4M17 Assignment 2

CCN: 5517B

January 2020

1 Introduction

In many different fields of engineering, a problem will often be encountered where an analytical solution is not possible, meaning numerical and algorithmic methods must be used. For some problems, this can be solved, but for a large number, the problem is NP-Hard, and therefore it is better to use a method with some influence of randomness, known as stochastic optimisation, to produce a good solution. A product of relying upon randomness is the uncertainty that a global minimum to a given function has been found. Therefore it is best to run these algorithms with the correct parameters based upon the problem, as well as repeating the algorithm for greater certainty that the best solution has been achieved.

2 Shubert's Function

The task is to find the minimum value of one of Shubert's Functions:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \sum_{j=1}^{5} j \sin((j+1)x_i + j), \quad \mathbf{x} \in \mathbf{R}^n$$

where n=5 for the given problem. However before solving the problem in 5D, it is best to analyse the problem in 1D, to learn some features of the problem as well as try to look for possible solutions to check validity of the algorithms later on.

The first thing to notice is the dimensional symmetry, as the function is a sum of the same value in each dimension, so the result scales with the number of dimensions, and x_i for each dimension should be the same at the minimum. Therefore we can easily produce a good answer for the function minimum by looking at the 1D function:

$$f(x) = \sum_{j=1}^{5} j \sin((j+1)x + j)$$

This is plotted in Figure 1, and shows there to be a clear minimum within the given bounds at around x=-1. We can improve upon this guess by applying a Newton method to the gradient of the function (to find the zeros in the gradient), and therefore give the x value at the function minimum. The code for this is given in the appendix, and results in an x value of -1.11409968758 (to a tolerance of 1e-12), and a function minimum of $f(x)_{min} = -14.8379500257$. Therefore when solving the problem in 5D, due to the dimensional symmetry, we expect the minimum value to be $f(\mathbf{x})_{min} = -74.18975012855296$.

3 Simulated Annealing

The first algorithm tested was Simulated Annealing. We begin by making the constrained an unconstrained one by applying a penalty function based on how far a value is outside the bounds. The effect of this is shown in Figure 6,

The algorithm works as a random walk, where new steps with a lower cost are accepted, and steps with higher cost are accepted with a probability based on the current 'temperature' and the increase in cost function. The initial temperature is decided based upon an initial survey of the function [1], and is found to be a value around 9.41 (average initial temperature based off a sample of 100 random starting locations followed by 500 random steps). This temperature is decremented ($T_{k+1} = \beta T_k$) every time the current chain of steps has reached a certain length (L_k), or if a certain amount of accepted solutions (η_{min}) have been achieved.

Each new step is decided based upon a scheme proposed by Parks [2]:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{D}\mathbf{u}$$

where u is a random vector ($u \in \mathbf{R}^n$) in the range (-1,1) and $\mathbf{D}$ is a diagonal matrix of the maximum change allowed in each control variable (Δx_{max}). When a new step is accepted, $\mathbf{D}$ is updated:

$$\mathbf{D}_{j+1} = (1-\alpha)\mathbf{D}_j + \alpha\omega\mathbf{R}$$

Figure 1: 1D version of Shubert's Function, showing minimum around x=-1.1141

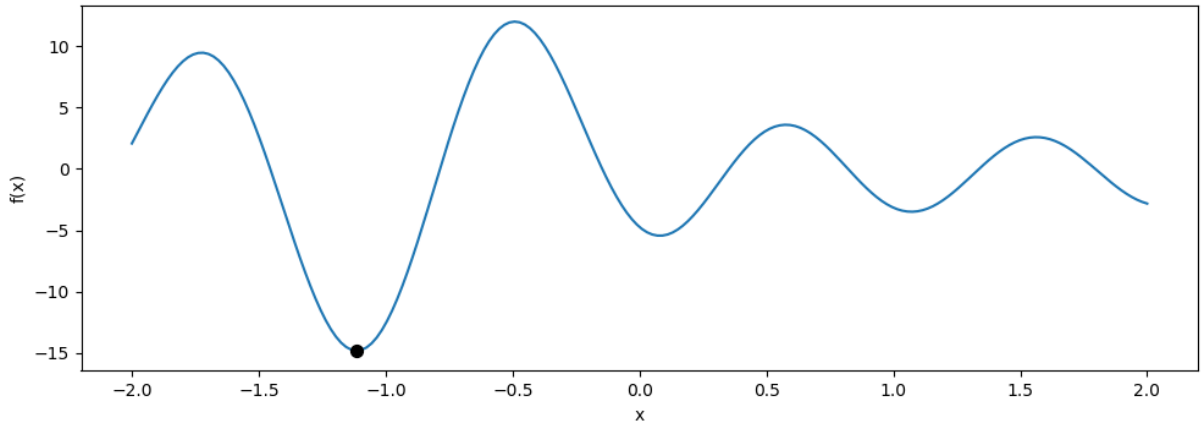
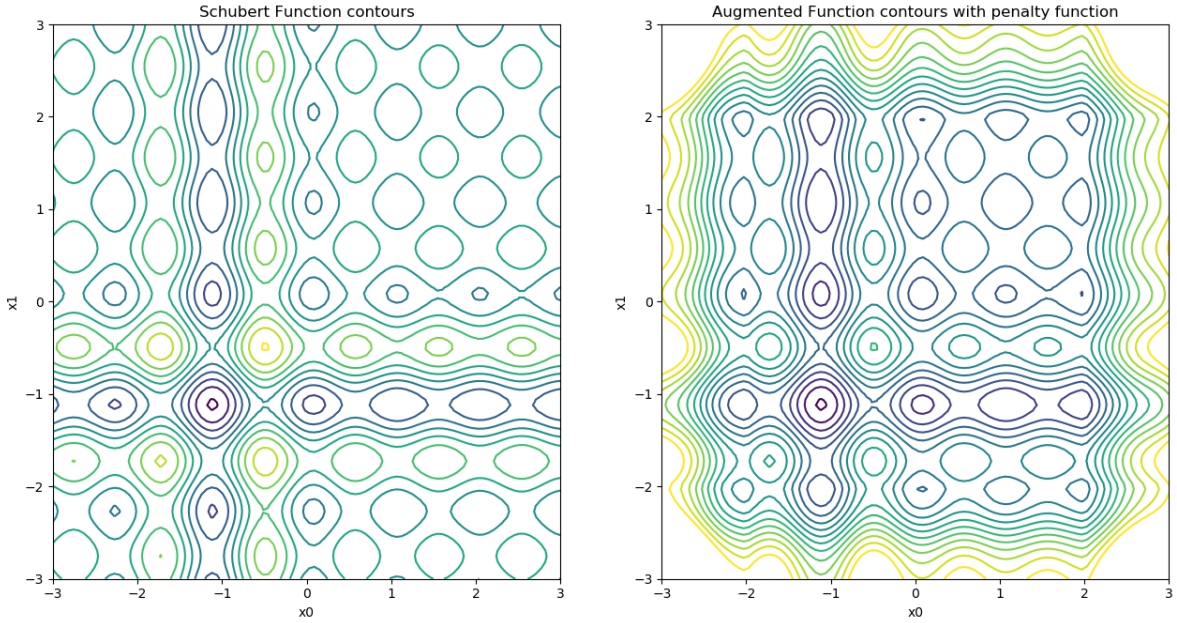


Figure 2: Contour plot of original and augmented functions, showing the effect of the penalty function on the search space



where as suggested [1], $\alpha = 0.1$, $\omega = 2.1$ and $\mathbf{R}$ is a diagonal matrix of the magnitudes of the changes in each control variable.

In addition to this, the probability of accepting a new solution when there is an increase in cost is changed to take into account the effectiveness of the change made, as well as still take into account the current solution temperature to allow the algorithm to find a more exact solution in its later stages.

This results in the parameters for the algorithm being β, L_k and Δx_{max}. The choice of these depends on the function being optimised, and therefore each of these are systematically varied to find the best combination for the given problem.

3.1 2D Performance

To test that the implementation of the Simulated Annealing Algorithm is correct, it is first tested against Shubert's Function in 2D. This is an easier problem to solve as the search space is smaller and there are fewer local minima than the 5D case. The convergence of the algorithm is shown in Figure 3. This shows the algorithm reached a value close to the minimum within 1000 accepted steps, and was restarted once as highlighted by the red line. Figure 4 shows the position of every 10th accepted step, and proves the algorithm is able to escape local minima and traverse the search space effectively.

Figure 3: Convergence of the Simulated Annealing Algorithm in 2D, with the red line showing the restart due to no accepted solutions in a chain

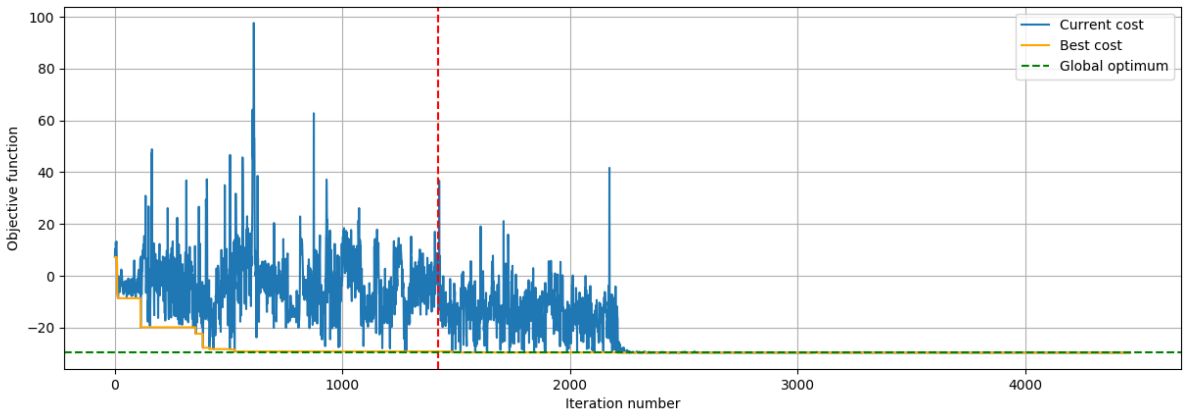
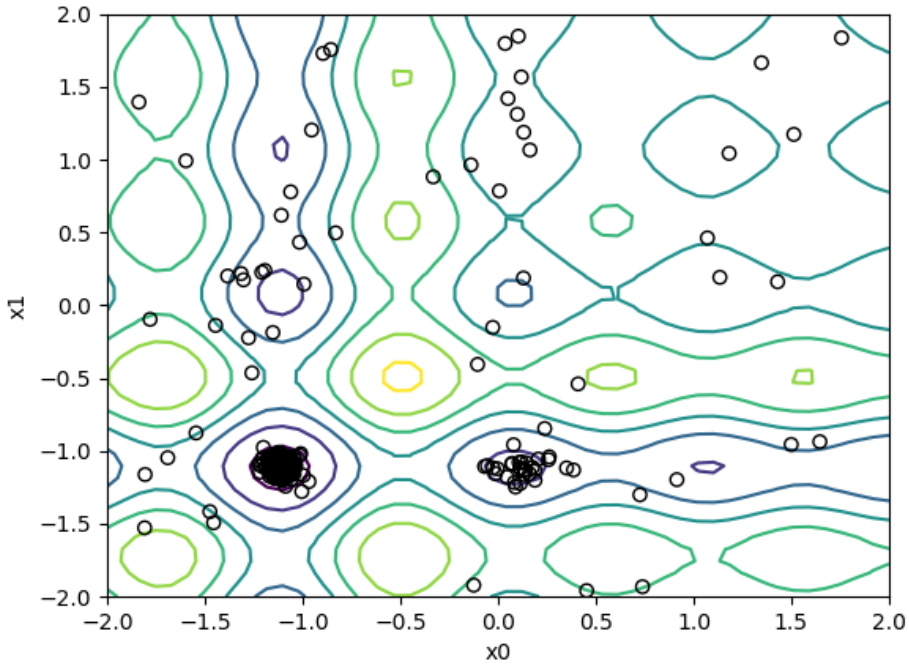


Figure 4: Accepted steps for the Simulated Annealing Algorithm in 2D, with the eventual minimum found at the densest cluster at $\mathbf{x} \approx [-1.14, -1.14]$

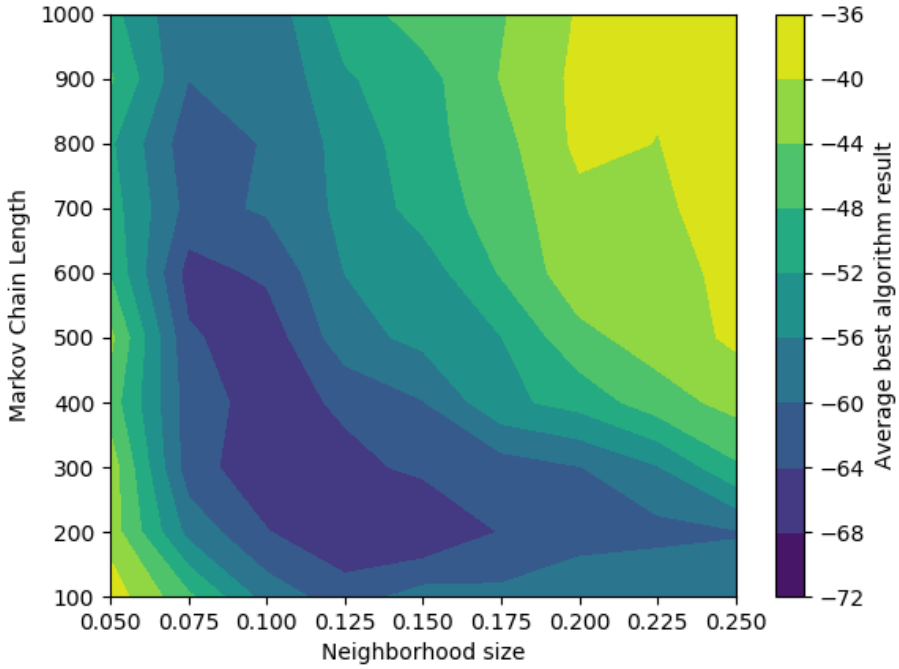


3.2 Choice of Parameters

With α set to 0.95[1], the two key parameters which affect the accuracy and reliability of the algorithm are the Neighborhood size and the maximum Markov Chain Length, L_k. We expect there to be a trade off in neighborhood size, as very small will mean the algorithm cannot traverse the feasible space fast enough before cooling, and too large will lead to a random sampling of the space rather than an optimisation. This is shown well in [3], where it is recommended the optimal neighborhood size for a travelling salesman problem of 128 points is 3.

Figure 5 shows the average value achieved by the algorithm when tested with 20 random seeds, and clearly shows the region centered at (0.1, 300) where the best performance is achieved. This data is only applicable to the 5D version of the problem, and the parameters must be tuned differently for different problems and dimensions.

Figure 5: Average performance of SA algorithm with variation in parameters over a sample of 20 tests per data point



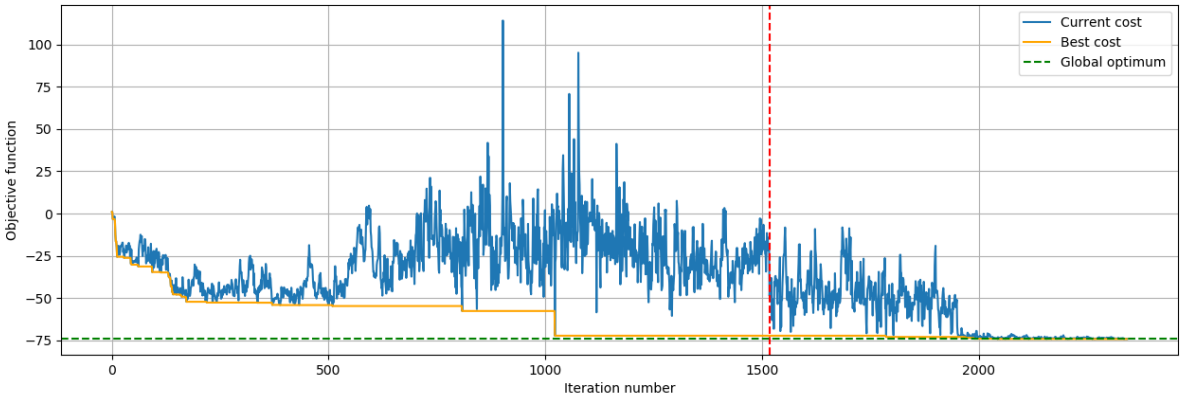
3.3 Variations in Implementation

Although the algorithm is well defined in [3], there are various variations and improvements proposed later which affect the performance of the algorithm on Shubert's Function.

The first variation is the decision between using a random step within the specified neighborhood, or to use the method proposed by Parks [2], as detailed above. When tested over 25 random seeds, using the optimal parameters as detailed above, the simple random step achieved an average value of -47.495, whereas the method proposed by Parks achieved an average value of -68.100. This is a clear improvement and therefore for the final version the latter variation is used.

Secondly, the possibility of restarting the algorithm is tested to see if this achieves a better optimum value on average. Without the possibility of restarts, the average value achieved over 25 random seeds was -65.567. In comparison, the average optimum value achieved with a restart allowed if 1000 steps had been made without any improvement was -68.100. Reducing this limit to 500 steps meant a slight decrease in performance to an average value of -67.784, and an increase to 2000 led to an average value of -65.887,so a restart limit of 1000 was chosen for the final implementation.

Figure 6: Convergence of Shubert's function in 5D with iteration number using the Simulated Annealing Algorithm, with red lines showing the restart points



4 Particle Swarm Optimisation

Initially proposed by Kennedy and Eberhart [4], this method is based upon the dynamics of a swarm of particles, and looks to avoid local minima by considering the best solutions of the entire swarm as well as an individual. Following a swarm initialisation within the feasible space , a random velocity ($\mathbf{v}$) for each particle is assigned (within bounds), and at each time-step the velocity and new position ($\mathbf{x}$) of the particle is calculated:

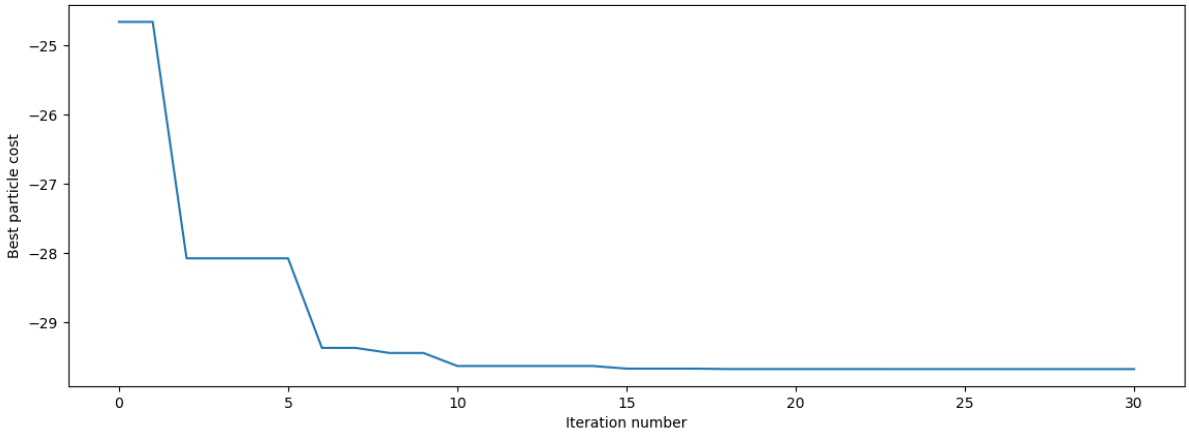
$$\mathbf{v}_{new} = \omega \mathbf{v}_{old} + \phi_p \mathbf{r}_p(\mathbf{p} - \mathbf{x}) + \phi_g \mathbf{r}_g(\mathbf{g} - \mathbf{x})$$

where $\mathbf{r}_p$ and $\mathbf{r}_g$ are random vectors between 0 and 1, $\mathbf{p}$ is the particle's best known position, and $\mathbf{g}$ is the swarm's best known position. The three scalar parameters represent the decay of velocity and the weighting of individual optimum and swarm optimum when calculating the velocity at the next time-step. This means a balance must be achieved between ϕ_p and ϕ_g to ensure the swarm is able to locate the global minimum without ignoring a large set of possible minimums (so as not to get stuck in a local minimum found early in the algorithm). In order to implement the constraints into the problem, the positions of particles are limited to the given bounds. This is done by checking when a new position has been calculated, and if the magnitude is greater to 2 then it is set to the corresponding limit. This has a minor drawback of resulting in more probability of particles ending at the edge of the feasible space, but this did not seem to hinder the effectiveness of the algorithm when tested on both 2D and 5D cases.

4.1 2D performance

In order to test whether the Particle Swarm method was working correctly, the algorithm was initially tested on the 2D version of Shubert's Function. The convergence and particle positions are the stated time-steps are given in Figures 7 and 8

Figure 7: Convergence of Particle Swarm Optimization on the 2D Shubert's function



A minimum of -29.6758784372 was found at [-1.11378195 -1.1142524] with a swarm size of 30, in 30 iterations, therefore taking 900 function evaluations. This gives an absolute error of 0.000352 in the position of the minimum, and an error of 0.0000728% for the minimum of the function.

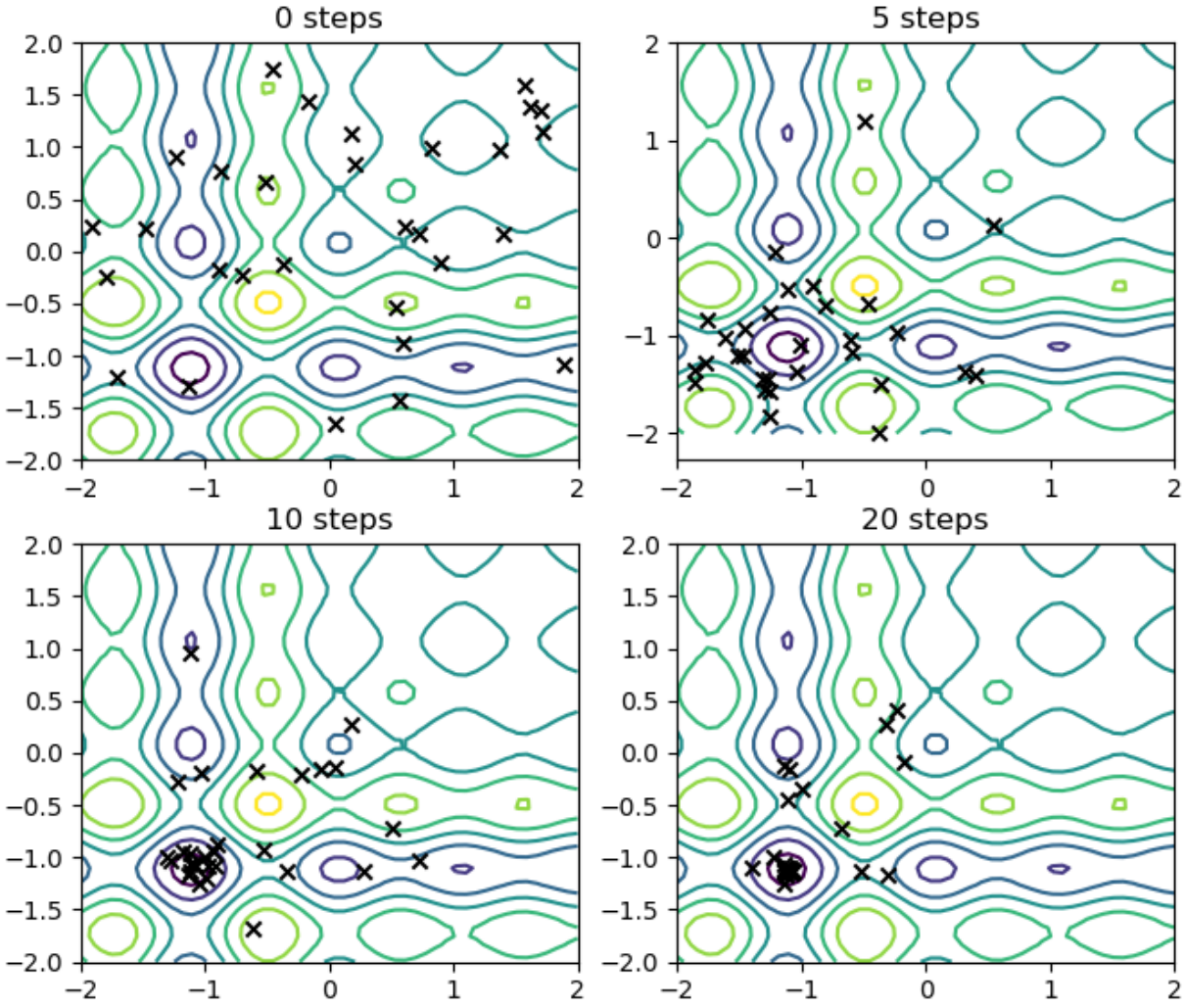
4.2 Parameter choice

As stated above, the choice of parameters for this algorithm is key to ensure the entire feasible space is searched effectively. Figure 9 shows that the choice of these parameters is not as clear cut as for Simulated Annealing, but highlights the improved algorithm performance overall (seen through the darker colours, and hence a lower cost achieved) when using $\omega = 0.9$. Following from this, the region where $\phi_g = 1.75$ and $\phi_p = 1.1$ appears to be the best performing set of parameters for this problem.

This however is not easily to visualise as there are three parameters (in addition to swarm size and iterations) to optimise over. In order to achieve the best set of parameters, it is possible to carry out meta-optimisation, where the new control variables are the parameters and the function is the result of the algorithm on the original function. Using the `scipy.optimize` package [5], the optimum was found to be:

$$\omega = 0.68255, \quad \phi_p = 0.60516, \quad \phi_g = 0.35750$$

Figure 8: Particle positions at the given time-step for PSO solution of the 2D Shubert's Function



Using this, the algorithm convergence is shown in Figure 10. The convergence plot is, as expected, strictly decreasing, as only better swarm optimums will be accepted. In addition to this, the archive of solutions contains the best solution for each particle, most of which are the global optimum by the end of the algorithm, but also can contain local minima. This means the algorithm is well suited to finding minima to problems which have multiple global minima, or where it is useful to find various local minima as methods to solve a problem.

4.3 Variation in Implementation

There are many variations of the implementation of Particle Swarm Optimisation, and the current "standard" is defined as SPSO-2011 [6], which acts as a benchmark for any further improvements. The implemented version for this coursework is somewhat more simple, but an investigation is made below as to a method to prevent premature stagnation of the algorithm.

Literature suggests that the parameter ω acts as a inertia weight [7] for each particle. A larger weight will mean a greater degree of exploration of the search space, but a reduction in exploitation (the following of a reduction to the best possible local minimum. [7] also suggests a variety of strategies for varying the inertia weight throughout the algorithm. The first obvious strategies are a linearly decreasing inertia weight and an exponentially decreasing inertia weight, which both give high exploration at lower iteration numbers, and higher exploitation later in the process. In addition to these, a varying random inertia weight and an oscillating inertia weight are also tested against the baseline constant inertia weight to see if there is an improvement.

Figure 9: Effect of parameter choice on Particle Swarm Optimisation average performance over 20 random seeds, with a population size of 20 and 200 maximum iterations in 5D

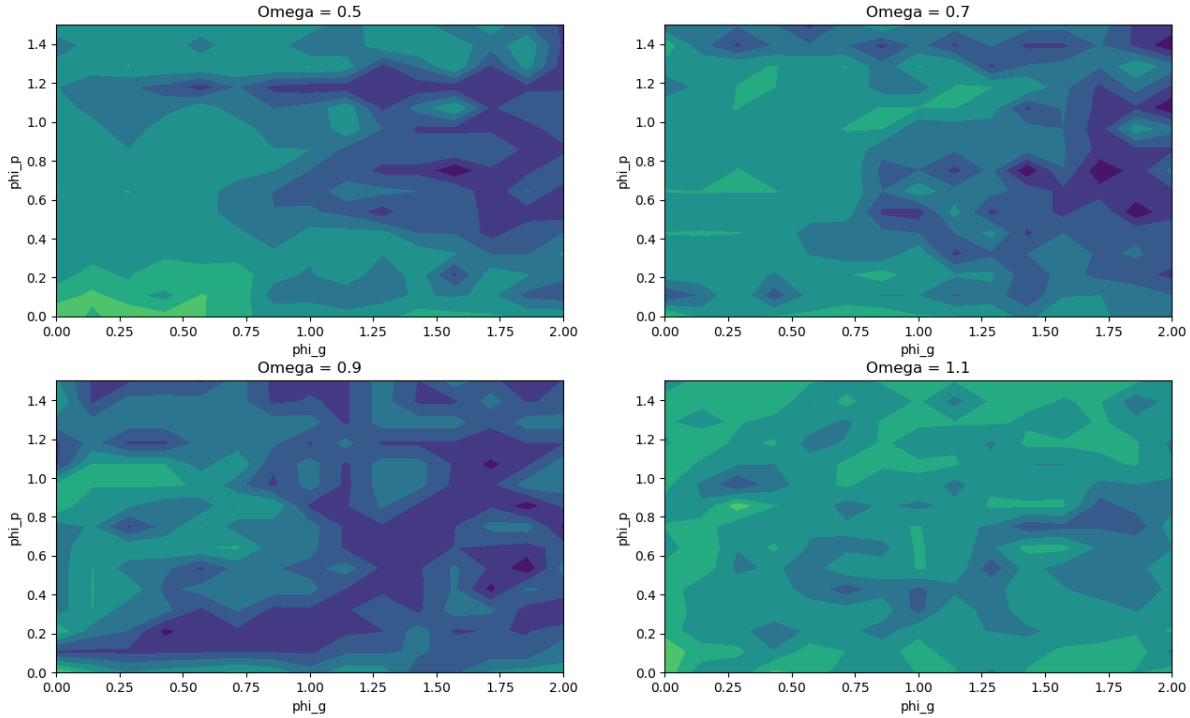
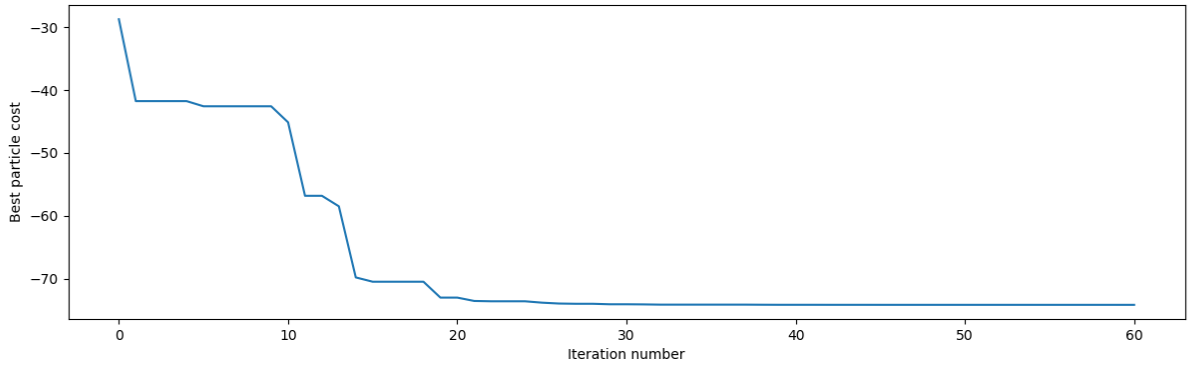


Figure 10: Convergence of the PSO on Shubert's Function in 5D



Inertia Weight Strategy	Average Value	"Good" solutions (%)
Constant (0.682)	-64.970	26
Linearly Decreasing	-51.869	3
Exponentially Decreasing	-51.	3
Random (± 10%)	-65.472	30
Oscillating (±10%)	-54.708	3

where a "good" solution is one within 1% of the known global minimum (lower than -73.44). From the table above, we can see that for the given problem parameters found in the earlier section, the only inertia weight scheme with any benefit is to have a random inertia weight at every time-step which is ±10% around the original inertia weight.

The main issue encountered when using PSO is premature convergence, that is the stopping of the algorithm at a local minimum rather than pursuing a further global minimum. A method to reduce the chance of this occurring is the detect the point at which most of the particles have converged, and reset the particles in a space or around this local minimum, in an attempt to re-introduce diversity to the population. This can be done in a variety of ways, and literature suggests [8] this can be easily done by comparing the opposite point around the known swarm minimum, and accepting the new position if the function is lower at this opposite position. Adding this means extra function evaluations, and therefore reduces the maximum

possible iterations of the PSO algorithm, but may lead to better exploration of the solution space. In order to implement this, a condition must be met to decide whether the particles had sufficiently converged. This was done by calculating the average euclidean distance from the swarm optimum, and if this was less than ϵ times the max side length of the feasible space, the algorithm was said to have converged. After adding this, the average value found was -66.769, with 43% "good" solutions. This is a significant increase in the number of "good" solutions, and therefore is of benefit to include in the final algorithm.

A final change in implementation which is possible is to reduce the rate of information flow through the swarm. This is achieved by the introduction of another parameter, the local optimum, which may be different for every particle. On finding a new particle optimum solution, a particle is able to inform particles in the neighborhood. This means all particles will not converge on a single minimum, but instead multiple minima will be found, therefore improving the exploration of the search space without causing a large negative to the exploitation of the algorithm. This is the method followed by SPSO-2011, and is said to scale well with large increases in dimensions [6]. On the addition to the implementation, there was found to be no benefit to the average best solution obtained, and therefore it has been omitted from the final code. The benefit of this strategy was to more local minima at the end of the iterations, but as the task for this function is to find the single global minimum, this is not of the highest importance.

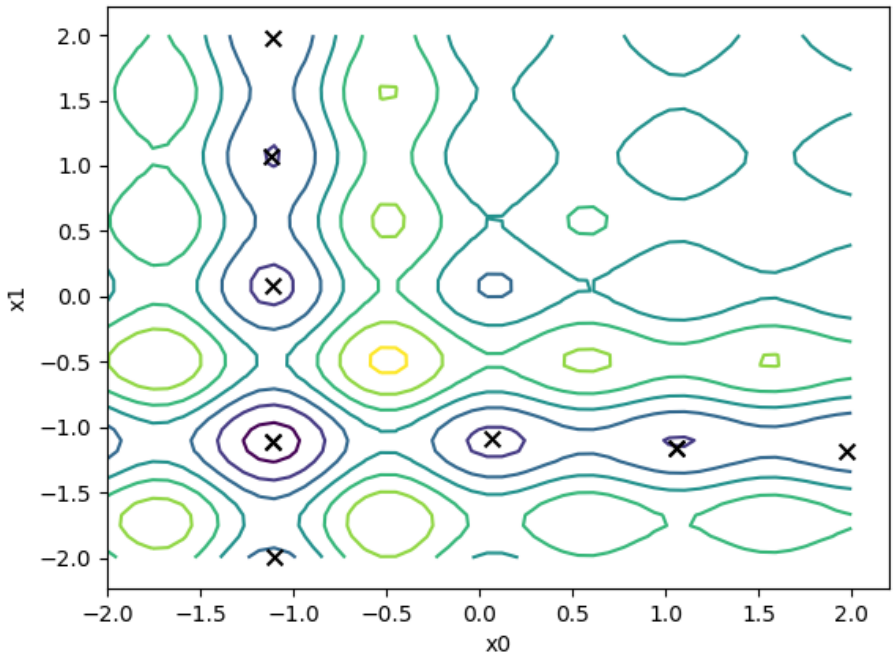
5 Comparison of Algorithms

When comparing the performance of two algorithms, many different aspects can be looked at to see which is better. Often one algorithm will perform well for a certain task or objective function, but for this comparison only the performance against Shubert's Function will be examined.

Firstly, the performance of each algorithm in the minimum found is compared, with a limit of 10000 function evaluations. Simulated Annealing was able to find a minimum to within 0.0004% of the known global minimum, whereas Particle Swarm Optimisation was able to 1.73×10^{-7}%. Therefore there is a margin gain in accuracy when using PSO, but both provide good estimates of the global minimum and therefore both work well for the given problem.

It is also best to look at the ability of each algorithm to generate a range of dissimilar solutions, to help provide more information about the function which is being optimised. Using archiving of dissimilar solutions as detailed in [1], SA is able to find a variety of local minima. Figure 11 shows the archived dissimilar solutions for the 2D Shubert's Function, and does well to identify a large amount of local minima.

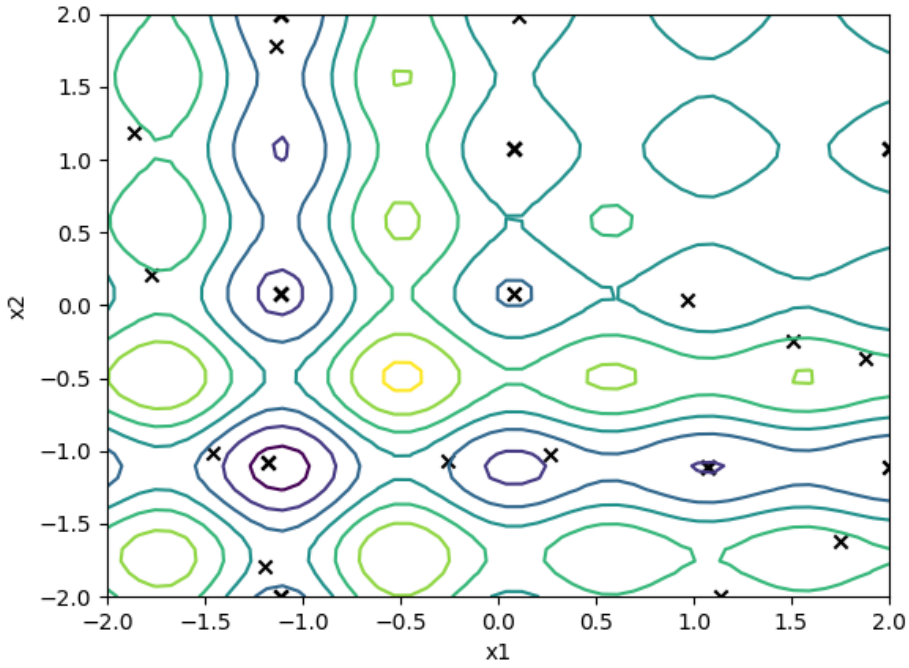
Figure 11: Positions of the archived best dissimilar solutions when using Simulated Annealing on the 2D Shubert's Function



Particle Swarm Optimisation presents a tricky problem when finding dissimilar best solutions, as by the end of the iterations, most (if not all) of the particles have converged to a single minimum (which in some cases is the global minimum). By changing the parameters such that there is a stronger bias towards a particle's best solution rather than the swarm's best

solution, the algorithm is able to generate a good range of dissimilar solutions. It is also possible to achieve this by stopping particles from communicating with the entire swarm, and instead allowing them only to communicate with particles in their neighborhood. The neighborhood can be defined as any particles whose euclidean distance to a certain particle is less than a parameter. By setting this neighborhood size parameter to be a small fraction of the search space size, we can stop all particles from converging to a single minimum, so generating a large amount of dissimilar best solutions. Figure 12 shows the results of setting this parameter to 0.1.

Figure 12: Particle positions after 100 iterations using the PSO on the 2D Shubert's Function with the neighborhood size set to 0.1



We can see that both algorithms are able to generate a good selection of dissimilar solutions, but that PSO is limited by the swarm size. Furthermore, the complexity of the algorithm is increased by a factor of $Nlog(N)$ (N = swarm size) when implementing the neighborhood communication as the particles must be sorted to find those close to a specific particle.

Finally we compare the consistency of each algorithm with the parameters found above, to decide which achieves a lower value on average with 10000 function evaluations. The results are detailed in the table below:

Algorithm	Average Value	"Good" solutions (%)
SA	-66.956	40
PSO	-64.604	56

There is little difference between the two, but it appears there is a marginal improvement in average value obtained when using Simulated Annealing, at the cost of a larger standard deviation in results, and therefore fewer "good" solutions.

A Function Analysis Code

Code used to find the minimum of the 1D Shubert's Function, as well as plot the function and its minimum

```
# Import necessary libraries
import numpy as np
from matplotlib import pyplot as plt
from scipy import optimize as opt

def f(x):
    #1D Shubert Function
    sum = 0
    for i in range(1,6):
```

```
        sum += i*np.sin((i+1)*x + i)
    return sum

def df_dx(x):
    #1st Derivative
    sum = 0
    for i in range(1,6):
        sum += i*(i+1)*np.cos((i+1)*x + i)
    return sum

def d2f_dx2(x):
    #2nd Derivative
    sum = 0
    for i in range(1,6):
        sum += -i*(i+1)**2*np.sin((i+1)*x + i)
    return sum

def newton(x0,f,gradf, tolerance):
    # Newton method to find zero in given function
    x = x0
    while abs(f(x)) > tolerance:
        x = x - f(x)/gradf(x)
    return x

x0 = -1
tol = 1e-12
# solve to find the zero in the gradient of the function, starting near the obvious minimum to ensure converge
x_min = newton(x0, df_dx,d2f_dx2,tol)

print("Minimum of f(x)={} found at x={} to a tolerance of {}".format(f(x_min), x_min, tol))

# Plot 1D function and results of minimum location
x = np.linspace(-10,10,200)
y = f(x)
plt.plot(x,y, zorder=1)
plt.xlabel("x")
plt.ylabel("f(x)")
plt.scatter(x_min, f(x_min), color='black', s=50, zorder=2)
plt.show()
```

B Simulated Annealing Code

```
# Import necessary libraries
import numpy as np
from matplotlib import pyplot as plt
from scipy import optimize as opt

class Problem:
    """Problem class, to initialise and solve problem via SA algorithm"""
    def __init__(self, max_iterations, dimensions, bound, parameters):
        # Initialise problem parameters and solution archive

        self.max_iterations = max_iterations
        self.dimensions = dimensions
        self.bound = bound
        self.parameters = parameters

        self.x = np.random.uniform(-bound, bound, dimensions)
        self.temperature = np.inf
```

```
        self.cost = self.f_penalty(self.x)
        self.d = np.diag(parameters[1]*np.ones(self.x.size))
        self.all_x = [self.x]
        self.all_costs = [self.cost]

        self.archive_length = 10

        self.archive = {
            "best_x": self.x,
            "best_cost": self.cost,
            "best_index": 0,
            "resets":[],
            "best_dissimilar_cost":[],
            "best_dissimilar_x":[],
            "all_best_costs":[]
            }

    def update_archive(self, iteration):
        # Updating problem archive

        # Update best known cost
        if self.cost < self.archive["best_cost"]:
            self.archive["best_x"] = self.x
            self.archive["best_cost"] = self.cost
            self.archive["best_index"] = iteration

        # Fill out dissimilar solutions if not full
        if len(self.archive["best_dissimilar_cost"]) < self.archive_length:
            self.archive["best_dissimilar_cost"].append(self.cost)
            self.archive["best_dissimilar_x"].append(self.x)
        else:
        # If not full, compare against archive to try to replace a known solution
            dissimilar = True
            for solution in zip(self.archive["best_dissimilar_x"], self.archive["best_dissimilar_cost"]):
                if np.linalg.norm(solution[0]-self.x, 2) < 0.5:
                    dissimilar = False
                    if self.cost < solution[1]:
                        pos = self.archive["best_dissimilar_cost"].index(solution[1])
                        self.archive["best_dissimilar_x"][pos] = self.x
                        self.archive["best_dissimilar_cost"][pos] = self.cost

            # If dissimilar and a better solution, replace maximum item of dissimilar solutions
            if dissimilar and self.cost < max(self.archive["best_dissimilar_cost"]):
                pos = self.archive["best_dissimilar_cost"].index(max(self.archive["best_dissimilar_cost"]))
                self.archive["best_dissimilar_x"][pos] = self.x
                self.archive["best_dissimilar_cost"][pos] = self.cost

    def probability_accept(self, new_cost, x_step):
        # Probabilty of accepting new solution
        d_bar = np.sum(np.abs(x_step))
        if new_cost < self.cost:
            return 1
        else:
            return np.exp(-(new_cost-self.cost)/(self.temperature*d_bar))

    def initial_survey(self):
        # Survey of function to find initial temperature
        temp_x = np.random.uniform(-bound, bound, dimensions)
        all_cost_steps = []
```

```
    #Take 500 random steps and find average cost step size
    for i in range(500):
        step = self.parameters[1]*np.random.uniform(-1, 1, self.x.size)
        cost_step = self.f_penalty(temp_x + step) - self.f_penalty(temp_x)
        if cost_step > 0:
            all_cost_steps.append(cost_step)
        temp_x = temp_x + step
    avg_cost_step = sum(all_cost_steps)/len(all_cost_steps)
    chi_0 = 0.8
    return -avg_cost_step/np.log(chi_0)

def check_restart(self, restart_count):
    restart_condition = 2000
    if restart_count - self.archive["best_index"] > restart_condition:
        # print("No new best solution found in {} iterations, restarting at best known solution".format(re
        return True
    return False

def solve(self, plot_convergence=False, plot_path=False):
    # Find initial temperature by conducting survey of function
    self.temperature = self.initial_survey()

    iterations = 0
    acceptances = 0
    restart_count = 0

    # Main algorithm loop
    while iterations < self.max_iterations:
        iterations+=1
        restart_count+=1

        # Take new step in positon
        x_step = np.dot(self.d, np.random.uniform(-1, 1, self.x.size))
        new_x = self.x + x_step
        new_cost = self.f_penalty(new_x)

        #Decide on whether to accept new solution and update if True
        if np.random.rand(1) < self.probability_accept(new_cost, x_step):
            acceptances += 1
            self.all_x.append(new_x)
            self.all_costs.append(new_cost)
            self.x = new_x
            self.cost = self.f_penalty(self.x)
            self.update_d(x_step)
            self.archive["all_best_costs"].append(self.archive["best_cost"])

        #Update archive
        self.update_archive(iterations)

        #Check for algorithm restart and restart if true
        if self.check_restart(restart_count):
            self.x = self.archive["best_x"]
            self.cost = self.archive["best_cost"]
            self.archive["resets"].append(len(self.all_x))
            restart_count = 0

        # Check for early termination if no better solutions found in chain
        if iterations % self.parameters[2] == 0 or acceptances > self.parameters[2]:
            if acceptances == 0:
                print("no acceptances in chain, exiting")
```

```
                break
            self.temperature = self.parameters[0]*self.temperature
            acceptances = 0

    # Output results
    print("Exited after {} iterations".format(iterations))
    print("Best objective function value: {}".format(self.archive["best_cost"]))
    print("Best value found at: {}".format(self.archive["best_x"]))

    # Plot results if necessary
    if plot_convergence:
        self.plot_convergence()

    if plot_path:
        self.plot_path()

    return self.x, self.cost, self.all_x, self.all_costs

def plot_convergence(self):
    # Method to plot convergence of algorithm best cost against iteration number

    #Plot current algorithm value
    plt.plot(self.all_costs, label="Current cost")
    plt.xlabel('Iteration number')
    plt.ylabel('Objective function')

    # Dotted red line for each reset point
    for xi in self.archive["resets"]:
        plt.axvline(xi, ls="--", color="red")

    # Plot best cost so far
    plt.plot(self.archive["all_best_costs"], color="orange", label="Best cost")

    #Plot function minimum
    plt.axhline(-14.83795*self.x.size, ls="--", color="green", label="Global optimum")
    plt.legend()
    plt.grid(True)
    plt.show()

def plot_dissimilar_solutions(self):
    # Method for plotting all dissimilar solutions in archive
    X,Y,Z = self.contours()
    plt.contour(X,Y,Z, zorder=-1)
    [plt.scatter(x[0], x[1], facecolors='black', edgecolors='black', zorder=1, marker='x', s=50) for x in
    plt.xlabel("x0")
    plt.ylabel("x1")
    plt.show()

def contours(self):
    # Method for supplying arrays for contour plots
    x = np.linspace(-2,2)
    y = np.linspace(-2,2)
    X, Y = np.meshgrid(x,y)
    Z = self.f(np.array([X,Y, np.zeros(X.shape), np.zeros(X.shape), np.zeros(X.shape)]))
    return X,Y,Z

def plot_path(self):
    # Plot path followed by algorithm
    plt.axes(xlim=(-self.bound, self.bound), ylim=(-self.bound, self.bound))
    X,Y,Z = self.contours()
```

```
    plt.contour(X,Y,Z, zorder=-1)
    subset_x = []
    for i, x in enumerate(self.all_x):
        # Include every 1 in 10 steps
        if i%10 == 0:
            subset_x.append(x)
    [plt.scatter(x[0], x[1], facecolors='none', edgecolors='black', zorder=1) for x in subset_x]
    plt.xlabel("x0")
    plt.ylabel("x1")
    plt.show()

def plot_f_adj(self):
    # Plot contours of standard and augmented function
    levels = np.linspace(-50,50,20)
    fig, axs = plt.subplots(1, 2)
    x = np.linspace(-3,3, 100)
    y = np.linspace(-3,3, 100)
    X, Y = np.meshgrid(x,y)
    Z = self.f(np.array([X,Y]))
    Z_penalty = self.g_penalty(np.array([X,Y]))
    contour0 = axs[0].contour(X,Y,Z, levels, cmap="viridis")
    axs[0].set_title("Schubert Function contours")
    axs[0].set_xlabel("x0")
    axs[0].set_ylabel("x1")
    contour1 = axs[1].contour(X,Y,Z_penalty, levels, cmap="viridis")
    axs[1].set_title("Augmented Function contours with penalty function")
    axs[1].set_xlabel("x0")
    axs[1].set_ylabel("x1")
    plt.show()

def g_penalty(self, x):
    # Penalty function for showing contours on 2D plot
    penalty = np.zeros(x.shape[1:])
    for i in range(penalty.shape[0]):
        for j in range(penalty.shape[1]):
            c_v = 0
            for k in range(x.shape[0]):
                if np.abs(x[k,i,j]) > bound:
                    c_v +=  np.abs(x[k,i,j])-self.bound
            penalty[i,j] += 50*c_v
    return self.f(x) + penalty

def f_penalty(self, x):
    # Augmented Shubert's Function with linear penalty function
    w = 1000*np.ones(x.shape)
    c = np.zeros(x.shape)
    for i, x_i in enumerate(x):
        if np.abs(x_i) > bound:
            c[i] = np.abs(x_i) - bound
    return self.f(x) + np.dot(w.T, c)/self.temperature

def f(self, x):
    # Shubert's n-dimensional function
    total = 0
    for i in range(x.shape[0]):
        for j in range(1,6):
            total += j*np.sin((j+1)*x[i]+j)
    return total

def update_d(self, x_step):
```

```
        # Parks 1990 method for finding new trial solutions
        alpha = 0.1
        omega = 2.1
        # Values for alpha and omega based of SA lecture notes
        self.d = (1-alpha)*self.d + alpha*omega*np.diag(np.abs(x_step))

def parameter_problem(u):
    # Parameter problem for multiple runs or meta-optimisation
    good = 0
    max_iterations = 10000
    dimensions = 5
    total = 0
    tests = 50
    for i in range(tests):
        np.random.seed(i)
        problem = Problem(max_iterations, dimensions, bound, u)
        problem.solve()
        total += problem.archive["best_cost"]
        if problem.archive["best_cost"] < -73.44:
            print(i)
            good+=1
    print(good)
    return total/tests

# Set random number seed
np.random.seed(1)

# Problem parameters
max_iterations = 10000
dimensions = 5
bound = 2
alpha = 0.95
neighborhood_size = 0.1
chain_length = 300

# Run problem multiple times to find average value
print(parameter_problem([alpha, neighborhood_size, chain_length]))

# Run simple problem and plot results
problem = Problem(max_iterations, dimensions, bound, [alpha, neighborhood_size, chain_length])
problem.solve(plot_convergence=True, plot_path=True)
problem.plot_dissimilar_solutions()
```

C Particle Swarm Optimisation Code

```
# Import necessary libraries
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation
import matplotlib.animation as animation
from scipy import optimize as opt

class Problem:
    """Problem class, to set up swarm and solve using PSO"""

    def __init__(self, iterations, swarm_size, parameters, bounds, dimension):
        # Initialise problem variables and swarm
        self.iterations = iterations
```

```
        self.swarm_size = swarm_size
        self.swarm = Swarm(dimension, swarm_size, bounds, parameters, self.f, self)
        self.resets = []
        self.inertia = parameters[0]

    def f(self, x):
        # Shubert's n-dimensional function
        total = 0
        for i in range(x.shape[0]):
            for j in range(1,6):
                total += j*np.sin((j+1)*x[i]+j)
        return total

    def solve(self, convergence_plot=False, animation=False, progression_plot=False):
        # Initialise variables for storing history of best swarm values
        all_best_costs = [self.swarm.swarm_best_cost]
        all_x = []
        new_x = []
        for i in range(self.swarm_size):
            new_x.append(self.swarm.swarm[i].x)
        all_x.append(new_x)

        # Main loop
        for i in range(self.iterations):
            self.check_reset(i)

            # update swarm at each time-step
            self.swarm.update(i, self.iterations)

            # Store new values in relevant arrays and variables
            all_best_costs.append(self.swarm.swarm_best_cost)
            new_x = []
            for i in range(swarm_size):
                new_x.append(self.swarm.swarm[i].x)
            all_x.append(new_x)

        self.solution = [self.swarm.swarm_best_x, self.swarm.swarm_best_cost, all_x, all_best_costs]

        # Output found solution after all iterations complete
        print("Final minimum x: {}".format(self.solution[0]))
        print("Final minimum f(x): {}".format(self.solution[1]))

        # Plot if necessary
        if convergence_plot:
            self.convergence_plot()
        if animation:
            self.animation()
        if progression_plot:
            self.progression_plot()

        return self.solution

    def check_reset(self, iteration):
        #check to see if swarm has converged
        return # used here to disable this feature
        all_dist = []
        total = 0
        epsilon = 0.01

        for particle in self.swarm.swarm:
```

```
            all_dist.append(np.linalg.norm(particle.x - self.swarm.swarm_best_x, 2))
        average = sum(all_dist)/len(all_dist)
        if average < (bounds[1]-bounds[0])*epsilon:
            self.resets.append(iteration)
            self.optimum_reset(max(all_dist))

    def optimum_reset(self, dist):
        # If swarm converged, check for better solutions on opposite of known global minimum
        for particle in self.swarm.swarm:
            if self.f(particle.x + 2*(self.swarm.swarm_best_x - particle.x)) < self.f(particle.x):
                particle.x = particle.x + 2*(self.swarm.swarm_best_x - particle.x)
            # particle.x =  particle.x
            # particle.velocity = np.random.uniform(low=bounds[0], high=bounds[1], size=(dimension,))

    def convergence_plot(self):
        # Method to plot convergence of algorithm against iteration number
        plt.plot(self.solution[3])
        plt.xlabel('Iteration number')
        plt.ylabel('Best particle cost')
        for xi in self.resets:
            plt.axvline(xi, ls="--", color="red")
        plt.show()

    def contours(self):
        x = np.linspace(-2,2)
        y = np.linspace(-2,2)
        X, Y = np.meshgrid(x,y)
        Z = self.f(np.array([X,Y, np.zeros(X.shape), np.zeros(X.shape), np.zeros(X.shape)]))
        return X,Y,Z

    def animation(self):
        # Method for producing an animation of the algorithm over all time-steps
        data = self.solution[2]
        X, Y, Z = self.contours()

        def animate(i, data, scatters):
            for j, point in enumerate(data[i]):
                scatters[j].set_offsets((point[0], point[1]))
            return scatters

        fig = plt.figure()
        ax = plt.axes(xlim=(bounds[0], bounds[1]), ylim=(bounds[0], bounds[1]))
        plt.xlabel('x1')
        plt.ylabel('x2')
        scatters = [ax.scatter(data[0][i][0], data[0][i][1], marker='x', color='black') for i in range(self.sw
        ax.contour(X,Y,Z)
        anim = animation.FuncAnimation(fig, animate, self.iterations, fargs=(data, scatters),
                interval=50, blit=False, repeat=True)
        # writer = animation.FFMpegWriter(fps=30, codec='libx264')
        # anim.save('ParticleSwarmAnimation.gif', writer=writer)
        plt.show()

    def progression_plot(self):
        #Method for plotting progression of particle locations at 4 time-steps

        steps = [0, 5, 10, 20]
        data = self.solution[2]
        X, Y, Z = self.contours()
        fig, axs = plt.subplots(2, 2)
```

```
        axs[0, 0].contour(X,Y,Z, zorder=-1)
        [axs[0, 0].scatter(data[steps[0]][i][0], data[steps[0]][i][1], marker='x', color='black', zorder=1) fo
        axs[0, 0].set_title('{} steps'.format(steps[0]))

        axs[0, 1].contour(X,Y,Z, zorder=-1)
        [axs[0, 1].scatter(data[steps[1]][i][0], data[steps[1]][i][1], marker='x', color='black', zorder=1) fo
        axs[0, 1].set_title('{} steps'.format(steps[1]))

        axs[1, 0].contour(X,Y,Z, zorder=-1)
        [axs[1, 0].scatter(data[steps[2]][i][0], data[steps[2]][i][1], marker='x', color='black', zorder=1) fo
        axs[1, 0].set_title('{} steps'.format(steps[2]))

        axs[1, 1].contour(X,Y,Z, zorder=-1)
        [axs[1, 1].scatter(data[steps[3]][i][0], data[steps[3]][i][1], marker='x', color='black', zorder=1) fo
        axs[1, 1].set_title('{} steps'.format(steps[3]))
        plt.show()


class Swarm:
    """Swarm class, to contain the list of particles contained within it"""

    def __init__(self, dimension, population_size, bounds, parameters, f, problem):
        #Initialise swarm and swarm best value
        self.swarm_best_cost = np.inf
        self.swarm = []
        for i in range(population_size):
            self.swarm.append(Particle(dimension, bounds, parameters, f, problem))

        #Set initial swarm best cost
        for particle in self.swarm:
            if particle.particle_best_cost < self.swarm_best_cost:
                self.swarm_best_x = particle.particle_best_x
                self.swarm_best_cost = particle.particle_best_cost

    def update(self, iteration, max_iterations):
        # Swarm update method
        # Neighborhood size for communication between particles, set to a large number for global communicatio
        neighborhood_size = 10

        #Update each particle in swarm and update local best solution of nearby particles
        for particle in self.swarm:
            particle.update(self.swarm_best_x, self.swarm_best_cost, iteration, max_iterations)
            for nearby_particle in self.swarm:
                if nearby_particle == particle:
                    continue
                if np.linalg.norm(nearby_particle.x - particle.x, 2) < neighborhood_size:
                    if nearby_particle.local_best_cost > particle.particle_best_cost:
                        nearby_particle.local_best_cost = particle.particle_best_cost
                        nearby_particle.local_best_x = particle.particle_best_x

        #Update swarm best cost
        for particle in self.swarm:
            if particle.particle_best_cost < self.swarm_best_cost:
                self.swarm_best_x = particle.particle_best_x
                self.swarm_best_cost = particle.particle_best_cost

class Particle:
    """Particle class, to hold information about particle's position, velocity, and known solutions""

    def __init__(self, dimension, bounds, parameters, f, problem):
```

```
        # Initialise particles state and known solutions based on problem

        self.bounds = bounds
        self.parameters = parameters
        self.inertia = self.parameters[0]
        self.x = np.random.uniform(low=bounds[0], high=bounds[1], size=(dimension,))
        self.cost = f(self.x)
        self.problem = problem
        self.velocity = np.random.uniform(low=bounds[0], high=bounds[1], size=(dimension,))
        self.particle_best_x = self.x
        self.particle_best_cost = self.cost
        self.local_best_x = self.x
        self.local_best_cost = self.cost

    def update(self, swarm_best_x, swarm_best_cost, iteration, max_iterations):
        # Update particle x and v, and force new x to be within bounds of problem
        for i in range(self.x.size):
            r_p = np.random.rand(1)
            r_g = np.random.rand(1)
            self.velocity[i] = self.parameters[0]*self.velocity[i] + self.parameters[1]*r_p*(self.particle_bes
        self.x = self.x + self.velocity
        self.x = np.clip(self.x, self.bounds[0], self.bounds[1])
        self.cost = self.problem.f(self.x)
        iteration = float(iteration)

        # Options for varying inertia weight with iteration number are below:
        # self.parameters[0] = 0.95*self.parameters[0]
        # self.parameters[0] = self.inertia + 0.1*self.inertia*np.cos(3*np.pi*iteration/max_iterations)
        # self.parameters[0] = (1-iteration/max_iterations)*self.inertia
        # self.parameters[0] = np.random.uniform(0.9*self.inertia, self.inertia*1.1)
        # self.parameters[0] = self.inertia + (self.inertia/3)*np.cos(np.pi*iteration/max_iterations)

        # Update best known costs
        if self.cost < self.particle_best_cost:
            self.particle_best_x = self.x
            self.particle_best_cost = self.cost

# Set random number seed
np.random.seed(0)

#Set Parameters
iterations = 200
omega = 0.683
phi_p = 0.605
phi_g = 0.358
swarm_size = 50
bounds = [-2,2]
dimension = 5

def parameter_problem(u):
    # Parameter problem for meta-optimisation or multiple random seed tests
    good = 0
    tests = 50
    total = 0
    for i in range(tests):
        np.random.seed(i)
        problem = Problem(iterations, swarm_size, u, bounds, dimension)
        solution = problem.solve()[1]
        if solution < -73.44:
            good += 1
```

```
        total += solution
    print(100*good/tests)
    return total/tests

# Run parameter problem to find average value obtained
print(parameter_problem(np.array([omega, phi_p, phi_g])))

# Meta-optimisation using scipy.optimise.minimise function
result = opt.minimize(parameter_problem,np.array([0.5, 0.5, 0.5]))
print(result.x)

# Simple problem solution with plotting
problem = Problem(iterations, swarm_size, [omega, phi_p, phi_g], bounds, dimension)
problem.solve(convergence_plot=True, animation=True, progression_plot=True)
```

References

[1] G. T. Parks. CUED 4M17 Course Notes.

[2] G. T. Parks. An intelligent stochastic optimization routine for nuclear fuel cycle design. *Nuclear Technology*, 89(2):233–246, 1990.

[3] L. B. Goldstein and M. S. Waterman. Neighborhood size in the simulated annealing algorithm, 1988.

[4] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, Nov 1995.

[5] Scipy python package, https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html.

[6] M. Zambrano-Bigiarini, M. Clerc, and R. Rojas. Standard particle swarm optimisation 2011 at cec-2013: A baseline for future pso improvements. In *2013 IEEE Congress on Evolutionary Computation*, pages 2337–2344, June 2013.

[7] J. C. Bansal, P. K. Singh, M. Saraswat, A. Verma, S. S. Jadon, and A. Abraham. Inertia weight strategies in particle swarm optimization. In *2011 Third World Congress on Nature and Biologically Inspired Computing*, pages 633–640, Oct 2011.

[8] M. Omran. *Using Opposition-based Learning with Particle Swarm Optimization and Barebones Differential Evolution*. 01 2009.