Optimisation Algorithms Week 8 Assignment

(a) Global random search

(i) Algorithm 1 goes through the main steps of the global random search. First, a range that is believed to contain the global minimum is picked. Then, a random vector in this range is sampled. In the case of x of length 2, such as the two functions observed here, this is a point on a plane. The function value in that point is computed. This is repeated a number of times and, at each step, if the new function value is lower than the previous minimum, the minimum is updated keeping track of the point or vector that corresponds to the minimum function value.

Algorithm 1 Global random search

for $k \in \overline{0, iterations}$ **do**
  $x \leftarrow$ random number within the range
  if $f(x) < f_{min}$ **then**
    $f_{min} \leftarrow f(x)$
  end if
end for

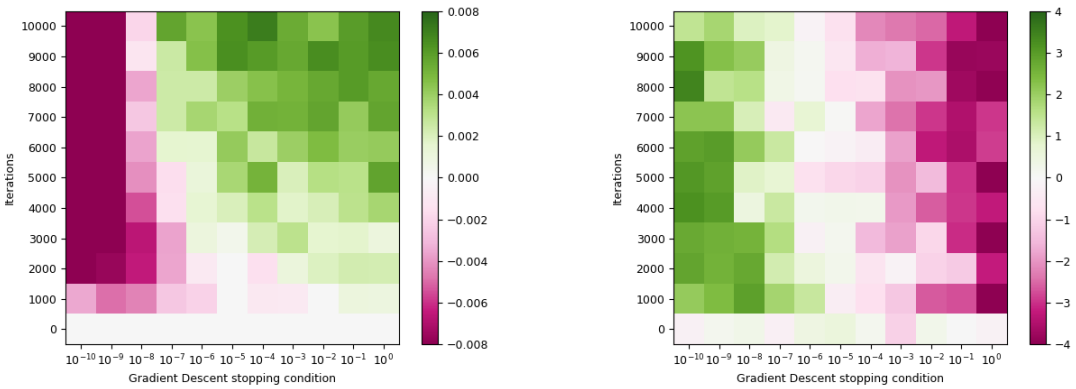
(ii) Figure 1 visually depicts the comparison between gradient descent and global random search when optimising the first function. The colour of the cells corresponds to the difference between the GRS measure and the GD measure. Figure a) shows the difference in running times, while figure b) shows the difference in function value, corresponding to the lowest function value encountered by each algorithm. Since minimisation is the desirable outcome for both of these measures, it follows that the lower (negative and higher in absolute value) the difference is, the more GRS outperforms GD for a specific combination of parameters, and the higher the difference, the better GD performs relatively to GRS. The colour scales have been manually adjusted to center on 0. Thus, cells are pink when GRS outperforms GD and green when the reverse hold, and the darker the colour the bigger the difference.

The number of iterations ranges from 0 to 10000 in increments of 1000. The set of stopping conditions is comprised of negative powers of 10 from 0 to -10. The difference in running times is measured in seconds, and the colour correspondence to different values is shown in the colour bar next to figure a). The difference in function values corresponds to the difference in the base 10 logarithm of the two function values. The Darkest shades of pink and green correspond to values that are, in absolute value, larger or equal to the ends of the depicted spectrum.

Each measure is computed for a combination of the number of iterations, which is the same for both algorithms, and the GD stopping condition. The stopping condition represent the difference allowance between two consecutive iteration function evaluations for which it is considered that the algorithm has converged and the iteration loop is broken. These parameters were chosen because I noticed when running the algorithms that they have a large impact on the outcome and I wanted to compare the two algorithms for a wide range of combinations of these parameters. The alpha parameter, or step size of the GD would also have an impact on the comparison, but for the purpose of these experiments it was kept equal to 0.01. The reasons for this are ease of visualisation, avoiding excessive plots, and the fact that the effects of alpha on gradient descent were already observed and discussed in past assignments.

By looking at figure 1 a) we can see that the running times of GD compared to GRS increase as the stopping condition becomes closer to 0. This is because a lenient stopping condition will allow GD to exit the iteration loop much sooner. As an effect, even though the maximum number of allowed iterations is the same for each algorithm, GD only ends up looping through a fraction of it. However, when very high accuracy is imposed, GD needs to loop through most or all of its iterations. A slightly less strong but still noticeable effect is that of the maximum number of iterations. For very few iterations, GRS ends up being faster than GD almost every time, but since GRS is forced to go through all of its iterations while GD can exit as soon as its decrease slows down enough, increasing the maximum number of iterations eventually forces GRS to take longer than GD. The crucial detail driving the difference in running times for different parameter combinations is the number of function calls. The GRS algorithm calls the function only once on every iteration. However, the GD algorithm uses a finite difference estimation of the function

gradient, which calls the function four times every time it is computed. Thus, since GD computes one gradient and one function value per iteration, it end up making five function calls per iteration. This is why if GR and GRS perform a similar number of iterations, due to strict GD stopping conditions or low iteration requirements on GRS, GRS will end up outperforming GD in terms of running time. However, GD can be quicker in some cases because, since it moves towards a minimum on each iteration, it only needs a few iterations to find satisfactory results.
Figure b) shows the same analysis but this time using the function value as a performance measure. This time, the biggest influence is carried by the GD stopping condition, with the number of iterations barely affecting the results. As the GD stopping condition becomes more strict, the function value found by it is closer to the global minimum. GRS needs a lot more iterations to randomly encounter values as close to the minimum as GD can find. When increasing the number of iterations from 1000 to 10000, the stopping condition at which the two algirithms have similar performances only decreses by one order of magintude from 10^{-5} to 10^{-6}
The two figures show that there is a tradeoff between accuracy and running time. However, they also show that for high numbers of iterations (6000+) and a moderately high stopping condition (10^-7), GD outperforms GRD. This does not, however, necessarily correspond to either of the two algorithms' best performing sets of parameters.

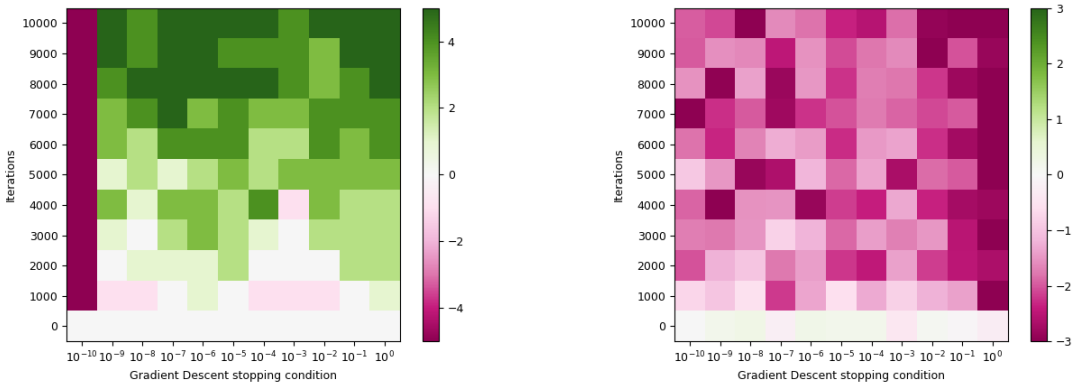


(a) Running time

(b) Function value (order of magnitude)

Figure 1: Difference in performance measures for the GD vs GRS on the first function for various parameter combinations

Figure 2 shows the same analysis as figure 1 but this time for the second function. Figure a) shows that GD achieves better running times for most parameter combinations, with the effect being stronger the more iterations GRS is forced to perform. For very few iterations, GRS manages to tie or even outperform, likely due to the same factors that are at play in the function 1 analysis. When GD is forced to iterate until its the function value change is lower than 10^{-10} it suddenly takes a very long time. One possible explanation is that due to the shape of the function near the minimum, GD oscillates around the minimum valley in such a way that never allows it to exit the iteration loop when this condition is imposed.
Figure b) shows the flip side of the coin - in spite of tahing longer to finish, GRS manages to get closer to the minimum for every non-zero number of iterations regardless of the GD stopping condition. Once again, it is likely that this is due to GD not managing to get close enough to the minimum and instead oscillating around it. This showcases one of the most useful applications of GRS - its ability to get close to the minimum is not limited in any way by the shape of the function, instead it is a matter of probability to randomly sample a point close enough to this minimum.



(a) Running time

(b) Function value

Figure 2: Difference in performance measures for the GD vs GRS on the second function for various parameter combinations

(b) Population search

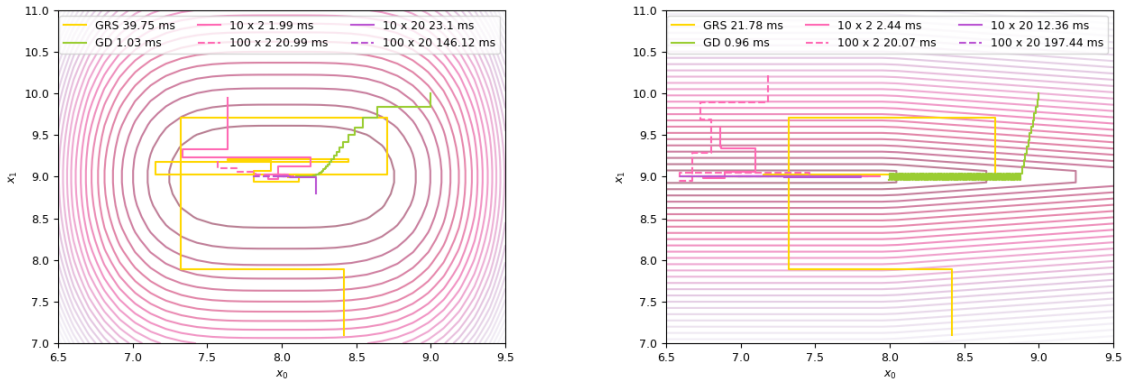
(i) Algorithm 2 shows the main steps of the population-based search. Just like for the global random search, a range believed to contain the global minimum is chosen. Then, N random vectors from this range are sampled. For each vector, M neighbours are randomly chosen. The neighbourhood range I am using is a square centered on the vector with a constant size set by default to 0.5 in each direction. I chose this approach because it is a simple but effective way of defining the neighbourhood. The code allows for the customisation of the neighbourhood range in order to experiment with the performance of the algorithm for different values. A neighbourhood that is too big would result in neighbours of points near the minimum overshooting and dropping out of the set, while reaching the minimum would be dependent on points far away from it randomly branching to a neighbour in the right direction. A neighbourhood that is too small would result in little improvement being made with each iteration. If all the initial points happen to be far away from the minimum, a small range would need many iteration to get close to it. The function value of the $N \times M$ vectors is computed and the N best points are kept. The algorithm is repeated a number of times. In my code I also included a stopping condition - if the smallest function value in the set remains unchanged for a number of iterations, the iteration loop breaks. The number of iterations after which an unchanged minimum causes the loop to end is customisable and set to 10 by default. The more iterations, the more likely it is that a point closer to the global minimum will be found. Thus, the more iterations are allowed to pass without a change, the closer the current minimum found is to the global minimum, and the more time it takes for the algorithm to run because not only it is allowed to run for more iterations without a change in the observed minimum, but every time a new minimum is found, the counter resets to 0.

Algorithm 2 Population-based search

$X \leftarrow$ N random vectors within the range
for $k \in \overline{0, iterations}$ **do**
  for $x_i \in X$ **do**
    $X_{neighbours} \leftarrow$ M random vectors within the neighbourhood of x_i
    $X \leftarrow X \cup X_{neighbours}$
  end for
  $X \leftarrow X_{\text{sorted by f}(x_i)}[: N]$
end for

(ii) Figure 3 shows the performance of GD, DRS, and population-based search with 4 parameter combinations. This is shown as a contour plot of the function, with added steps corresponding to the steps taken by each algorithm towards the minimum. For the GRS algorithm, each newly sampled vector that improves the function value is considered a step. For population-based search, a step is the vector corresponding to the best function value out of the M best vectors that are kept at each iteration. The GRS and GD are shown in yellow and green respectively. The PS are shown

in pink when two neighbours are sampled for each point, and purple when 20 neighbours are taken. The number of values kept at each iteration of the PS is encoded using the linestyle, with solid lines showing runs where the 10 best points are kept, and dashed lines when the 100 best points are kept.
In figure 3 a), showing the optimisation of the first function, GD behaves characteristically, starting in a predetermined point near the optimum and decreasing towards the optimum with decreasing steps. GRS starts in a random point in the allowed range (which corresponds to the depicted range), and then jumps to random points that are always closer to the minimum than the previous one but not necessarily in the same direction.
In figure 3 b), depicting function 2, we can see that the previous suspicions regarding the behaviour of the GD algorithm around the minimum valley are confirmed, with oscillations occurring until the algorithm runs out of iterations
PS has features resembling both GD and GRS. On the one hand, it moves in the same general direction towards the minimum, much like GD, because it relies on neighbour sampling instead of random sampling to find new points. On the other hand, this path is not as streamlined as GD, and the random sampling of neighbours can be observed in slight deviations and changes in the path.
The both figures, the two purple lines seem to show a straighter path towards the minimum. This could be due to the fact that when sampling more neighbours, it is more likely to find one that leads directly towards the minimum. The pink. Moreover, the start much closer to the minimum, likely because they start with more random points sampled within the range, which means they have a higher chance of finding a point close to the minimum right away. Visually, all algorithms seem to converge towards the minimum. Population-based search could get closer to it due to the higher number of sampled points compared to GRS, however this is conditioned by the initial random sampling, the neighbourhood size, the number of iterations, and the stopping condition.
In each case, GD outperforms all the other algorithms in terms of running time, finishing in 1 ms. Population-based search with small sets and few neighbours takes 2 - 3 times longer. GRS and population-based search with large sets & few neighbours or small sets & many neighbours are comparable, taking around 12-40 ms, and varying within this range between runs. Population search with large sets and many neighbours takes the longest: 150-200 ms. The driving force between the GRS running time is the fact that it is the only one without a stopping condition, being forced to call the function on each of the maximum iterations. Population-based search calls the function (N+1) x M times each iteration to sort and prune the sample. Thus, 10 x 20 and 100 x 2 will have similar running times. On occasion, the run with many neighbours might go a bit faster because neighbours generated outside the bounds are discarded. The second function is prone to this due to the minimum spanning an infinitely long valley.



(a) Function 1

(b) Function 2

Figure 3: Optimisation path of the three algorithms, with 4 parameter combinations shown for population-based search

(c) Both of the algorithms are given the same uppen and lower bound for each parameter. The batch size is between 32 and 256, the epochs are between 10 and 30, alpha is between 0.0001 and 0.001,

$beta_1$ and $beta_2$ are both between 0.2 and 0.9. The batch and epoch bounds are chosen based on the experiments ran for the week 8 assignment of the ML course. The Adam parameters are set based on common practice and the experiments ran in previous optimisation assignments. I am running GRS for N=40 iterations. I am running PS for 10 iterations with M = 2 and N = 2. A constant neighbourhood size wouldn't have worked for parameters that differ in their orders of magnitude. Moreover, the batch size and number of epochs must remain integer. Thus, the neighbourhood size is set to be one tenth of the difference between the upper and lower bound for each variable.
The population-based search algorithm finds an accuracy maximum of 0.515, while global random search finds a maximum of 0.3893. PS takes 400 seconds to run, while GRS takes 300 seconds. Thus, the increase in accuracy of PS is directly proportional with the increase in running time required. Whether this trade-off is worth making depends on the degree of specificity required as well as the available time and computational power.

Appendix A - Code

```
# function: 9*(x-8)^4+8*(y-9)^2
# function: Max(x-8,0)+8*|y-9|

import time
from math import *
import random
import numpy as np

import tensorflow as tf
from tensorflow import keras
from keras import layers, regularizers
from keras.optimizers import Adam
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
from sklearn.metrics import  accuracy_score
from sklearn.utils import shuffle

import matplotlib.pyplot as plt
from matplotlib import cm

rng = random.Random()
rng.seed(42)


def convolutions(x):
    batch_size, epochs, alpha, beta1, beta2 = x
    batch_size, epochs = int(batch_size), int(epochs)
    '''plt.rc('font', size=18)
    plt.rcParams['figure.constrained_layout.use'] = True'''
    import sys

    # Model / data parameters
    num_classes = 10
    input_shape = (32, 32, 3)

    # the data, split between train and test sets
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
    n = 5000
    x_train = x_train[1:n];
    y_train = y_train[1:n]

    with tf.device('/device:GPU:0'):
        # Scale images to the [0, 1] range
        x_train = x_train.astype("float32") / 255
        x_test = x_test.astype("float32") / 255
        #print("orig x_train shape:", x_train.shape)

        # convert class vectors to binary class matrices
        y_train = keras.utils.to_categorical(y_train, num_classes)
        y_test = keras.utils.to_categorical(y_test, num_classes)

        use_saved_model = False
        if use_saved_model:
            model = keras.models.load_model("cifar.model")
        else:
            model = keras.Sequential()
```

```
            model.add(Conv2D(16, (3, 3), padding='same', input_shape=x_train.shape[1:], activation='
            model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same', activation='relu'))
            model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
            model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same', activation='relu'))
            model.add(Dropout(0.5))
            model.add(Flatten())
            model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.l1(0.
            model.compile(loss="categorical_crossentropy",
                          optimizer=Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2), metrics=[
            model.summary()
            history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_s
            model.save("cifar.model")
            '''plt.subplot(211)
            plt.plot(history.history['accuracy'])
            plt.plot(history.history['val_accuracy'])
            plt.title('model accuracy')
            plt.ylabel('accuracy')
            plt.xlabel('epoch')
            plt.legend(['train', 'val'], loc='upper left')
            plt.subplot(212)
            plt.plot(history.history['loss'])
            plt.plot(history.history['val_loss'])
            plt.title('model loss')
            plt.ylabel('loss');
            plt.xlabel('epoch')
            plt.legend(['train', 'val'], loc='upper left')
            plt.show()'''
        preds = model.predict(x_test)
        y_pred = np.argmax(preds, axis=1)
        y_test1 = np.argmax(y_test, axis=1)
        accuracy = accuracy_score(y_test1, y_pred)
    return -accuracy

# (Un)comment to select function
def f(x):
    # y = max(x[0] - 8, 0) + 8 * abs(x[1] - 9)
    y = 9 * (x[0] - 8) ** 4 + 8 * (x[1] - 9) ** 2
    return y


def finite_difference(x, delta=10 ** (-5)):
    g0 = (f([x[0] + delta, x[1]]) - f([x[0] - delta, x[1]])) / (2 * delta)
    g1 = (f([x[0], x[1] + delta]) - f([x[0], x[1] - delta])) / (2 * delta)
    return np.array([g0, g1])


def global_random_search(l, u, N=4200, function=f):
    n = len(l)
    best_x = [rng.uniform(l[i], u[i]) for i in range(n)]
    best_f = function(best_x)
    # X0, X1 = [best_x[0]], [best_x[1]]
    X = [best_x]
    Y = [best_f]
    for k in range(N):
        x = [rng.uniform(l[i], u[i]) for i in range(n)]
        f_x = function(x)
        if f_x < best_f:
            best_f = f_x
```

```
            best_x = x
            # X0.append(x[0])
            # X1.append(x[1])
        X.append(best_x)
        Y.append(best_f)
        print(k, Y)
    return X, Y  # [X0, X1], Y


def population_search(l, u, iters=4200, M=10, N=2, neighbourhood=0.5, condition=50, function=f):
    n = len(l)
    x_sample = [[rng.uniform(l[i], u[i]) for i in range(n)] for j in range(N)]
    #X0, X1 = [], []
    X = []
    Y = []
    f_prev = 100
    change = 0
    for k in range(iters):
        x_neighbourhood = x_sample.copy()
        for x in x_sample:
            for m in range(N):
                neighbour = [x[i] + rng.uniform(-(u[i]-l[i])/10, (u[i]-l[i])/10) for i in range(n)]
                if l[0] < neighbour[0] < u[0] and l[1] < neighbour[1] < u[1]:
                    x_neighbourhood.append(neighbour)
        x_sample = sorted(x_neighbourhood, key=lambda x: function(x))[:M]
        f_x = function(x_sample[0])
        #X0.append(x_sample[0][0])
        #X1.append(x_sample[0][1])
        X.append(x_sample[0])
        Y.append(f_x)
        if f_prev == f_x:
            change = change + 1
        else:
            f_prev = f_x
            change = 0
        if change > condition:
            break
        print(k, Y)
    return X, Y#[X0, X1], Y


def gradient_descent(x0, alpha, condition=-5, iters=4200):
    x = np.array(x0)
    f_x = f(x)
    X0, X1 = [x[0]], [x[1]]
    Y = [f_x]
    for k in range(iters):
        gradient = finite_difference(x)
        step = alpha * gradient  # Default
        f_prev = f_x
        x = x - step
        f_x = f(x)
        X0.append(x[0])
        X1.append(x[1])
        Y.append(f_x)
        if abs(f_prev - f_x) < 10 ** (condition):
            break
    return [X0, X1], Y
```

```
def makeZ(X, Y):
    Z = np.zeros_like(X)
    for i in range(len(X)):
        for j in range(len(X[0])):
            Z[i, j] = f([X[i, j], Y[i, j]])
    return Z


def plot3D(X, Y, Z):
    plt.rcParams['font.size'] = 12
    norm = plt.Normalize(Z.min(), Z.max())
    colors = cm.PuRd_r(norm(Z))
    ax = plt.axes(projection='3d')
    ax.plot_surface(X, Y, Z, facecolors=colors, shade=False).set_facecolor((0, 0, 0, 0))
    ax.set_xlabel('$x_{0}$')
    ax.set_ylabel('$x_{1}$')
    ax.set_zlabel('f')
    plt.show()


def plotContour(X, Y, Z):
    plt.rcParams['font.size'] = 9
    fig, ax = plt.subplots(1, 1)
    ax.contour(X, Y, Z, levels=30, alpha=0.5, cmap='PuRd_r')
    tic = time.time_ns()
    steps, line = global_random_search([6.5, 7], [9.5, 11])
    toc = time.time_ns()
    ax.step(steps[0], steps[1], color='gold', label='GRS ' + str((toc - tic) // 1000 / 100) + ' ms')
    tic = time.time_ns()
    steps, line = gradient_descent([9, 10], 0.01, condition=-5)
    toc = time.time_ns()
    ax.step(steps[0], steps[1], color='yellowgreen', label='GD ' + str((toc - tic) // 10000 / 100) +
    colours = ['hotpink', 'mediumorchid']
    lines = ['-', '--']
    for n, N in enumerate([2, 20]):
        for m, M in enumerate([10, 100]):
            tic = time.time_ns()
            steps, line = population_search([6.5, 7], [9.5, 11], N=N, M=M)
            toc = time.time_ns()
            ax.step(steps[0], steps[1], color=colours[n], linestyle=lines[m],
                    label=str(M) + ' x ' + str(N) + ' ' + str((toc - tic) // 10000 / 100) + ' ms')
    ax.legend(ncol=3)
    ax.set_xlabel('$x_{0}$')
    ax.set_ylabel('$x_{1}$')
    plt.show()


def compare_gd_grs():
    clock = []
    function = []
    for N in range(10000, -1000, -1000):
        clock_row = []
        function_row = []
        for p in range(-10, 1, 1):
            tic = time.time_ns() / 1000000
            x_grs, y_grs = global_random_search([6.5, 7], [9.5, 11], N=N)
```

```
            toc = time.time_ns() / 1000000
            grs = (toc - tic)
            tic = time.time_ns() / 1000000
            x_gd, y_gd = gradient_descent([9, 10], 0.01, condition=p, iters=N)  # 0.01 f1, 0.02 f2
            toc = time.time_ns() / 1000000
            gd = (toc - tic)
            function_row.append(log10(y_grs[-1]) - log10(y_gd[-1]))
            # function_row.append((y_grs[-1] - y_gd[-1]))
            clock_row.append(grs - gd)
        clock.append(clock_row)
        function.append(function_row)
    for grid in [clock]:
        plt.rcParams['font.size'] = 9
        plt.imshow(grid, cmap='PiYG', vmin=-4, vmax=4)
        plt.xlabel('Gradient Descent stopping condition')
        tick_vals = range(0, 11)
        tick_labels = [r'$10^{%s}$' % i for i in range(-10, 1)]
        plt.xticks(ticks=tick_vals, labels=tick_labels)

        plt.ylabel('Iterations')
        tick_vals = range(0, 11)
        tick_labels = [i for i in range(10000, -1000, -1000)]
        plt.yticks(ticks=tick_vals, labels=tick_labels)

        plt.colorbar()
        plt.show()


x = np.linspace(6.5, 9.5, 30)
y = np.linspace(7, 11, 30)
X, Y = np.meshgrid(x, y)
Z = makeZ(X, Y)
# plot3D(X, Y, Z)
#plotContour(X, Y, Z)
# compare_gd_grs()

tic = time.time()
#print(global_random_search([32, 10, 0.0001, 0, 0], [256, 40, 0.5, 1, 1], N=40, function=convolutions
print(population_search([32, 10, 0.0001, 0, 0], [256, 40, 0.5, 1, 1], iters=10, M=2, N=2, condition=5
toc = time.time()
print(toc-tic)
```