:

$$\begin{aligned} f(x) = -a \cdot e^{-b \cdot \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2}} - e^{\frac{1}{n} \cdot \sum_{i=1}^{n} \cos(cx_i)} + a + e^1 \\ \text{for} \quad -32.768 \leq x_i \leq 32.768, \quad i = 1, \ldots, n \end{aligned} \tag{1}$$

We will use the standard parameters $a = 20, b = 0.2, c = 2\pi$. Ackley's function has several local minima, and a global minimum of $f(x) = 0$, obtainable at $x_i = 0$ for $i = 1, \ldots, n$. Take a moment to familiarize yourself with the shape of the two-dimensional Ackley's function, using the supplied visualizer function:

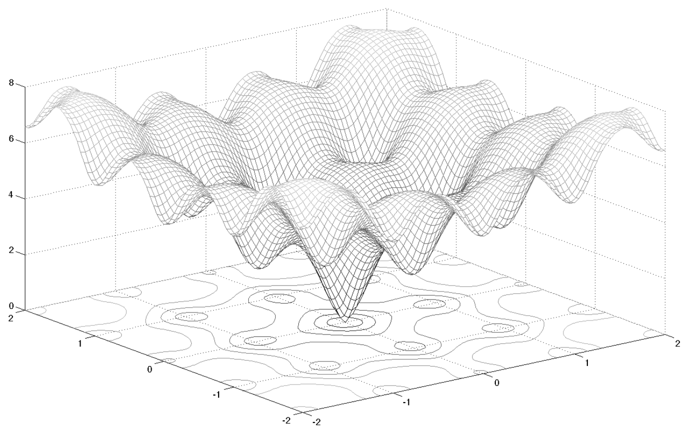
Figure 2: The area around the global minimum for Ackley's function in 2-D.

```
ack = GAparams;
ack.visual.type = 'mesh';
ga_visual_ackley([],[],[],[],[],[],ack.visual,[],[]);
```

This function is challenging partly because of the strong local minima that surround the global minimum. To get a better look, you can change the bounds of the surface plot as follows:

```
ack.visual.bounds = [-2, 2];
ack.visual.interval = 0.05;
ga_visual_ackley([],[],[],[],[],[],ack.visual,[],[]);
```

You can also change the plotting function to use many of the 3-D plots supplied by MATLAB , by substituting the name of the function (`'mesh'`) specified in `ack.visual.type`. Examples include '`contour`', '`surf`', '`surfc`', etc. Just remember that any adjustments you make to `ack.visual` will also affect the visualization you get when you run the solver.

Question 10: *Why would the global minimum be difficult to find using a gradient descent method?*

Set up the problem for solving the 20 dimensional Ackley's function:

```
ack.stop.direction = 'min';
ack.visual.func = 'ackley';
ack.visual.active = true;
[best, fit, stat] = GAsolver(20, [-32, 32], ...
   'ackley', 200, 250, ack);
```

This can be a difficult function to minimize, and the initial settings may not work well. You can try to modify the selection or replacement strategies, as we did in the last section, but this time we'll also try a few other approaches.

First, try some different crossover strategies. You can find some possibilities listed in Table 1. The 1-point, n-point, and uniform crossover operations operate by swapping the values of some genes between the parents. This is how binary encoded crossover usually works, but it's also valid for floating point encodings like we're using here. The arithmetic, blend, and linear operations are specifically intended for floating point encodings, and all generate offspring that are some weighted linear combination of the parents.[1] The effectiveness of the float operators depends on the `weight` parameter, a vector of length n that gives weights to each parent (and, incidentally, determines the number of parents). The blend operation (sometimes called BLX-α) is slightly different: it always takes 2 parents, it takes a single weight parameter (0.5 is the default), and it generates a pair of weights stochastically for each position.

Try a number of crossover operators, and try to determine which performs best on this problem. Don't forget to pay attention to the population diversity, as well as the fitness of the best solution.

Question 11: *What effect does the choice of crossover operation appear to have on the population diversity?*

Plot 4: *Include a diversity plot for the operator that generated the best looking diversity curve. Be sure to specify which operator it was!*

The mutation operator can also have a profound effect on performance. There are only two operators provided here, uniform and inorder (the latter restricts possible mutations to a randomly selected region of the gene). However, there are a couple of different strategies for dynamically adjusting the mutation rate. The first strategy is to start with a large mutation rate, and decrease it over time. This strategy is controlled by setting the `mutate.decay` parameter to `'none'`, `'linear'`, or `'exponential'`. The second strategy is to base the probability of mutation on the fitness of the individual, where less fit individuals are more likely to mutate. This is controlled by setting `mutate.proportional` to true or false.

Try both of these strategies (you can also combine the two). You may need to increase the mutation probability, as the number given will now be the

[1]The geometric crossover is also a weighted average of multiple parents, but it generates imaginary values when the search space includes negative numbers.

maximum probability of a mutation.

Question 12: *Which (if any) of the dynamic mutation probability strategies improved the performance of the algorithm? What difference do you observe in the results?*

Now, let's revisit the replacement strategy. In generational GA, the offspring replace the parents each generation. So far, we've been replacing all the parents except for the best individual, but this might not be the best strategy. Another option is to only take offspring that have a better fitness. We'll call this *comparative* replacement (there doesn't seem to be a commonly accepted name for it). You can activate it by setting `replace.comparative` to true. Try comparative replacement now, using a few of the more promising combinations of parameters you've encountered up to this point.

Question 13: *Describe the effect (if any) comparative selection has on population diversity and convergence.*

Question 14: *What was the fitness value and position of the best solution you were able to locate? Report the parameter settings you used to generate that solution.*

Question 15: *Try using GAsolver to find the minimum for the 100 dimensional Ackley's function, using the same parameters as the last question. How well does it perform now? Can you improve the performance?*

Task 3: Particle Swarm Optimization

There are a number of Particle Swarm toolboxes available for MATLAB , although at this time there are none that are very polished. For this task, you'll be using an open source PSO toolbox modeled after the MATLAB Optimization Toolbox. You will need the file `psopt.zip`, which you can find on the lab web page. Unpack it into a directory you can access, and **add both the `psopt` directory and its sub-directories to the matlab path (by right-clicking the directory and selecting "Add to Path")**.

We'll start by exploring Ackley's function once more.

```
options = psooptimset('DemoMode','fast',...
        'PlotFcns',{@psoplotswarmsurf,@psoplotbestf},...
        'PopInitRange',[-32;32],'InertiaWeight',1,...
        'SocialAttraction',2,'CognitiveAttraction',2);
[x, fval] = pso(@ackleysfcn,20,[],[],[],[],[],[],[],options);
```

The first line creates a structure of parameters. You can view the available parameters by typing `psooptimset` with no arguments or return values. When you create an options structure like `options` here, you only need to specify fields and values that you want to differ from the defaults. You can pass another options structure to `psooptimset` as well, which will copy the old structure and then apply any changes. For example, to change only the number of generations specified by `options`:

```
options = psooptimset(options, 'Generations', 500);
```

The `pso` function takes a handle to the objective function, the number of dimensions in the problem, and the options structure. The empty arguments in the middle could be used to specify linear and nonlinear constraints, but we'll ignore them.

When you run the swarm, you'll get a window with the two plots specified above. One shows the particle positions and fitness surface of the function (in the first two dimensions of the problem only). The other shows the best and average fitness scores in each iteration.

In this first trial, we've used settings that replicate what you could call "classical" PSO. The resulting swarm behavior demonstrates one of the biggest problems with PSO algorithms, a phenomenon known as *swarm explosion*: the velocities of the particles steadily increase until the swarm flies apart. There have been lots of methods proposed to deal with swarm explosion over the years, and we'll investigate a few right now.

The simplest is *velocity clamping*: we impose a maximum velocity in each dimension, and any time a particle's velocity exceeds that maximum, we reset it to the maximum and continue. To try velocity clamping, you need to set the parameter `VelocityLimit` to $\delta(\max_i - \min_i)$ in each dimension, where $\delta \in (0, 1]$. For this problem, there are no real differences between the dimensions, so you can pass a single velocity limit and it will be applied to all dimensions. Try clamping with velocity limits of 30, 15, 5, and 1. In order to better see the effect of velocity clamping, add `@psoplotvelocity` to the list of plotting functions.

Question 16: *What happens to the maximum velocity of the swarm over time when using velocity clamping? How does the maximum velocity compare with the theoretical velocity limit (the red line at the top of the velocity plot)?*

A more interesting approach to controlling swarm explosion is by introducing *inertia weights.* Inertia in a PSO refers to the amount of influence a particle's current velocity has on the particle's next velocity. To understand the inertia weight, we need to consider the velocity equation for global best, or *gbest*, PSO (the effect on *lbest* and *pbest* is similar).

$$v_{id}(t) = wv_{id}(t-1) + c_1 r_1[p_{id} - x_{id}(t-1)] + c_2 r_2[p_{gd} - x_{id}(t-1)] \qquad (2)$$

Here $v_{id}(t)$ is the velocity of particle i in dimension d at time step t, $x_{id}(t)$ is the position of that same particle, p_{id} is the particle's personal best position, and p_{gd} is the global best position. The constants c_1 and c_2 are positive acceleration factors controlling the particles' cognitive and social components: high c_1 values cause the particle to have increased confidence in its own previous best, while high c_2 values cause the particles to have increased respect for the group's best position. You can change the values of c_1 and c_2 with the `CognitiveAttraction` and `SocialAttraction` options, respectively. Frequently the values assigned to these constants are somewhere around 2.

The inertia weight is another constant, w, controlled by the option `InertiaWeight`. Up to now we've used a value of $w = 1$, which replicates the *gbest* velocity equation without inertial weight. Setting $w \geq 1$ causes the particles to accelerate constantly, while $w < 1$ should cause the particles to gradually slow down. By carefully balancing the three parameters w, c_1 and c_2, it is possible to fine tune the algorithm, balancing both the social and cognitive components, and the tension between exploration and exploitation. There are a lot of suggested combinations for these values in the literature; for example $c_1 = c_2 = 2$ and $w = 1$, or $c_1 = c_2 = 1.49$ and $w = 0.7968$. Try those combinations now, as well as a few others, and see how well behaved the resulting swarms are. Be sure to try both with and without velocity clamping turned on.

Question 17: *What values for w, c_1 and c_2 worked best on this problem? How close did this swarm come to locating the global optimum?*

Question 18: *Was velocity clamping still necessary to prevent swarm explosion, or were you able to find a combination of values that kept the swarm together?*

Another very common modification is to let the inertia weight decay over time. To cause the weight to linearly decay, set the inertia weight to a vector $[w_{\max}; w_{\min}]$. A frequently used range is $[0.9; 0.4]$. Try switching off velocity clamping, and see if you can get the inertia weight set to some combination of values that reliably comes close to the optimum, and avoids the swarm explosion.

Question 19: *What was the best combination of decaying inertia weights, and social and cognitive coefficients? What was the best fitness value you found using that combination?*

At this point, it should be clear that the right combination of inertia weight, social and cognitive constants, and velocity clamping can yield a pretty reliable particle swarm. There is one more quite interesting method for avoiding swarm explosion that we should consider, however, which is called *constriction*. The constriction velocity equation for *gbest* is

$$v_{id}(t) = \chi[v_{id}(t-1) + \phi_1 r_1(p_{id} - x_{id}(t-1)) + \phi_2 r_2(p_{gd} - x_{id}(t-1))] \qquad (3)$$

χ is called the *constriction coefficient*. Letting $\phi = \phi_1 + \phi_2$, we further require that $\phi > 4$, and then define χ in terms of ϕ.

Question 20: *What is the equation for the value of the constriction coefficient in terms of ϕ?*

The advantage of constriction is that it does not require velocity clamping. As long as the parameters are selected as described, the swarm is guaranteed to converge. Unfortunately, the toolkit we are using does not include constriction. We can, however, implement constriction using the inertia weight approach. By choosing the right values of w, c_1 and c_2, we can make equation (2) be equivalent to equation (3) for a specific ϕ_1 and ϕ_2.

Question 21: *Consider the constriction equation with $\phi_1 = 4$, $\phi_2 = 2$. What is the constriction coefficient for these values? What values of w, c_1 and c_2 would we have to use in* (2) *in order to implement constriction with these values?*

Try constriction now for $\phi_1 = 4$, $\phi_2 = 2$, using the appropriate parameters in the inertia weight model.

Question 22: *Describe the behavior of the swarm when using constriction. Does it locate the global optimum? How quickly does it converge?*

Task 4: Global Trajectory Optimisation

At this point you should be familiar with using both GA and PSO. In this final task, you will apply both to a difficult real-world optimisation problem: calculating an optimal spacecraft trajectory.

In this problem, we have an interplanetary probe equipped with a chemical engine. The probe will be launched from Earth some time within a given launch window, with another planet or asteroid in the solar system as its destination. This is a Multiple Gravity Assist (MGA) problem, meaning that the probe will use the gravity wells of other planets to alter its trajectory along the way. The probe can apply thrust during one of these gravity assist encounters, and is also allowed to apply one course correction at some point between each pair of planets. The objective is to *minimise* the thrust used during the mission, as measured in kilometres per second of altered velocity.

This is a difficult class of problems, requiring a derivative-free global optimisation method such as GA or PSO. Presumably a global optimum exists for each problem, but it is not known. In the problem you will attempt now, the destination of the probe is Saturn, and the trajectory includes fly-bys of Earth, Venus, and Jupiter. The black box objective function is found in `mag_dsm.m`, and the problem description in `gtop_lab4.mat`. Note that the objective function is *not* vectorised; that is, to evaluate the fitness of a population, the objective function will be invoked once for each individual, which may lead to slower execution in MATLAB . This is a 22-dimensional problem, with hard upper- and lower-bound constraints for each dimension.

The best known solutions for this problem have an objective value in the range of 8–9 km/s[2]., but these solutions are *extremely* difficult to locate. It should be trivial to locate solutions over 30 km/s, but beyond that it is hard to give too strict a guideline as to what you should expect. We, i.e., the lab assistants, have a limited experience with trying to solve it ourselves—during the setup for this lab, none of the assistants observed a solution better than 17 km/s! Some of you will probably find a way to improve on that result—and we look forward to seeing how you do it—but this is not a requirement of the lab. Just find the best solution that you can less than trivial solutions.

Question 23: *To get started, download the file `gtop.zip` from the student portal. Extract the contents and add the folder to your* MATLAB *path, as always. However, before trying to solve this problem: make a hypothesis. Consider what little you know about the problem. As a further help, you may want to open the `mag_dsa.m` function file and read about what the variables represent, if you want to gather a rough picture of what we are trying to fit. Again, note that none of the assistants are any experts in this field, so this may be regarded as a semi-blackbox problem, which is part of the point of this Task. Now, make an educated guess: what is one choice for setting up either GA or PSO which you think might lead to better performance on this problem? For example, you could hypothesize that a particular crossover operation will be more effective than the others, or that higher values of social attraction will improve PSO performance. Your choice needs to be something you can test in* MATLAB *, and should be more than simply changing the number of iterations or population size. Briefly explain why you think that your choice might make a difference on this problem. (Note that there isn't a right or wrong answer here.) Now, when you have your hypothesis: double-click the file `gtop_lab4.mat` to add some problem variables to your workspace.*

[2]More information and the world records, the parameters and within what reasonable bounds these parameters are, can be found at the projects official web page: `http://www.esa.int/gsp/ACT/inf/projects/gtop/cassini2.html`

4.1 GA

The fitness function you need to use for `GAsolver` is a general MGA fitness function. You will need to pass it a struct that holds the definition of this specific optimisation problem. The following is a minimal setup for solving this problem with `GAsolver`:

```
mgapar = GAparams;
mgapar.stop.direction = 'min';
load('gtop_lab4.mat');
mgapar.objParams.problem = MGADSMproblem;
[best, fit, stat] = GAsolver(22, PopInitRange', 'mgadsm', ...
                                       50, 100, mgapar);
```

Important: you have to pass `PopInitRange'` as the bounds matrix to `GAsolver`, rather than simply `PopInitRange`.

The parameters should be within some bounds to make sense (e.g., that an angle does not have nonsensical values). You can check that the found parameters are within the stated bounds[3] (defined in the `MGADSMproblem` structure) by using the `verify` function.

```
[verification, lower, upper] = verify(best, MGADSMproblem);
```

If `verification` = 1, then all parameters are withing the bounds; else, one may follow `lower` or `upper` to see which parameter failed, i.e., equals 0.

Finding a GA setup that works well here will require some experimentation. Try running several short trials with different combinations of parameters. Once you settle on an approach that seems promising, you can run a larger trial (with more epochs and/or a larger population) to try to find a good solution.

Question 24 ★: *Describe your final GA setup, in enough detail that your experiment can be replicated (the output of `ga_show_parameters` can be useful here). Which settings seemed to have the most effect on your final results?*

Question 25: *What was the best solution you were able to locate? Be sure to include both the fitness value, and the full 22-dimensional vector representing the solution. How long did it take to run the solver?*

[3]See footnote 2.

4.2 PSO

For the PSO solver, the provided fitness function will handle only this specific problem. You will need to generate an appropriate options struct as follows:

```
options = pso_gtop_lab4('init');
```

This problem has hard boundary constraints: values outside those boundaries will cause the fitness function to throw an error. You will need to extract the bounds from the options structure you just generated, so that you can pass them to the `pso` function.

```
LB = options.PopInitRange(1,:);
UB = options.PopInitRange(2,:);
```

Now you can run the PSO. Make sure to pass the lower and upper bounds!

```
[x,fval] = pso(@pso_gtop_lab4,22,[],[],[],[],LB,UB,[],options);
```

Again, finding a PSO setup that works well requires some experimentation. Once you settle on an approach that seems promising, run a larger trial to try to find a good solution.

Question 26 ★: *Describe your final PSO setup, in enough detail that your experiment can be replicated (the output of `psooptimset` can be useful here). Which settings seemed to have the most effect on your final results?*

Question 27: *What was the best solution you were able to locate? Be sure to include both the fitness value, and the full 22-dimensional vector representing the solution. How long did it take to run the solver?*

4.3 Testing Your Hypothesis

In Question 23 you came up with a hypothesis about the performance of either GA or PSO. Now you will test that hypothesis.

You may need to recast your hypothesis slightly in order to have (at least) two possibilities to compare, giving you a statement like "approach X is better than approach Y." Next, design a simple experiment to test that hypothesis. You will need to run several tests for each possibility, say 5–10 runs each, depending on how long an individual run takes. The simplest way to measure performance here is to record the best fitness value found in each trial, but depending on your hypothesis you may be able to use another measure (e. g., time required to get fitness below a certain threshold, population diversity after X generations, etc.). Be imaginative!

Question 28 ★: *Describe your experimental setup, and summarise your results. Did the experiment support or contradict your hypothesis? Be persuasive. (Note: plots are persuasive.)*

Task 5: Wrapping up

We are constantly trying to improve the labs, and to help us, we appreciate your feedback.Please take a moment to briefly answer the following questions, we especially interested about the final task:

Question 29: *How well did today's lab help to reinforce concepts from the lectures? Is there a specific concept that the lab has helped to clarify for you? Is there a concept from the relevant lectures that should have been covered more?*

Question 30: *If you attended it, then how helpful did you find today's lab session? What could have been improved?*