Harikumar, Vaishnavi

Project 2 Part 1: Bayesian Inference and Decisions

Problem: flying robot with scent sensing capabilities

The problem considers a flying robot that consists of a sensor capable of sensing scent. While this sensor capability allows the robot to identify locations at which the scent is predominant as well as measure the value of the scent, it also requires high levels of power and can only function a limited number of times before having to stop. In this problem, we are considering a non-smooth 2-dimensional terrain where the ground is not structured normally. The 2-D terrain is normalized to [-1,1] x [-1, 1]. An object that diffuses a strong scent is buried underground on this terrain that is required to be located. The scent diffuses to different locations on the terrain and can get trapped in pockets where there are cavities. The robot can only obtain 12 sets of readings at a time which consists of the x-coordinate, y-coordinate, and the scent level. The problem asks to find the location where the object is buried using the 12 sets of readings received from the robot during one flight.

A. Introduction and approach to the problem

Introducing the idea to approach the problem:

The object buried in the defined 2-D space dissipates a scent that spreads over the terrain. The distribution of scent would be continuous with smooth troughs and crests rather than sharp peaks and dips due to the nature of diffusion of scents. The scent distribution over the terrain can be considered a gaussian. The mean of the gaussian represents the location at which the scent is at its peak which means the object is buried at the same location. This would have been a simple solution if the terrain was smooth and structured normally. However, the terrain is uneven and has small peaks and dips which causes cavities. These cavities hold pockets of the scent that is distributed over the terrain. I have considered each such location an individual gaussian distribution. But each of these distributions are dependent on each other since they are part of the same scent distribution. This makes it a Gaussian Process (GP). A Gaussian Process can be specified using a mean and covariance function. One idea that can be used to solve this problem is to find the location on the subject 2-D space where the mean of the Gaussian Process is maximum which would be the point where the scent is at its peak. Another idea is to find the location where the variance of the Gaussian Process is minimum. The point that has minimum variance is the point where the error in reading can be considered the least. The confidence that the sensor reading at this location is "correct" or as close to the true location where the object is buried, is high. This is the method I used for this problem.

The execution of the idea:

I have used Gaussian Process Regression to solve this problem. The definition of a Gaussian Process was adopted from "Definition 12.2" from LectureNotes2019. A Gaussian Process can be completely specified using a mean and covariance function. Mathematically, these two functions can be written as follows:

$$m(x) = E[f(x)]$$

$$k(x, x') = E[f(x) - m(x))(f(x') - m(x'))]$$

Where, $m(x)$ represents the expectation or mean of the Gaussian Process at any given $x \in X$, x being an element of any random variable X. And $k(x, x')$ represents the covariance of the values of the Gaussian Process at any two points $x, x' \in X$.

For an input/output pair, where $\bar{x} = [x^1, \ldots\ldots, x^n]$ are the inputs and $\bar{y} = [y^1, \ldots\ldots, y^n]$ are the outputs, and x^* is the location that needs to be predicted, the mean and variance of the Gaussian Process is defined as follows:

$$E[f(x^*)|\bar{y}\,] = m(x^*) + k(x^*, \bar{x})(k(\bar{x}, \bar{x}) + \sigma^2 I_{n\times n})^{-1}\big(\bar{y} - m(\bar{x})\big)$$

$$Var[f(x^*)|\bar{y}\,] = k(x^*, x^*) + k(x^*, \bar{x})(k(\bar{x}, \bar{x}) + \sigma^2 I_{n\times n})^{-1}k(\bar{x}, x^*)$$

In this problem, we have two inputs, the location $(\bar{x}, \bar{y})$ and one output value from the sensor that can be considered as $\bar{z}$. σ^2 is the variance of the Gaussian Process. For the subject problem to find a location (x^*, y^*), the equations for mean and variance of the Gaussian Process can be re-written as follows:

$$E[f(x^*, y^*)|\bar{z}\,] = m(x^*, y^*) + k(x^*, y^*, \bar{x}, \bar{y})(k(\bar{x}, \bar{y}, \bar{x}, \bar{y}) + \sigma^2 I_{n\times n})^{-1}\big(\bar{z} - m(\bar{x}, \bar{y})\big) \quad \textbf{(1)}$$

$$Var[f(x^*, y^*)|\bar{z}\,] = k(x^*, y^*, x^*, y^*) + k(x^*, y^*, \bar{x}, \bar{y})(k(\bar{x}, \bar{y}, \bar{x}, \bar{y}) + \sigma^2 I_{n\times n})^{-1}k(\bar{x}, \bar{y}, x^*, y^*) \quad \textbf{(2)}$$

These are the equations that I used to construct the functions in the code to find the mean and variance of the Gaussian Process. Furthermore, to find the required location, (x^*, y^*), I ran an optimizer on the equation for variance.

B. Description of code and method used

Problem setup:

First, I created a mesh grid of points which represents the predicted values in x-coordinate and y-coordinate. Each value in the grid is of the range [-1,1] x [-1,1], sliced into 50 divisions. In the code, the x-coordinate and y-coordinate of the grid originally has a dimension of 50 x 50. I reshaped both coordinates into 2500 x 1 since the functions used in the code take inputs as column vectors.

Next, I defined two arrays with the known locations of x and y coordinates, and a third array containing the sensor output values. These arrays get updated each time I find a new set of known locations after optimization. I have explained this process in detail in the upcoming sections. The first 10 set of values for all three were taken from the prompt of the given problem.

Then, I defined a prior mean. This is an array of size the same as the number of known locations. All elements are set to the mean value of the known sensor values. The prior mean also gets

updated each time I receive sensor data using the obtained locations from the application provided by the professor.

Method:

Gaussian Process regression are of two types: parametric and non-parametric. Here, I have used a non-parametric Gaussian Process regression. The steps to perform a non-parametric Gaussian Process regression is described in detail as follows:

1. **Choosing Hyperparameters:** There are three methods stated in the lecture notes to choose hyperparameters; Hierarchical Bayes, Cross-Validation, and Maximizing Marginal Likelihood. The first two methods would work better when we have a significant amount of data. As stated in "LectureNotes2019", page 13-2, under section 13.4, the method of maximizing the marginal likelihood is best suitable for a low data regime. Here we are initially only given 10 sets of data points and are requesting another 12 from the sensor data application. For the number of datapoints we are working with, maximizing the marginal likelihood would give the best results because the code mathematically optimizes the mathematical expression giving me the most optimal hyperparameters. Tuning these hyperparameters each time a new set of datapoints is added to the original data, will also be far more yielding because this process is highly methodical, as opposed to the process used in the first two methods where we choose hyperparameters by trial and error. To find the optimal values of hyperparameters, I minimized the following expression:

$$\log p(\bar{z}\,|\bar{x},\bar{y};\theta) = -\frac{1}{2}(\bar{z} - m(\bar{x},\bar{y}))^T(k(\bar{x},\bar{y},\bar{x},\bar{y}) + \sigma^2 I)^{-1}(\bar{z} - m(\bar{x},\bar{y})) - \frac{1}{2}\log|k(\bar{x},\bar{y},\bar{x},\bar{y}) + \sigma^2 I| - \frac{n}{2} log\ 2\pi \quad (3)$$

Here, $\log p(\bar{z}\,|\bar{x},\bar{y};\theta)$ represents the log of the marginal likelihood, $m(\bar{x},\bar{y})$ represents the prior mean and n is the total number of known datapoints. All other terms have been described in the previous section.
For this process, I used the functions given in the jupyter notebook file "Bayesian non-parametric gaussian process regression". I made changes to the 1-D input functions into 2-D input functions since we have two sets of inputs. The functions I used from the notes are, sqexp () and build_covariance (). I also constructed two more functions for this process, max_marg_likelihood () and var_from_trace (). The use and description of these functions are as follows:

- The square exponential kernel: sqexp ()
  The use of this function is to calculate the covariance between two given sets of points. This function accepts as parameters, an array of the known locations in x-coordinate and y-coordinate, the predicted x-coordinate, and y-coordinate values, and two hyperparameters, τ and L. It returns the covariance of two given inputs.The square exponential kernel is an appropriate choice to model smooth functions. The reason I am using this kernel is because of the nature of the distribution of scent. I

am assuming that a scent distribution is smooth and continuous due to the nature of diffusion of scent. It cannot be a periodic function with peaks and dips. So, a square exponential kernel is chosen here over a periodic kernel. The mathematical expression I used to model this function was taken from the same jupyter notebook file as previously mentioned. To construct a 2-D kernel, I used the following expression:

$$k((\bar{x},\bar{y})(x^*,y^*)) = \tau_1 \exp\left(-\frac{12(\bar{x}-x^*)^2}{L_1^2}\right)\tau_2 \exp\left(-\frac{12(\bar{y}-y^*)^2}{L_1^2}\right) \quad \textbf{(4)}$$

Here, there are two parameters each for the x and y inputs $(\tau_1, L_1, \tau_2, L_2)$ in this expression. However, since I'm dividing the predicted locations in both coordinates by the same step-size, I have assumed that the value for each of the hyperparameters for both coordinates are the same i.e., $\tau_1 = \tau_2 = \tau$ and $L_1 = L_2 = L$.

- The build covariance function: build_covarance ()
  This function is used to store all the covariances calculated using the sqexp () function in a matrix. It accepts as parameters, an array of the known locations in x-coordinate and y-coordinate, the predicted x-coordinate, and y-coordinate values, the function sqexp (), and two of the hyperparameters τ and L. It returns the covariance matrix for the given set of input values.

- The marginal likelihood function: max_marg_likelihood ()
  This function is used to optimize the three hyperparameters. It takes τ, L, σ^2 as three variables that need to be optimized. Inside this function, I call the build_covariance () function to obtain a covariance matrix for the required set of inputs. Using the covariance matrix, I construct an equation for the marginal likelihood equation as specified in the choosing hyperparameters section. This function uses the marginal likelihood equation as the objective function to optimize the hyperparameters τ, L, σ^2. I used the inbuilt function scipy.optimize.minimize () to perform the optimization. The optimal values for the hyperparameters are displayed as results from this function.

2. Gaussian Process Regression:

Once the hyperparameters are found, I used Gaussian Process Regression to obtain the predicted mean and predicted covariance. For this process, I used the function gpr () from the jupyter notebook "Bayesian non-parametric gaussian process regression".

I modified the function to incorporate two input values. In this function I use equations (1) and (2) specified in this report to find the predicted mean and predicted variance. This function accepts as parameters, an array of the known locations in x-coordinate and y-coordinate, the known sensor values, the predicted x-coordinate, and y-coordinate values, the prior mean, the function sqexp (), and all three of the hyperparameters, τ, L, σ^2. It returns the predicted mean and predicted variance values.

3. **Finding new locations to request sensor data:**
   I came across three different algorithms that can be used to find new locations, the points at which can be used to request for sensor data from the application. They are briefly described as follows:

   I. The first algorithm is choosing points at the locations where the predicted mean of the Gaussian Process, obtained from gpr (), is the highest. The mean distribution represents the scent distribution, which means the points at which the mean is comparatively high, are the possible locations where the object producing the scent is buried. Plotting the mean distribution curve can help identify the points at which the mean tends to be high. Potential points can be chosen by eyeballing the plot and these points can be used to request sensor data. Using the new set of locations and their corresponding sensor data, we can repeat the process of choosing hyperparameters and then performing Gaussian Process Regression. However, this method seems less promising since the locations are chosen manually from the graph.

   II. The second algorithm is similar to the first algorithm. But, instead of eyeballing values at the highest means, we would have to look at locations where the variance is highest. Choosing points at which the variance is highest and obtaining the sensor data values at those location would help in significantly reducing the overall variance of the Gaussian Process because uncertainty at these locations have decreased. This process can be repeated for the number of times that we can request for sensor data from the application. This algorithm has the same disadvantage as the first one since the values are manually chosen and may not be accurate.

   III. **Method used:** I used a third algorithm for finding the appropriate locations. In this algorithm, first I used the predicted covariance matrix obtained from gpr () and found the trace of the matrix which is the sum of the diagonal elements. The diagonal elements of a covariance matrix represent the covariance of a point with itself, which is also equal to the variance at that point. Therefore, the sum of the diagonal elements essentially represents the overall variance of the Gaussian Process. Now, since the goal of the problem is to reduce the overall variance of the Gaussian Process, we can minimize this variance in order to find the optimal locations at which we can request sensor data. These locations would lie in the regions where the variance is the highest because the optimizer chooses points that would help the model. If we obtain the sensor data at which the variance is highest and incorporate it into the map we are trying to construct, the overall variance of the Gaussian Process decreases. I constructed another function var_from_trace () to perform this optimization. This function takes 8 design variables $(x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4)$ which represent the four sets of locations that need to be found. (x_1, x_2, x_3, x_4) are added at the end of the known x-coordinate values and (y_1, y_2, y_3, y_4) are added at the end of the known y-coordinate values. These sets

of data, along with the predicted x-coordinates and y-coordinates as well as the already known sensor data, uses gpr() to compute the predicted covariance, which is used as the objective function in this optimization process.
Again, to optimize I used scipy.optimize.minimize() to minimize the objective function (variance of GP) which gives the four location predictions. Using these locations, I then request for sensor data from the application.

These three steps provide the algorithm for performing Gaussian Process Regression for a given set of data. I repeated these steps for three different iterations to obtain 12 pairs of locations at which the sensor data can be requested. I also ran a fourth and final iteration to obtain a final curve fit of all the known locations and their corresponding sensor data.

Note 1: An important point to note in this algorithm is the fact that the set of locations and their corresponding sensor data obtained at the end of each iteration must be incorporated into the known values of the locations and their sensor data before the start of the next iteration. This means, we obtain a new set of hyperparameters at each iteration, which is used to run the functions gpr () and var_from_trace (). In turn we get a new set of locations from the optimizer.

Note 2: Another important point worth mentioning is that for any reasonable prior value, the posterior or predicted mean should remain the same. I first set the prior mean to an array of zeros with size same as that of the number of known sensor values. However, when I ran the optimizer the hyperparameters that resulted did not provide a good curve fit. In fact, the mean distribution was almost flat. This meant that the optimizer did not converge correctly for the chosen prior. Then I tried a different prior where the mean was an array of the same size as that of the number of known values, and the value at all positions was the arithmetic mean of all the known sensor values. This prior mean provided far better results for the hyperparameters and in turn better new locations. The predicted mean distribution when plotted showed a smooth curve with crests and troughs. As the number of known values increased at each iteration, the mean distribution showed higher peaks. Choosing the second prior makes the most sense because we are setting the prior to a value calculated from the known values. This gives the optimizer a head-start at finding the optimal hyperparameters and locations.

In the next section, I have specified the process of execution of this code and provided the results and plots after each iteration.

C. Execution and Results

Iteration 1

The set of known values are:

$$x^* = [0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9]$$

$$y^* = [0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9]$$

$$Sensor\ Output = [3.39382006, 3.2073034, 3.39965035, 3.68810201, 2.96941623, 2.99495501, 3.94274928, 2.7968011, 3.34929734, 3.91296165]$$

I also created a mesh-grid of values ranging from [-1,1] x [-1,1] sliced into 50 divisions. These are the predicted x and y coordinates, which remain the same for each iteration.

Using these values, I found the hyperparameters using optimization. I used different sets of initial points to run the optimizer. Note that the objective function used here is the marginal likelihood function. Also, I set the bounds for each hyperparameter to $[(0, None), (0, None), (0, None)]$. The optimized results have been tabulated below:

Table 1: Choosing Hyperparameters for Iteration 1

Initial Points	$\boldsymbol{\tau}$	$\boldsymbol{L}$	$\boldsymbol{\sigma^2}$
$(\tau_0 = 0.8, L_0 = 0.5, \sigma_0^2 = 0.01)$	0.6156	0.2641	$1.02 * 10^{-6}$
$(\tau_0 = 0.7, L_0 = 0.4, \sigma_0^2 = 0.01)$	0.6156	0.2641	0
$(\tau_0 = 0.95, L_0 = 0.5, \sigma_0^2 = 0.01)$	0.2766	0.1685	0.3699
$(\tau_0 = 0.6, L_0 = 0.45, \sigma_0^2 = 0.01)$	0.6156	0.2641	0

The best values obtained for the hyperparameters are shown in the first row of Table 1 which has been boxed in red. I chose the set of values that had the lowest non-zero variance since we are ultimately trying to decrease the overall variance of the Gaussian Process.

Using the hyperparameters, I ran the Gaussian Process Regression and obtained the predicted mean and predicted covariance. Figure 1 on the next page, shows the plot of the mean distribution obtained from the predicted mean, over the mesh-grid. The figure also shows the band of uncertainty above and below the mean distribution curve.

After running gpr (), I performed the next optimization to find new locations. The objective function used here is the variance obtained from var_from_trace (). The array of known locations for both x and y coordinates need to be updated here as such:

$$x^* = [0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9, x_1, x_2, x_3, x_4]$$

$$y^* = [0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9, y_1, y_2, y_3, y_4]$$

Here, the design variables are $(x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4)$ and the variance is being optimized. For this optimization, I used only one set of initial points, $(0, 0, 0, 0, 0, 0, 0, 0)$. I set the bounds to $((-1,1), (-1,1), (-1,1), (-1,1), (-1,1), (-1,1), (-1,1), (-1,1))$ for all 8 design variables.

The optimizer gave a set of 4 locations. I used these locations to request sensor data values from the robot sensor application which provided 4 scent values at each input location. The optimized results obtained, and corresponding sensor data are tabulated in Table 2 as shown below:

Table 2: Finding new locations for Iteration 1

Location Number	Location Values	Sensor Data
(x_1, y_1)	$(-0.771, -0.256)$	3.93840588
(x_2, y_2)	$(-0.264, -0.236)$	3.27396201
(x_3, y_3)	$(-0.474, -0.704)$	3.45780891
(x_4, y_4)	$(0.035, 0.793)$	2.59370420

The sensor data obtained was added at the end of the array of known sensor data for the next iteration. This is necessary to find the next set of hyperparameters and correspondingly the next set of known locations.

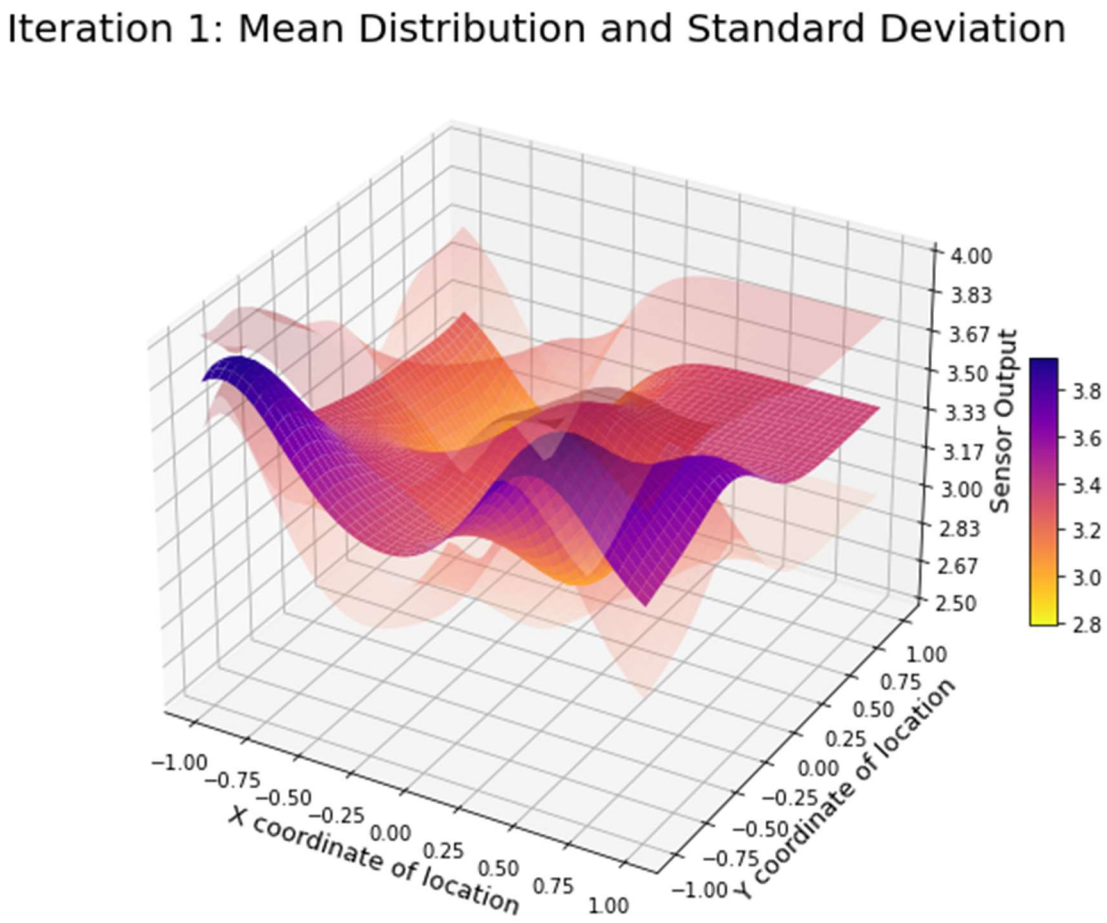


Figure 1: Mean distribution and standard deviation for the first iteration

In Figure 1, the curve fit in the middle of solid color represents the mean distribution obtained from gpr(). This curve fit was created from 10 known locations and their corresponding sensor data. The two curve fits above and below the mean distribution of a lower gradient, represents the band of uncertainty in the obtained values. Depending on the number of known data, this curve fit changes. In the later iterations, the map is modified since the number of known data values increase.

Figure 2 represents the mesh of the variance of the Gaussian Process and the black points represents the locations provided by the optimizer. In the figure, the pink region represents areas of high variance, and the yellow regions represent areas of lower variance. It is evident from this figure that the locations obtained from optimization lie in the regions of higher variance. This is beneficial because when these locations and their corresponding sensor values are added to the array of known locations for the next iteration, the variance at these locations decreases, thereby decreasing the overall variance of the Gaussian Process.

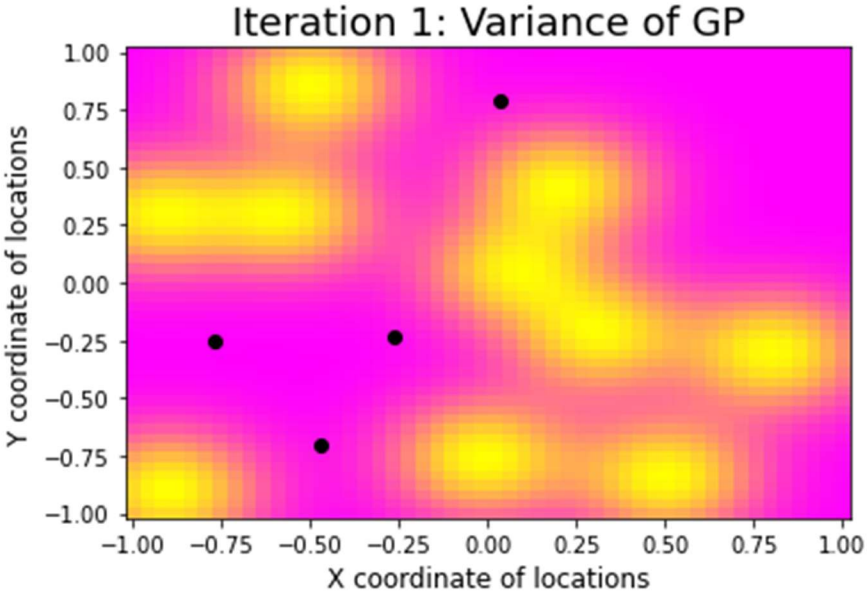


Figure 2: Variance of GP and new optimal locations for iteration 2

Iteration 2

The set of known values are:

$$x^* = [0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9, -0.771, -0.264, -0.474, 0.035]$$

$$y^* = [0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9, -0.256, -0.236, -0.704, 0.793]$$

$$Sensor\ Output = [3.39382006, 3.2073034, 3.39965035, 3.68810201, 2.96941623, 2.99495501, 3.94274928, 2.7968011, 3.34929734, 3.91296165, 3.39382006, 3.2073034, 3.39965035, 3.68810201]$$

The predicted x and y coordinates, remain the same for this iteration: values ranging from [-1,1] x [-1,1] sliced into 50 divisions

Using these values, I found a new set of hyperparameters using optimization. In this iteration I used the hyperparameters obtained from the previous iteration as the initial point. The objective function used here is the same as the last iteration, the marginal likelihood function and so are the bounds, $[(0, None), (0, None), (0, None)]$. The optimized results have been tabulated below:

Table 3: Choosing Hyperparameters for Iteration 2

Initial Points	$\boldsymbol{\tau}$	$\boldsymbol{L}$	$\boldsymbol{\sigma^2}$
$(\tau_0 = 0.6156, L_0 = 0.2641, \sigma_0^2 = 1.02 * 10^{-6})$	0.5738	0.2104	$1.02 * 10^{-6}$

Using the new hyperparameters, I ran the Gaussian Process Regression again and obtained the predicted mean and predicted covariance. Figure 3 on the next page, shows the plot of the mean distribution obtained from the predicted mean, over the mesh-grid. The figure also shows the band of uncertainty above and below the mean distribution curve.

After running gpr (), I performed the next optimization to find new locations. The objective function used here is the variance obtained from var_from_trace (). The array of known locations for both x and y coordinates need to be updated here as such:

$$x^* = [0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9, \\ -0.771, -0.264, -0.474, 0.035, x_1, x_2, x_3, x_4]$$

$$y^* = [0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, \\ -0.9, -0.256, -0.236, -0.704, 0.793, y_1, y_2, y_3, y_4]$$

Here, the design variables are $(x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4)$ and the variance is being optimized. For this optimization, I used only one set of initial points, $(0, 0, 0, 0, 0, 0, 0, 0)$. I set the bounds to $((-1,1), (-1,1), (-1,1), (-1,1), (-1,1), (-1,1), (-1,1), (-1,1))$ for all 8 design variables.

The optimizer gave a set of 4 locations. I used these locations to request sensor data values from the robot sensor application which provided 4 scent values at each input location. The optimized results obtained, and corresponding sensor data are tabulated in Table 4 as shown below:

Table 4: Finding new locations for Iteration 2

Location Number	**Location Values**	**Sensor Data**
(x_5, y_5)	$(0.835, -0.726)$	4.08231430
(x_6, y_6)	$(-0.814, -0.551)$	3.13022618
(x_7, y_7)	$(-0.830, 0.722)$	3.19713096
(x_8, y_8)	$(0.682, 0.633)$	3.28089254

The sensor data obtained was added at the end of the array of known sensor data for the next iteration. This is necessary to find the next set of hyperparameters and correspondingly the next set of known locations.

In Figure 3, the curve fit in the middle of solid color represents the mean distribution obtained from gpr(). This curve fit was created from 14 known locations and their corresponding sensor data. The two curve fits above and below the mean distribution of a lower gradient, represents the band of uncertainty in the obtained values. It is evident from the figure that the curve fit has created a better map showing higher crests and lower troughs. This essentially means that the accuracy of the model has increased and correspondingly the confidence in the model has also increased.

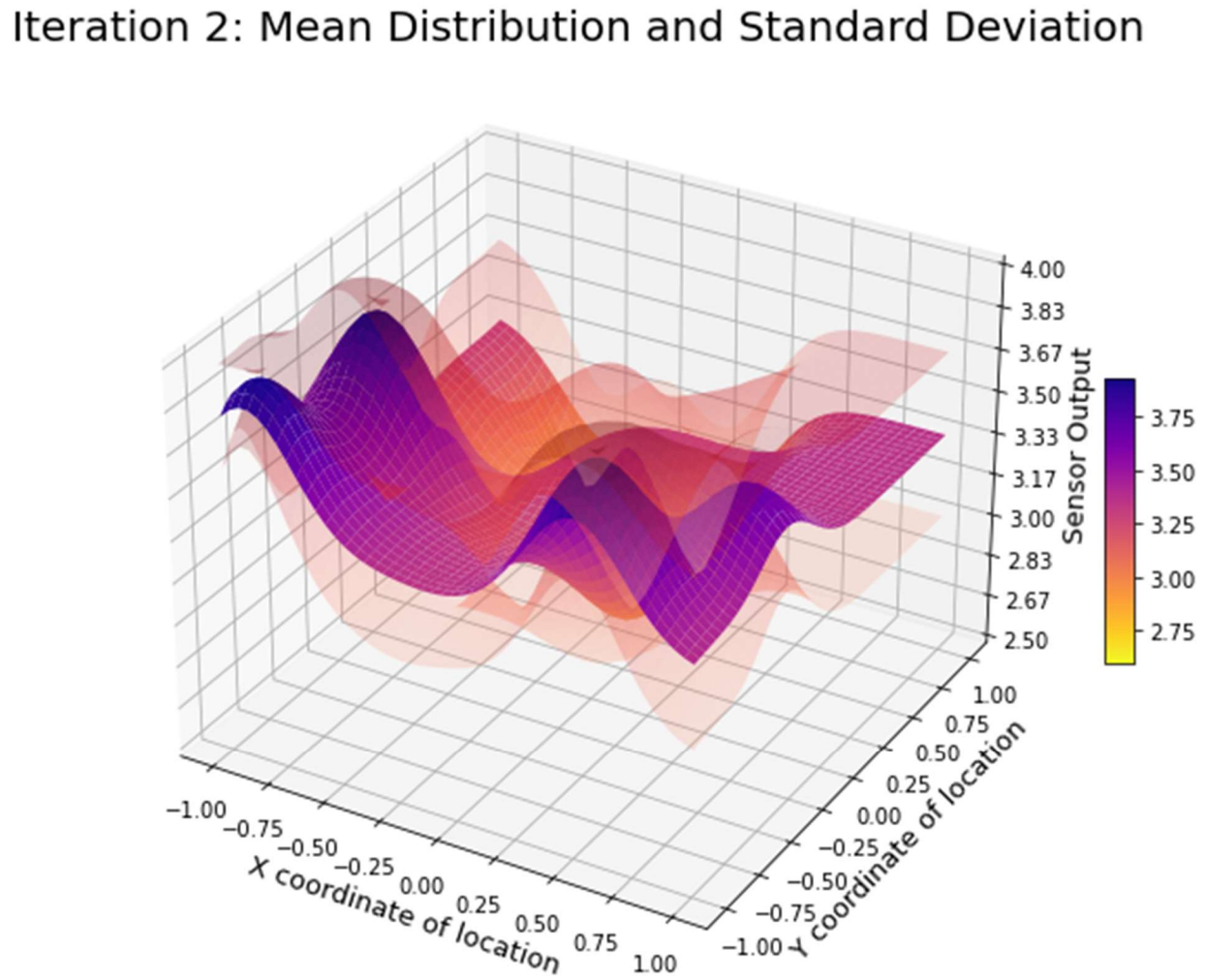


Figure 3: Mean distribution and standard deviation for the second iteration

Figure 4 represents the mesh of the variance of the Gaussian Process similar to Figure 2. The black points represent the new locations provided by the optimizer. It is evident from this figure that the locations obtained from optimization for this iteration, also lie in the regions of higher variance. I added these points to the arrays of known locations for the next iteration.

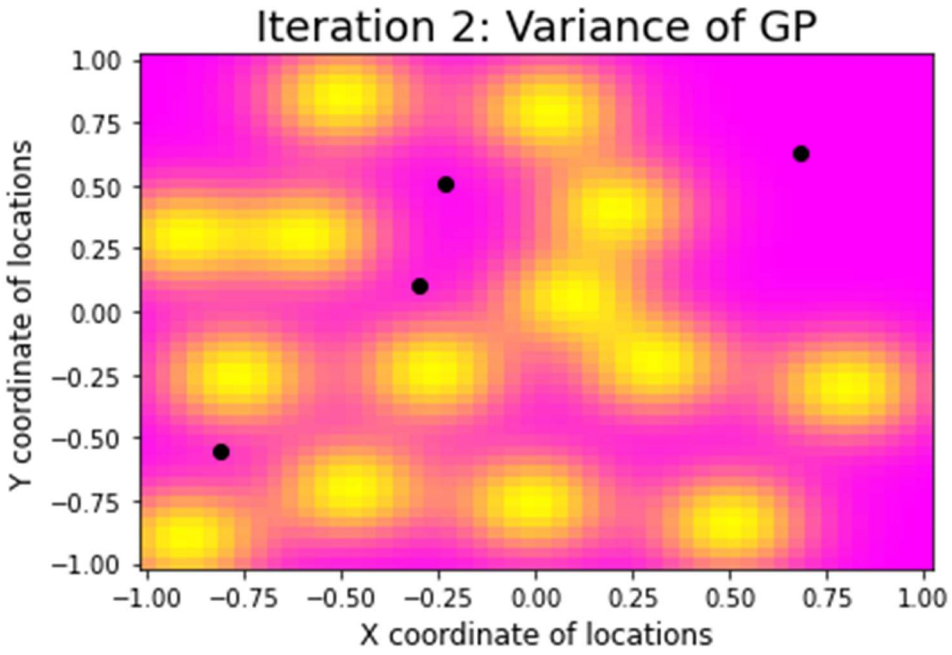


Figure 4: Variance of GP and new optimal locations for Iteration 2

Iteration 3

The set of known values are:

$$x^* = [0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9, -0.771, -0.264, -0.474, 0.035, 0.835, -0.814, -0.830, 0.682]$$

$$y^* = [0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9, -0.256, -0.236, -0.704, 0.793, -0.726, -0.551, 0.722, 0.633]$$

$$Sensor\ Output = [3.39382006, 3.2073034, 3.39965035, 3.68810201, 2.96941623, 2.99495501, 3.94274928, 2.7968011, 3.34929734, 3.91296165, 3.39382006, 3.2073034, 3.39965035, 3.68810201, 4.08231430, 3.13022618, 3.19713096, 3.28089254]$$

The predicted x and y coordinates, remain the same for this iteration: values ranging from [-1,1] x [-1,1] sliced into 50 divisions

Using these values, I found a third new set of hyperparameters using optimization. In this iteration I used the hyperparameters obtained from the previous iteration as the initial point. The objective function used here is the same as the last iteration, the marginal likelihood function and so are the bounds, $[(0, None), (0, None), (0, None)]$. The optimized results have been tabulated below:

Table 5: Choosing Hyperparameters for Iteration 3

Initial Points	τ	L	σ^2
$(\tau_0 = 0.5738, L_0 = 0.2104, \sigma_0^2 = 1.02 * 10^{-6})$	0.5798	0.2244	$1.06 * 10^{-6}$

Using the new hyperparameters, I ran the Gaussian Process Regression again and obtained the predicted mean and predicted covariance. Figure 3 on the next page, shows the plot of the mean distribution obtained from the predicted mean, over the mesh-grid. The figure also shows the band of uncertainty above and below the mean distribution curve.

After running gpr (), I performed the next optimization to find new locations. The objective function used here is the variance obtained from var_from_trace (). The array of known locations for both x and y coordinates need to be updated here as such:

$$x^* = [0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9, \\ -0.771, -0.264, -0.474, 0.035, x_1, x_2, x_3, x_4]$$

$$y^* = [0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, \\ -0.9, -0.256, -0.236, -0.704, 0.793, y_1, y_2, y_3, y_4]$$

Here, the design variables are $(x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4)$ and the variance is being optimized. For this optimization, I used only one set of initial points, $(0, 0, 0, 0, 0, 0, 0, 0)$. I set the bounds to $((-1,1), (-1,1), (-1,1), (-1,1), (-1,1), (-1,1), (-1,1), (-1,1))$ for all 8 design variables.

The optimizer gave a set of 4 locations. I used these locations to request sensor data values from the robot sensor application which provided 4 scent values at each input location. The optimized results obtained, and corresponding sensor data are tabulated in Table 4 as shown below:

Table 4: Finding new locations for Iteration 2

Location Number	Location Values	Sensor Data
(x_9, y_9)	$(0.835, -0.726)$	4.08231430
(x_{10}, y_{10})	$(-0.814, -0.551)$	3.13022618
(x_{11}, y_{11})	$(-0.830, 0.722)$	3.19713096
(x_{12}, y_{12})	$(0.682, 0.633)$	3.28089254

The sensor data obtained was added at the end of the array of known sensor data for the next iteration. This is necessary to find the next set of hyperparameters and correspondingly the next set of known locations.

In Figure 5, the curve fit in the middle of solid color represents the mean distribution obtained from gpr(). This curve fit was created from 18 known locations and their corresponding sensor data. The two curve fits above and below the mean distribution of a lower gradient, represents the band of uncertainty in the obtained values. This curve fit has created an even better map than the one in the second iteration. Now we can see significant peaks and valleys that are continuous. This means that we are getting closer to the answer. In the final curve fit, the location at which the

predicted mean is at its highest would be the location at which the object emitting the scent is buried. To obtain the final curve fit, I would have to run one more iteration of gpr () to incorporate the last four sets of locations into the model.

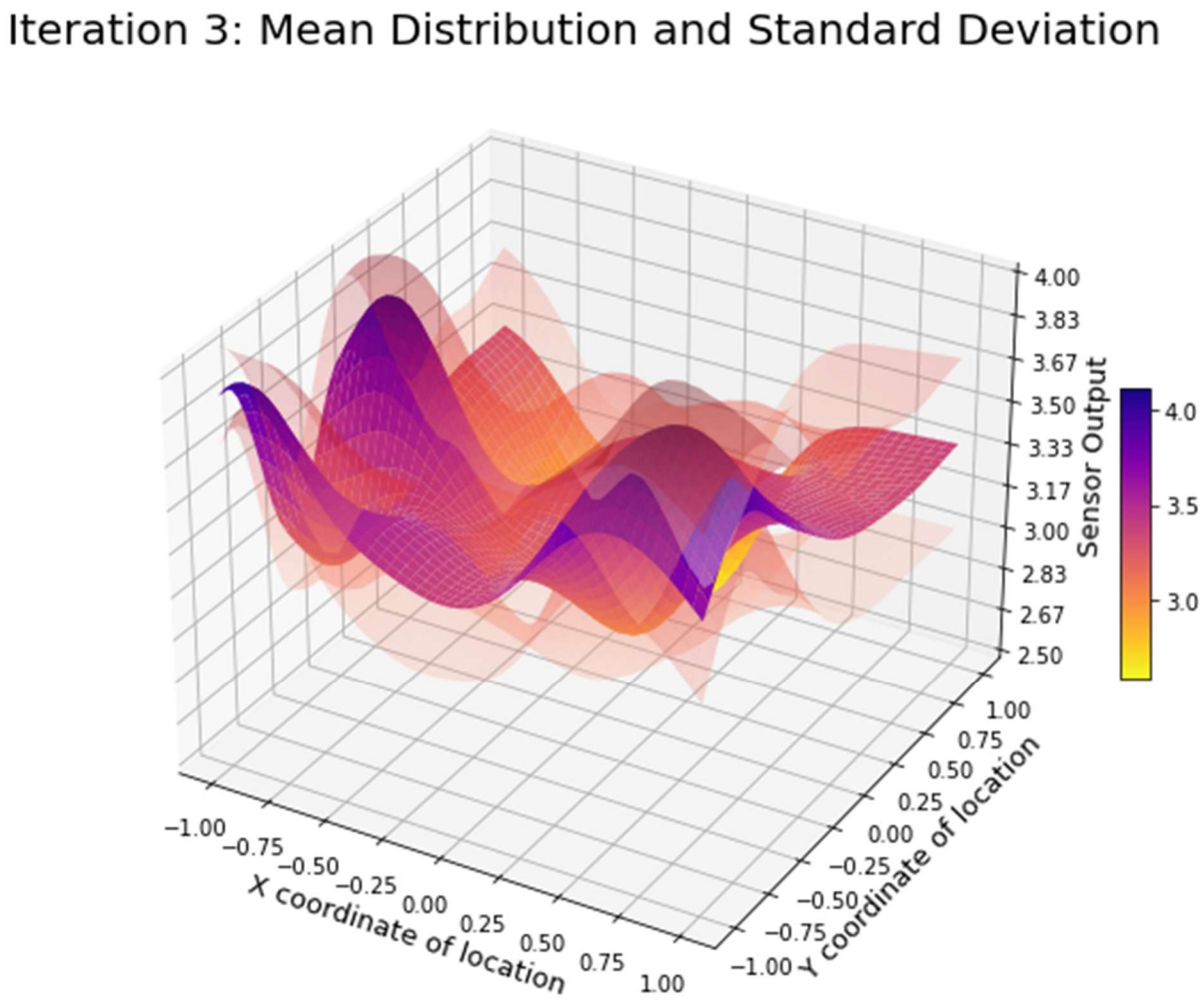


Figure 5: Mean distribution and standard deviation for the second iteration

Figure 6 represents the mesh of the variance of the Gaussian Process similar to Figure 2 and Figure 4. The black points represent the new locations provided by the optimizer. It is evident from this figure that the locations obtained from optimization for this iteration, also lie in the regions of higher variance.

A point worth noting here is the fact that at the locations that were found in the previous iterations, the color of the variance has changed from pink to yellow. This means that the variance at these locations have decreased since the sensor values at these locations are known.

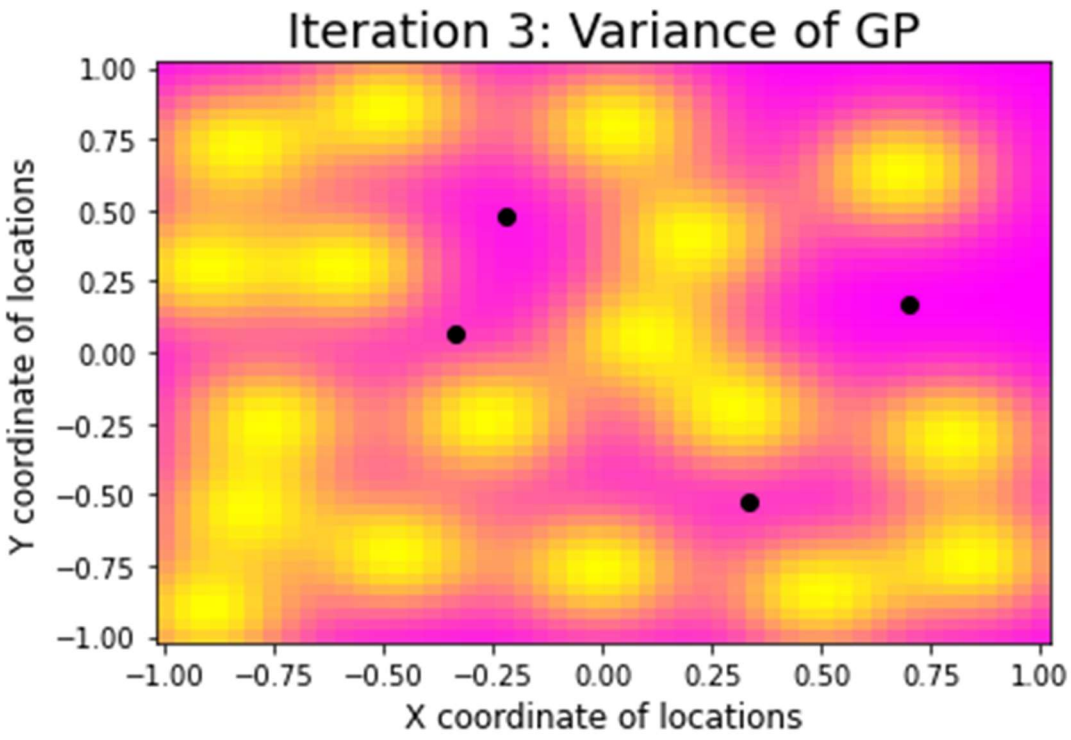


Figure 6: Variance of GP and new optimal locations for Iteration 3

Iteration 4

The set of known values are:

$$x^* = [0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9, -0.771, -0.264, -0.474, 0.035, 0.835, -0.814, -0.830, 0.682, 0.331, -0.340, -0.223, 0.698]$$

$$y^* = [0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9, -0.256, -0.236, -0.704, 0.793, -0.726, -0.551, 0.722, 0.633, -0.529, 0.0703, 0.483, 0.168]$$

$$Sensor\ Output = [3.39382006, 3.2073034, 3.39965035, 3.68810201, 2.96941623, 2.99495501, 3.94274928, 2.7968011, 3.34929734, 3.91296165, 3.39382006, 3.2073034, 3.39965035, 3.68810201, 4.08231430, 3.13022618, 3.19713096, 3.28089254, 4.16626462, 2.90255492, 4.80939220, 4.60522274]$$

The predicted x and y coordinates, remain the same for this iteration: values ranging from [-1,1] x [-1,1] sliced into 50 divisions

Using these values, I found the last four set of hyperparameters using optimization. In this iteration I used the hyperparameters obtained from the previous iteration as the initial point. The objective

function used here is the same as the last iteration, the marginal likelihood function and so are the bounds, $[(0, None), (0, None), (0, None)]$. The optimized results have been tabulated below:

Table 5: Choosing Hyperparameters for Iteration 4

Initial Points	τ	L	σ^2
$(\tau_0 = 0.5798, L_0 = 0.2244, \sigma_0^2 = 1.06 * 10^{-6})$	0.7219	0.1615	$1.42 * 10^{-6}$

Using the hyperparameters, I ran the Gaussian Process Regression again and obtained the predicted mean and predicted covariance. Figure 7 shows the plot of the mean distribution obtained from the predicted mean, over the mesh-grid. The figure also shows the band of uncertainty above and below the mean distribution curve.

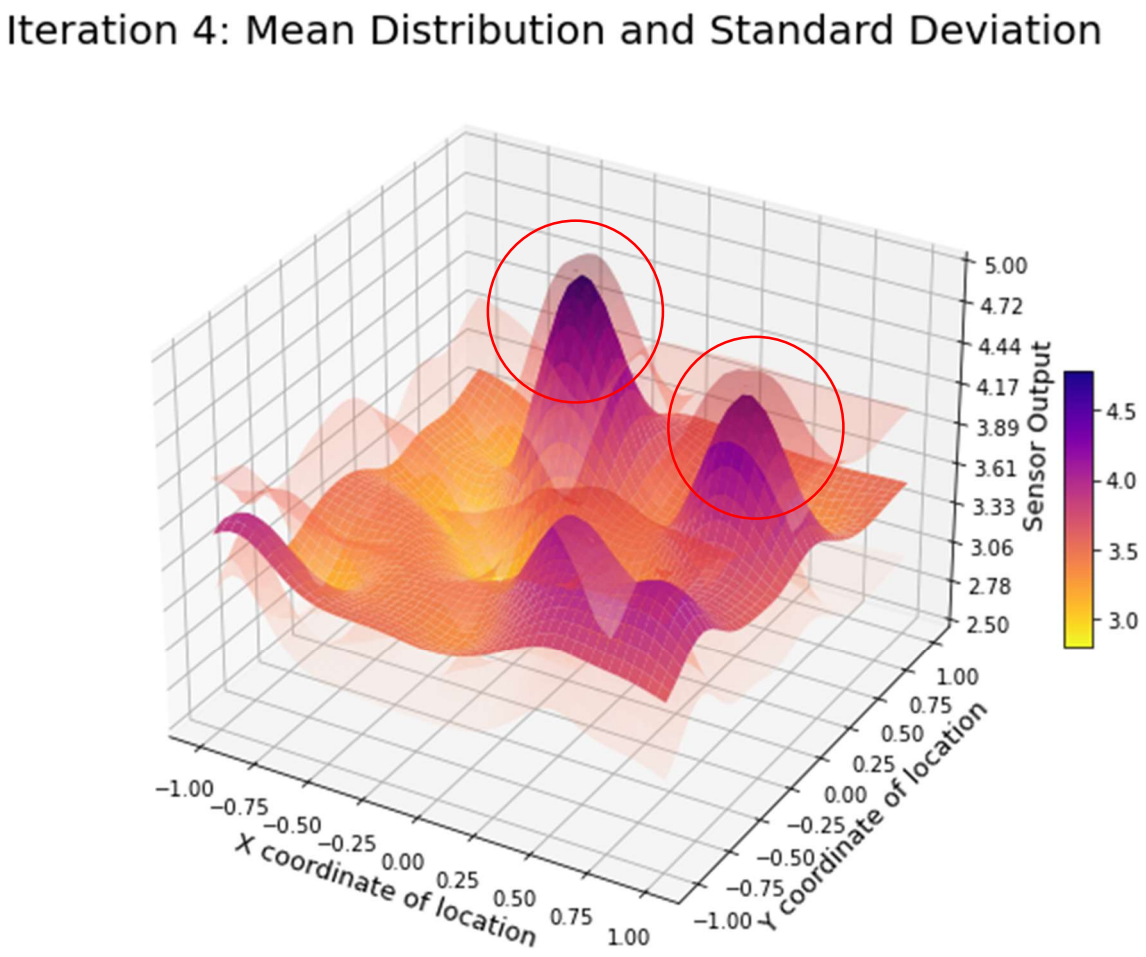


Figure 7: Mean distribution and standard deviation for the fourth iteration

Figure 7 represents the final curve fit of all known locations and their corresponding sensor data. The map in the middle is the mean distribution and the two gradient-colored maps above and below the mean distribution represents the standard deviation from the mean. From Figure 7, we can see two locations that have significantly higher mean values than the rest of the Gaussian Process.

There is a high probability that the object emanating the scent is buried at one of these two locations, based on the model obtained. These two locations have been called out using red circles in Figure 7.

Figure 8 shows a contour plot of the mean distribution of the final curve fit where 22 sets of datapoints are known.

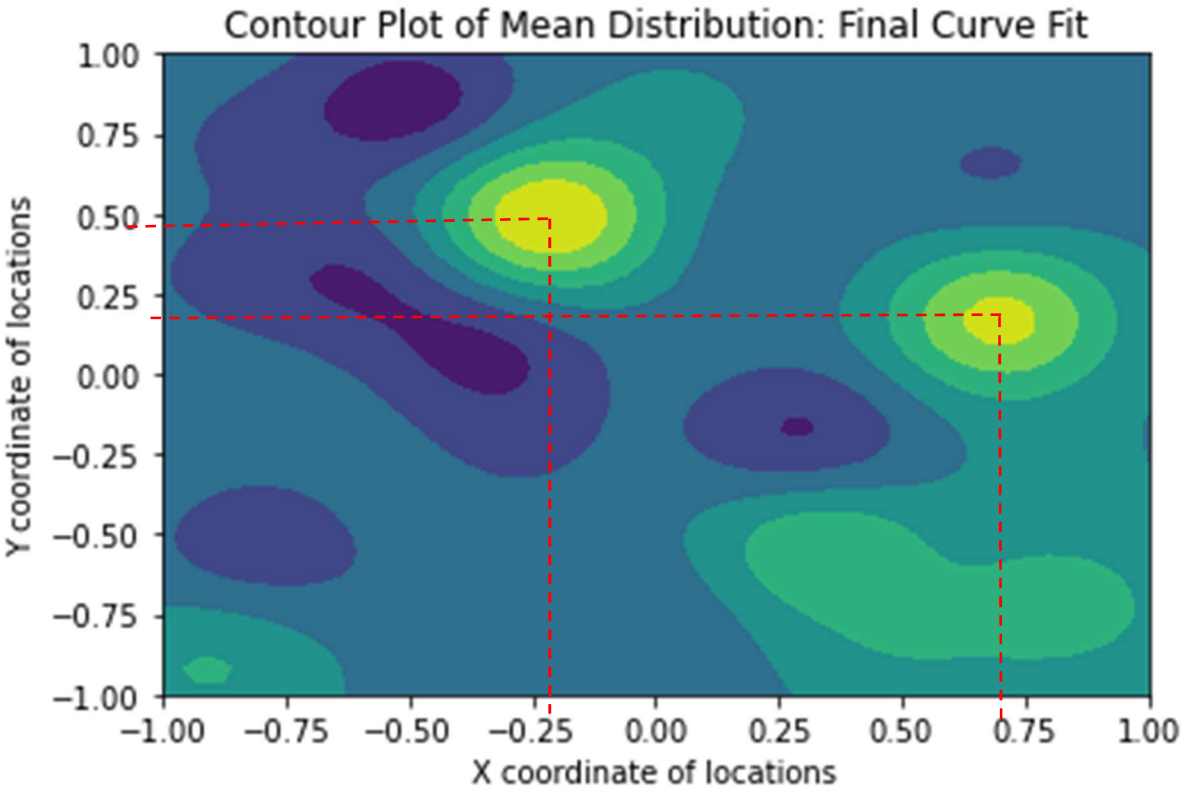


Figure 8: Contour plot of final curve fit

In Figure 8, the two yellow regions represent the locations of the highest mean values. These locations have been called out using red dotted lines. To exactly locate these two sets of datapoints, I studied the sensor data obtained from the robot sensor application for all 12 points found by the optimizer. These values are shown in Figure 9 along with their location. This figure is a screenshot of the data obtained from the application. The two locations in question are the requested points 11 and 12 that have the two highest scent values.

Results:

Highest scent sensor value = $\mathbf{4.809}$
Location of highest sensor value: $(\mathbf{-0.223, 0.483})$
This is also the location of highest mean value as shown in the contour plot above.

Second highest scent sensor value = **4.605**
Location of second highest sensor value: $(\mathbf{0.698, 0.168})$

AE567: Project 2 Part 1: Find the Source!

Logged in as Harikumar,Vaishnavi

Log out

Real Problem

You have 0 requests remaining to query 0 points.

timestamp	request_no	point_no	input	output
2022-11-04T06:17:25.000Z	3	12	0.698,0.16800001	4.605222740786023
2022-11-04T06:17:25.000Z	3	11	-0.223,0.483	4.809392196773036
2022-11-04T06:17:25.000Z	3	10	-0.34,0.07	2.902554927742022
2022-11-04T06:17:25.000Z	3	9	0.33100003,-0.52900004	4.16626461945789
2022-11-04T04:33:45.000Z	2	8	0.68200004,0.633	3.2808925410765357
2022-11-04T04:33:45.000Z	2	7	-0.83,0.72200006	3.197130960203049
2022-11-04T04:33:45.000Z	2	6	-0.814,-0.551	3.1302261886957137
2022-11-04T04:33:45.000Z	2	5	0.83500004,-0.726	4.082314307733955
2022-11-04T02:09:08.000Z	1	4	0.035,0.79300004	2.593704198779242
2022-11-04T02:09:08.000Z	1	3	-0.47400004,-0.70400006	3.45780891463846
2022-11-04T02:09:08.000Z	1	2	-0.263,-0.23600002	3.2739620121530977
2022-11-04T02:09:08.000Z	1	1	-0.77,-0.256	3.9384058763735057

The two points specified in the results are the two possible points that I have chosen to claim the location at which the object is buried. The highest probability is at the point of highest mean and the second highest probability is at the point of second highest mean. The scent value at the point where the object is buried will be the highest. The mean values essentially correspond to the sensor data. This is the reason for making this claim. The mean distribution curve of the Gaussian Process and the contour plot of the mean distribution gives high confidence to this claim.

Finding Confidence Intervals:

A method of choosing the level of confidence for the model is to run a Monte-Carlo algorithm on the Gaussian Process Regression. For example, if we run the gpr () function for 10^5 times and find an estimator of the predicted mean for the same set of hyperparameter, we can obtain the probability of the highest mean being at the same location. Essentially, here I would find the predicted mean from gpr () for n runs and find a mean estimator at each location. Then I would count the number of times that a predicted mean was highest at an exact given location. Dividing this count by the total number of trials, n would give me the probability of the mean being at the given location. This probability would give us the level of confidence we have in the model. I did not have the time to implement this idea in code, but I do believe that this method would help me find the confidence in my model and correspondingly, the confidence intervals.

Project 2a

In [32]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
from matplotlib import cm
from matplotlib.ticker import LinearLocator
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

In [33]:
```
%matplotlib inline
import numpy as np
import scipy.stats as stats
import scipy.special as scispec
import scipy.optimize
import matplotlib.pyplot as plt
import sympy
from sympy.utilities.lambdify import lambdify
from scipy import optimize
from scipy.optimize import minimize
```

In [34]:
```
!pip install --upgrade scipy==1.4.1
```

```
Requirement already satisfied: scipy==1.4.1 in c:\users\ivaishi\anaconda3\lib\s
ite-packages (1.4.1)
Requirement already satisfied: numpy>=1.13.3 in c:\users\ivaishi\anaconda3\lib
\site-packages (from scipy==1.4.1) (1.21.5)
```

In [35]:
```
def build_covariance(x, xp, y, yp, func, tau, L):
    """Build a covariance matrix

    Inputs
    -------
    x: (N) array of inputs
    xp: (M) array of inptus
    kern: a function mapping inputs to covariance

    Outputs
    -------
    cov: (N, M) covariance matrix
    """
    kern = func
    out = np.zeros((x.shape[0], xp.shape[0]))
    for jj in range(xp.shape[0]):
        out[:, jj] = kern(x, xp[jj], y, yp[jj], tau, L)
    return out
```

In [36]:
```
def sqexp(x, xp, y, yp, tau, L):
    """Squared exponential kernel (2 dimensional)

    Inputs
    ------
    x : (N), array of multiple inputs
    xp: float
    y : (N), array of multiple inputs
    yp: float

    Returns
    -------
    cov (N,) -- Covariance between each input at *x* and the function values at *
    """
    cov = tau**2 * np.exp(-1/2 * (x - xp)**2 / L**2)*tau**2 * np.exp(-1/2 * (y -
    return cov
```

In [37]:
```
# constructing gridspace
n_div = 50
x = np.linspace(-1, 1, n_div)
y = np.linspace(-1, 1, n_div)
xbar, ybar = np.meshgrid(x,y)
xbar_new = np.reshape(xbar,(n_div**2))
ybar_new = np.reshape(ybar,(n_div**2))
ybar_new = ybar_new.T
print(xbar_new.shape)
#print(xbar_new)
print(ybar_new.shape)
#print(ybar_new)
```

```
(2500,)
(2500,)
```

In [38]:
```
n_init = 10
xinit = np.array([0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9])
xinit = xinit.T
yinit = np.array([0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9])
yinit = yinit.T
sen_out = np.array([3.39382006, 3.2073034, 3.39965035, 3.68810201, 2.96941623, 2.
print((sen_out.T).shape)
sen_out_new = np.reshape(sen_out,(-1,1))
mean_func = np.mean(sen_out)*np.ones((n_init,1))
print(sen_out_new.shape)
```

```
(10,)
(10, 1)
```

In [39]:
```
def max_marg_likelihood(des_var):
    tau, L, sigma = des_var
    cov = build_covariance(xinit, xinit, yinit, yinit, sqexp, tau, L)
    mean_func = np.mean(sen_out)*np.ones((n_init,1))

    subterm_1 = (sen_out_new - mean_func).T
    subterm_2 = np.linalg.inv(cov + (sigma**2)*np.identity(xinit.shape[0]))
    subterm_3 = sen_out_new - mean_func

    term1 = np.dot(subterm_1,np.dot(subterm_2,subterm_3))
    term2 = np.log(np.linalg.det((cov + (sigma**2)*np.identity(xinit.shape[0]))))
    term3 = (n_init)*np.log(2*np.pi)

    mx_marg_lh = (1/2)*(term1 + term2 + term3)

    return mx_marg_lh
```

In [63]:
```
const = ((0,None),(0,None),(0,None))
results = scipy.optimize.minimize(max_marg_likelihood, (0.65, 0.45, 0.01), bounds
#results = scipy.optimize.minimize(max_marg_likelihood, (0.95, 0.5, 0.01), bounds
#results = scipy.optimize.minimize(max_marg_likelihood, (0.8, 0.5, 0.01), bounds
#results = scipy.optimize.minimize(max_marg_likelihood, (0.7, 0.4, 0.01), bounds
#results = scipy.optimize.minimize(max_marg_likelihood, (0.95, 0.5, 0.01), bounds

print(results)
```

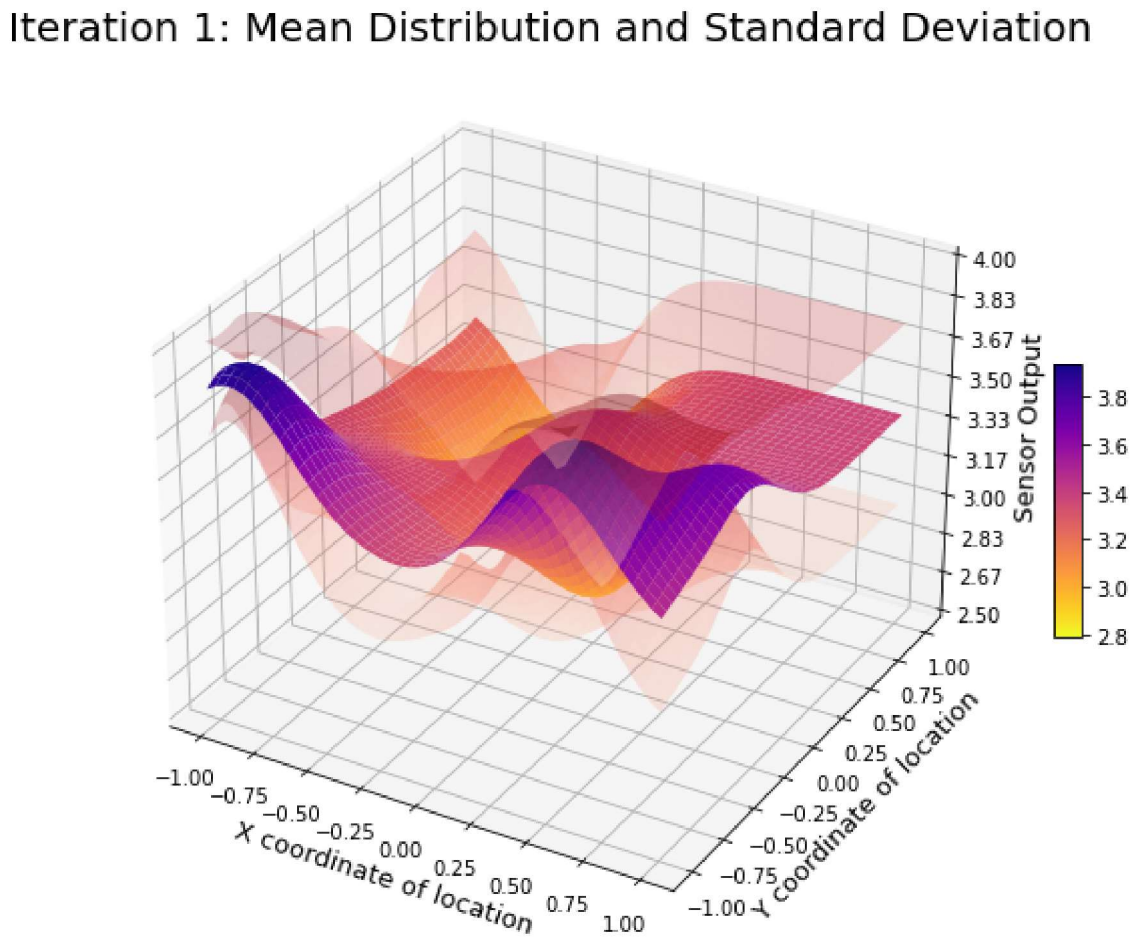
```
      fun: array([[4.06783639]])
 hess_inv: <3x3 LbfgsInvHessProduct with dtype=float64>
      jac: array([6.30606678e-06, 1.24344979e-06, 1.77635684e-07])
  message: b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
     nfev: 92
      nit: 20
   status: 0
  success: True
        x: array([0.6156354 , 0.26413172, 0.        ])
```

In [41]:
```
def gpr(xinit, yinit, xpred, ypred, out_sensor, noise_var, mean_func, kernel, tau
    """Gaussian process regression Algorithm

    Inputs
    -------
    xinit: (N, ) training inputs
    yinit: (N, ) training outputs
    xpred:  (M, ) locations at which to make predictions
    ypred:  (M, ) locations at which to make predictions
    noise_var: (N, ) noise at every training output
    mean_func: function to compute the prior mean
    kernel: covariance kernel

    Returns
    -------
    pred_mean : (M, ) predicted mean at prediction points
    pred_cov : (M, M) predicted covariance at the prediction points
    --
    """
    # constructing covariance matrix from initial datapoints given
    cov = build_covariance(xinit, xinit, yinit, yinit, kernel, tau, L)
    u, s, v = np.linalg.svd(cov)
    sqrtcov = np.dot(u, np.sqrt(np.diag(s)))

    # pseudoinverse is better conditioned
    invcov = np.linalg.pinv(cov + np.diag(noise_var))
    mean_func = np.mean(sen_out)
    vec_pred = build_covariance(xpred, xinit, ypred, yinit, kernel, tau, L)
    print(vec_pred.shape)
    pred_mean = mean_func*np.ones(xpred.shape[0]) + np.matmul(vec_pred, np.matmul

    cov_predict_pre = build_covariance(xpred, xpred, ypred, ypred, kernel, tau, L
    cov_predict_up = np.dot(vec_pred, np.dot(invcov, vec_pred.T))
    pred_cov = cov_predict_pre - cov_predict_up

    return pred_mean, pred_cov
```

In [42]:
```
tau = 6.15635315e-01
L = 2.64131426e-01
sigma = 1.01836023e-06
mean_func = np.mean(sen_out)*np.ones((n_init,1))
xinit = np.array([0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9])
yinit = np.array([0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9])
sen_out = np.array([3.39382006, 3.2073034, 3.39965035, 3.68810201, 2.96941623, 2.
sigma_opt = sigma * np.ones((xinit.shape[0]))
mean_out, cov_out = gpr(xinit, yinit, xbar_new, ybar_new, sen_out, sigma_opt, mea
print(mean_out)
```

```
(2500, 10)
[3.84000689 3.86259248 3.87394273 ... 3.3654187  3.36544956 3.36547045]
```

In [112]:
```
fig, ax = plt.subplots(subplot_kw={"projection": "3d"}, figsize = (10,10))
# Plot the surface.
pred_mean = np.reshape(mean_out, (50,50))
#print(pred_mean)
mean = ax.plot_surface(xbar, ybar, pred_mean, cmap='plasma_r')
var = np.sqrt(np.diag(cov_out))
var_to_plot = np.reshape(var, (50,50))
#print(var_to_plot.shape)
#print(pred_mean.shape)
var_1 = ax.plot_surface(xbar, ybar, pred_mean + var_to_plot, cmap='Reds', alpha =
var_2 = ax.plot_surface(xbar, ybar, pred_mean - var_to_plot, cmap='Reds', alpha =
#
# Customize the z axis.
ax.set_zlim(2.5, 4)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.set_xlabel('X coordinate of location', fontsize = 13)
ax.set_ylabel('Y coordinate of location', fontsize = 13)
ax.set_zlabel('Sensor Output', fontsize = 13)
# A StrMethodFormatter is used automatically
ax.zaxis.set_major_formatter('{x:.02f}')
ax.set_title('Iteration 1: Mean Distribution and Standard Deviation', fontsize =


# Add a color bar which maps values to colors.
fig.colorbar(mean, shrink=0.25, aspect=10)

plt.show()
```



In [104]:
```
# Finding the trace of the covariance matrix in order to find the variance of the
# Covariance of an element with itself gives its variance
# The diagonal of the covariance matrix represents the variances of all the eleme
# The trace of the covariance matrix is the sum of diagonal elements
#
# tau = 1.90872019
# L = 11.93990254
# sigma = 0.27930536

tau = 6.15635315e-01
L = 2.64131426e-01
sigma = 1.01836023e-06

def var_from_trace_1(points):

    x1,x2,x3,x4,y1,y2,y3,y4 = points
    xinit = np.array([0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9, x1,
    yinit = np.array([0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9,

    sigma_new = sigma * np.ones((xinit.shape[0]))
    # constructing covariance matrix from initial datapoints given
    cov = build_covariance(xinit, xinit, yinit, yinit, sqexp, tau, L)
    u, s, v = np.linalg.svd(cov)
    sqrtcov = np.dot(u, np.sqrt(np.diag(s)))

    # pseudoinverse is better conditioned
    invcov = np.linalg.pinv(cov + np.diag(sigma_new))
    vec_pred = build_covariance(xbar_new, xinit, ybar_new, yinit, sqexp, tau, L)

    cov_predict_pre = build_covariance(xbar_new, xbar_new, ybar_new, ybar_new, sc
    cov_predict_up = np.dot(vec_pred, np.dot(invcov, vec_pred.T))
    pred_cov = cov_predict_pre - cov_predict_up

    var_dist = np.trace(pred_cov)

    return var_dist
```

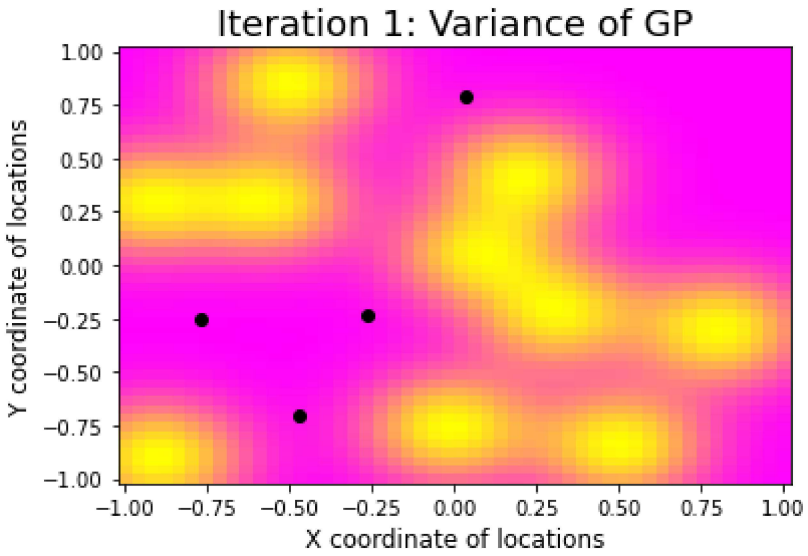
In [114]:
```
const = ((-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1))
results = scipy.optimize.minimize(var_from_trace_1, (0, 0, 0, 0, 0, 0, 0, 0), bou
print(results)
```

```
      fun: 159.12646040404
 hess_inv: <8x8 LbfgsInvHessProduct with dtype=float64>
      jac: array([ 2.52953214e-04, -4.12114787e-04, -1.16529009e-04,  2.8421709
4e-05,
        2.67164069e-04, -1.19371180e-04,  3.72324394e-04,  1.81898940e-04])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 468
      nit: 37
   status: 0
  success: True
        x: array([-0.77028974, -0.26354258, -0.47413339,  0.03540891, -0.256467
05,
       -0.23566883, -0.70396699,  0.79324487])
```

In [153]:
```
# n_div = 50
# x = np.linspace(-1, 1, n_div)
# y = np.linspace(-1, 1, n_div)
# xbar, ybar = np.meshgrid(x,y)
var = np.diag(cov_out)
var_to_plot = np.reshape(var, (50,50))
print(var_to_plot)
c = plt.pcolormesh(xbar,ybar,var_to_plot, cmap = "spring_r")
x_values = [(results.x[0],results.x[1],results.x[2],results.x[3])]
y_values = [(results.x[4],results.x[5],results.x[6],results.x[7])]
plt.scatter(x_values, y_values, c = 'black')
plt.title('Iteration 1: Variance of GP', fontsize = 18)
plt.xlabel('X coordinate of locations', fontsize = 12)
plt.ylabel('Y coordinate of locations', fontsize = 12)
```

```
[[0.03580318 0.02527739 0.01978343 ... 0.13534716 0.13866329 0.14078912]
 [0.02527719 0.01372404 0.00769384 ... 0.13381456 0.13772055 0.14022968]
 [0.01978297 0.00769358 0.00138348 ... 0.13245378 0.13685009 0.13968626]
 ...
 [0.1393667  0.13653245 0.13214999 ... 0.14364396 0.14364525 0.14364581]
 [0.14001111 0.1374611  0.13349939 ... 0.143645   0.14364569 0.14364598]
 [0.14062117 0.13843005 0.13501724 ... 0.14364558 0.14364593 0.14364608]]
```

Out[153]: Text(0, 0.5, 'Y coordinate of locations')



In [ ]:
```
```

In [1]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
from matplotlib import cm
from matplotlib.ticker import LinearLocator
```

In [2]:
```
%matplotlib inline
import numpy as np
import scipy.stats as stats
import scipy.special as scispec
import scipy.optimize
import matplotlib.pyplot as plt
import sympy
from sympy.utilities.lambdify import lambdify
from scipy import optimize
from scipy.optimize import minimize
```

In [3]:
```
def build_covariance(x, xp, y, yp, func, tau, L):
    """Build a covariance matrix

    Inputs
    -------
    x: (N) array of inputs
    xp: (M) array of inptus
    kern: a function mapping inputs to covariance

    Outputs
    -------
    cov: (N, M) covariance matrix
    """
    kern = func
    out = np.zeros((x.shape[0], xp.shape[0]))
    for jj in range(xp.shape[0]):
        out[:, jj] = kern(x, xp[jj], y, yp[jj], tau, L)
    return out
```

In [4]:
```
def sqexp(x, xp, y, yp, tau, L):
    """Squared exponential kernel (2 dimensional)

    Inputs
    ------
    x : (N), array of multiple inputs
    xp: float
    y : (N), array of multiple inputs
    yp: float

    Returns
    -------
    cov (N,) -- Covariance between each input at *x* and the function values at *
    """
    cov = tau**2 * np.exp(-1/2 * (x - xp)**2 / L**2)*tau**2 * np.exp(-1/2 * (y -
    return cov
```

In [5]:
```
# constructing gridspace
n_div = 50
x = np.linspace(-1, 1, n_div)
y = np.linspace(-1, 1, n_div)
xbar, ybar = np.meshgrid(x,y)
xbar_new = np.reshape(xbar,(n_div**2))
ybar_new = np.reshape(ybar,(n_div**2))
ybar_new = ybar_new.T
print(xbar_new.shape)
#print(xbar_new)
print(ybar_new.shape)
#print(ybar_new)
```

```
(2500,)
(2500,)
```

In [6]:
```
])

968011, 3.34929734, 3.91296165, 3.39382006, 3.2073034, 3.39965035, 3.68810201])
```

```
(14,)
(14, 1)
```

In [7]:
```
def max_marg_likelihood(des_var):
    tau, L, sigma = des_var
    cov = build_covariance(xinit, xinit, yinit, yinit, sqexp, tau, L)

    mean_func = np.mean(sen_out)*np.ones((n_init,1))

    subterm_1 = (sen_out_new - mean_func).T
    subterm_2 = np.linalg.pinv(cov + (sigma**2)*np.identity(xinit.shape[0]))
    subterm_3 = sen_out_new - mean_func

    term1 = np.dot(subterm_1,np.dot(subterm_2,subterm_3))
    term2 = np.log(np.linalg.det((cov + (sigma**2)*np.identity(xinit.shape[0]))))
    term3 = (n_init)*np.log(2*np.pi)

    mx_marg_lh = (1/2)*(term1 + term2 + term3)

    return mx_marg_lh
```

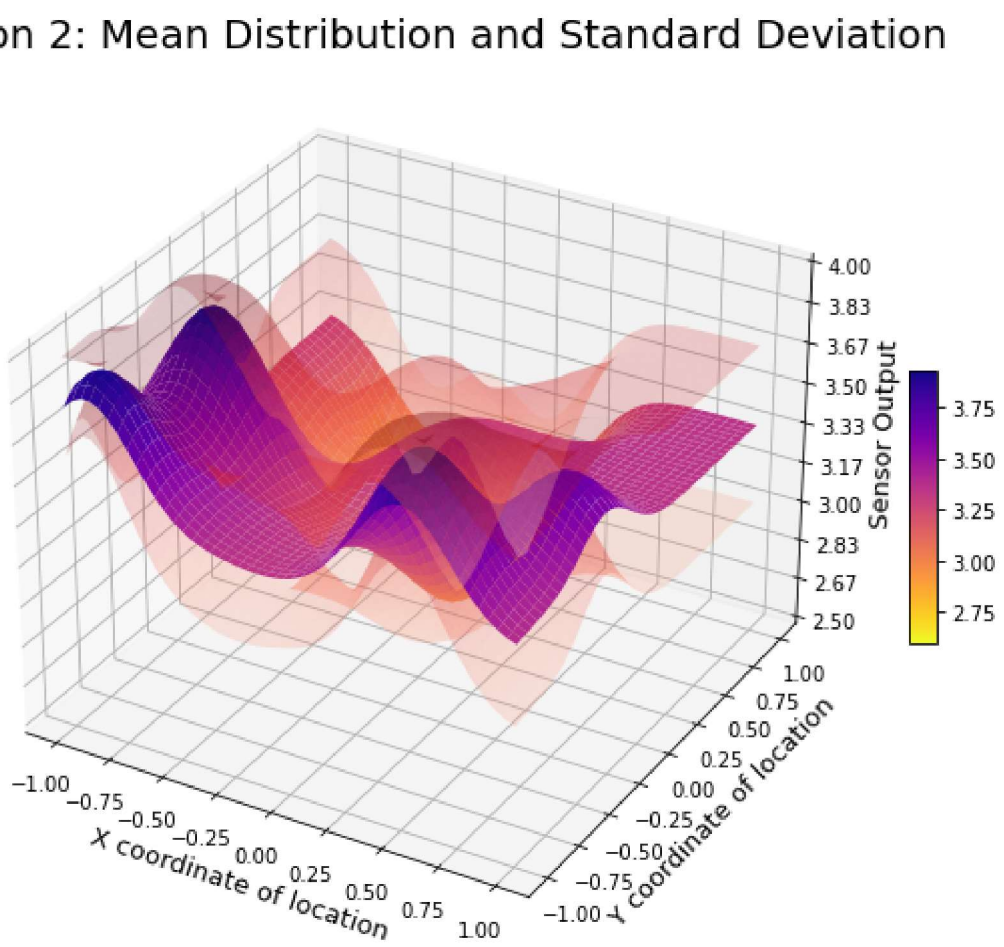
In [8]:
```
const = ((0,None),(0,None),(0,None))
#results = scipy.optimize.minimize(max_marg_likelihood, (0.65, 0.45, 0.01), bound
#results = scipy.optimize.minimize(max_marg_likelihood, (0.95, 0.5, 0.01), bounds
hyperParameters = scipy.optimize.minimize(max_marg_likelihood, (6.15635315e-01, 2
#results = scipy.optimize.minimize(max_marg_likelihood, (0.7, 0.4, 0.01), bounds
#results = scipy.optimize.minimize(max_marg_likelihood, (0.1, 0.45, 0.001), bound

print(hyperParameters)
```

```
      fun: 4.132540815538965
      jac: array([ 2.50339508e-06, -1.19209290e-07,  8.34465027e-07])
  message: 'Optimization terminated successfully.'
     nfev: 90
      nit: 8
     njev: 18
   status: 0
  success: True
        x: array([5.73767342e-01, 2.10367359e-01, 1.02150013e-06])

C:\Users\ivaishi\Anaconda3\lib\site-packages\scipy\optimize\_minimize.py:522: R
untimeWarning: Method CG cannot handle constraints nor bounds.
  RuntimeWarning)
```

In [9]:
```
def gpr(xinit, yinit, xpred, ypred, out_sensor, noise_var, mean_func, kernel, tau
    """Gaussian process regression Algorithm

    Inputs
    -------
    xinit: (N, ) training inputs
    yinit: (N, ) training outputs
    xpred:  (M, ) locations at which to make predictions
    ypred:  (M, ) locations at which to make predictions
    noise_var: (N, ) noise at every training output
    mean_func: function to compute the prior mean
    kernel: covariance kernel

    Returns
    -------
    pred_mean : (M, ) predicted mean at prediction points
    pred_cov : (M, M) predicted covariance at the prediction points
    --
    """
    # constructing covariance matrix from initial datapoints given
    cov = build_covariance(xinit, xinit, yinit, yinit, kernel, tau, L)
    u, s, v = np.linalg.svd(cov)
    sqrtcov = np.dot(u, np.sqrt(np.diag(s)))

    # pseudoinverse is better conditioned
    invcov = np.linalg.pinv(cov + np.diag(noise_var))
    mean_func = np.mean(sen_out)*np.ones((n_init,1))
    mean_func = np.mean(sen_out)
    vec_pred = build_covariance(xpred, xinit, ypred, yinit, kernel, tau, L)
    pred_mean = mean_func*np.ones(xpred.shape[0]) + np.matmul(vec_pred, np.matmul

    cov_predict_pre = build_covariance(xpred, xpred, ypred, ypred, kernel, tau, L
    cov_predict_up = np.dot(vec_pred, np.dot(invcov, vec_pred.T))
    pred_cov = cov_predict_pre - cov_predict_up

    return pred_mean, pred_cov
```

In [10]:
```
tau = hyperParameters.x[0]
L = hyperParameters.x[1]
sigma = hyperParameters.x[2]
mean_func = np.mean(sen_out)*np.ones((n_init,1))
xinit = np.array([0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9,-0.770,
yinit = np.array([0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9, -0.2
sen_out = np.array([3.39382006, 3.2073034, 3.39965035, 3.68810201, 2.96941623, 2.
sigma_opt = sigma * np.ones((xinit.shape[0]))
mean_out, cov_out = gpr(xinit, yinit, xbar_new, ybar_new, sen_out, sigma_opt, mea
```

In [11]:
```
fig, ax = plt.subplots(subplot_kw={"projection": "3d"}, figsize = (10,10))
# Plot the surface.
pred_mean = np.reshape(mean_out, (50,50))
#print(pred_mean)
mean = ax.plot_surface(xbar, ybar, pred_mean, cmap='plasma_r')
var = np.sqrt(np.diag(cov_out))
var_to_plot = np.reshape(var, (50,50))
#print(var_to_plot.shape)
#print(pred_mean.shape)
var_1 = ax.plot_surface(xbar, ybar, pred_mean + var_to_plot, cmap='Reds', alpha =
var_2 = ax.plot_surface(xbar, ybar, pred_mean - var_to_plot, cmap='Reds', alpha =
#
# Customize the z axis.
ax.set_zlim(2.5, 4)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.set_xlabel('X coordinate of location', fontsize = 13)
ax.set_ylabel('Y coordinate of location', fontsize = 13)
ax.set_zlabel('Sensor Output', fontsize = 13)
# A StrMethodFormatter is used automatically
ax.zaxis.set_major_formatter('{x:.02f}')
ax.set_title('Iteration 2: Mean Distribution and Standard Deviation', fontsize =


# Add a color bar which maps values to colors.
fig.colorbar(mean, shrink=0.25, aspect=10)

plt.show()
```


Iteration 2: Mean Distribution and Standard Deviation
Sensor Output
4.00
3.83
3.67
3.50
3.33
3.17
3.00
2.83
2.67
2.50
3.75
3.50
3.25
3.00
2.75
X coordinate of location
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
Y coordinate of location
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

In [12]:
```
# tau = 5.73767342e-01
# L = 2.10367359e-01
# sigma = 1.02150013e-06

def var_from_trace_1(points):

    x1,x2,x3,x4,y1,y2,y3,y4 = points
    xinit = np.array([0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9, -0.
    yinit = np.array([0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9,

    sigma_new = sigma * np.ones((xinit.shape[0]))
    # constructing covariance matrix from initial datapoints given
    cov = build_covariance(xinit, xinit, yinit, yinit, sqexp, tau, L)
    u, s, v = np.linalg.svd(cov)
    sqrtcov = np.dot(u, np.sqrt(np.diag(s)))

    # pseudoinverse is better conditioned
    invcov = np.linalg.pinv(cov + np.diag(sigma_new))
    vec_pred = build_covariance(xbar_new, xinit, ybar_new, yinit, sqexp, tau, L)

    cov_predict_pre = build_covariance(xbar_new, xbar_new, ybar_new, ybar_new, sq
    cov_predict_up = np.dot(vec_pred, np.dot(invcov, vec_pred.T))
    pred_cov = cov_predict_pre - cov_predict_up

    var_dist = np.trace(pred_cov)

    return var_dist
```

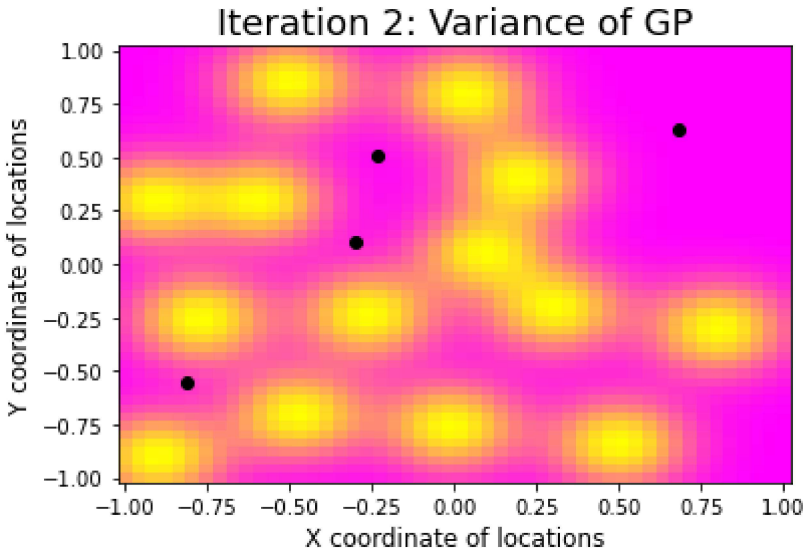
In [13]:
```
const = ((-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1))
results = scipy.optimize.minimize(var_from_trace_1, (-0.771, -0.264, -0.474, 0.03
print(results)
```

```
      fun: 135.10643767823058
 hess_inv: <8x8 LbfgsInvHessProduct with dtype=float64>
      jac: array([-0.00185594,  0.00024727,  0.00019327,  0.00059117,  0.001017
5 ,
        0.00025011,  0.00100613, -0.00247269])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 405
      nit: 29
   status: 0
  success: True
        x: array([-0.23180735, -0.8141927 , -0.30386648,  0.68297476,  0.504082
11,
       -0.55159598,  0.10188397,  0.63305779])
```

In [15]:
```
var = np.diag(cov_out)
var_to_plot = np.reshape(var, (50,50))
print(var_to_plot)
c = plt.pcolormesh(xbar,ybar,var_to_plot, cmap = "spring_r")
x_values = [(results.x[0],results.x[1],results.x[2],results.x[3])]
y_values = [(results.x[4],results.x[5],results.x[6],results.x[7])]
plt.scatter(x_values, y_values, c = 'black')
plt.title('Iteration 2: Variance of GP', fontsize = 18)
plt.xlabel('X coordinate of locations', fontsize = 12)
plt.ylabel('Y coordinate of locations', fontsize = 12)
```

```
[[0.03912862 0.0282699  0.02244635 ... 0.10721104 0.10785858 0.10816366]
 [0.02821793 0.01565861 0.0089189  ... 0.10686629 0.10770426 0.1080993 ]
 [0.0223228  0.00884783 0.0016016  ... 0.10655686 0.10756362 0.10803911]
 ...
 [0.10801855 0.10752745 0.1064886  ... 0.10837854 0.10837854 0.10837854]
 [0.10808152 0.10766562 0.10678388 ... 0.10837854 0.10837854 0.10837854]
 [0.10814713 0.10781992 0.10712564 ... 0.10837854 0.10837854 0.10837854]]
```

Out[15]: Text(0, 0.5, 'Y coordinate of locations')



In [ ]:
```
```

In [1]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
from matplotlib import cm
from matplotlib.ticker import LinearLocator
```

In [2]:
```
%matplotlib inline
import numpy as np
import scipy.stats as stats
import scipy.special as scispec
import scipy.optimize
import matplotlib.pyplot as plt
import sympy
from sympy.utilities.lambdify import lambdify
from scipy import optimize
from scipy.optimize import minimize
```

In [3]:
```
def build_covariance(x, xp, y, yp, func, tau, L):
    """Build a covariance matrix

    Inputs
    -------
    x: (N) array of inputs
    xp: (M) array of inptus
    kern: a function mapping inputs to covariance

    Outputs
    -------
    cov: (N, M) covariance matrix
    """
    kern = func
    out = np.zeros((x.shape[0], xp.shape[0]))
    for jj in range(xp.shape[0]):
        out[:, jj] = kern(x, xp[jj], y, yp[jj], tau, L)
    return out
```

In [4]:
```
def sqexp(x, xp, y, yp, tau, L):
    """Squared exponential kernel (2 dimensional)

    Inputs
    ------
    x : (N), array of multiple inputs
    xp: float
    y : (N), array of multiple inputs
    yp: float

    Returns
    -------
    cov (N,) -- Covariance between each input at *x* and the function values at *
    """
    cov = tau**2 * np.exp(-1/2 * (x - xp)**2 / L**2)*tau**2 * np.exp(-1/2 * (y -
    return cov
```

In [5]:
```
# constructing gridspace
n_div = 50
x = np.linspace(-1, 1, n_div)
y = np.linspace(-1, 1, n_div)
xbar, ybar = np.meshgrid(x,y)
xbar_new = np.reshape(xbar,(n_div**2))
ybar_new = np.reshape(ybar,(n_div**2))
ybar_new = ybar_new.T
print(xbar_new.shape)
#print(xbar_new)
print(ybar_new.shape)
#print(ybar_new)
```

```
(2500,)
(2500,)
```

In [6]:
```
73034, 3.39965035, 3.68810201, 4.08231430, 3.13022618, 3.19713096, 3.28089254])
```

```
(18,)
(18, 1)
```

In [7]:
```
def max_marg_likelihood(des_var):
    tau, L, sigma = des_var
    cov = build_covariance(xinit, xinit, yinit, yinit, sqexp, tau, L)

    mean_func = np.mean(sen_out)*np.ones((n_init,1))

    subterm_1 = (sen_out_new - mean_func).T
    subterm_2 = np.linalg.pinv(cov + (sigma**2)*np.identity(xinit.shape[0]))
    subterm_3 = sen_out_new - mean_func

    term1 = np.dot(subterm_1,np.dot(subterm_2,subterm_3))
    term2 = np.log(np.linalg.det((cov + (sigma**2)*np.identity(xinit.shape[0]))))
    term3 = (n_init)*np.log(2*np.pi)

    mx_marg_lh = (1/2)*(term1 + term2 + term3)

    return mx_marg_lh
```

In [8]:
```
g_likelihood, (0.65, 0.45, 0.01), bounds = const)
g_likelihood, (0.95, 0.5, 0.01), bounds = const)
max_marg_likelihood, (5.75420193e-01, 2.05142386e-01, 1.01729362e-06), bounds = c
rg_likelihood, (0.7, 0.4, 0.01), bounds = const)
g_likelihood, (0.1, 0.45, 0.001), bounds = const)
```

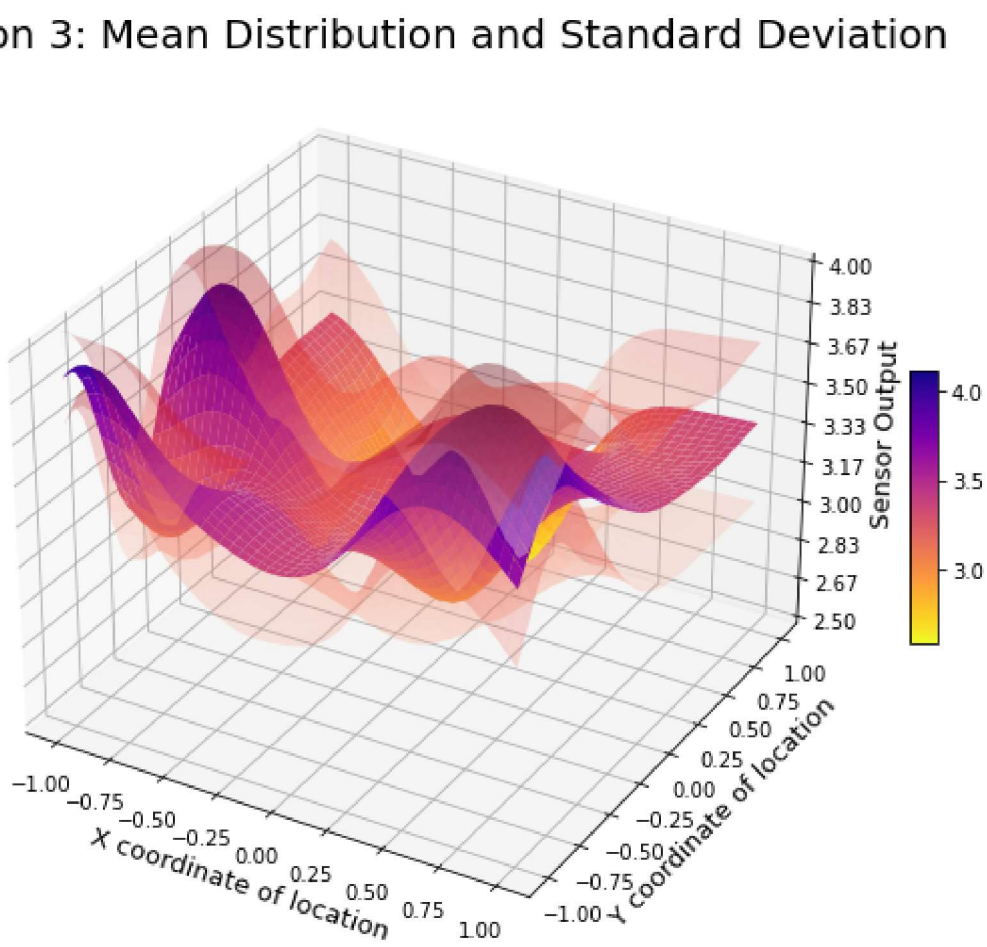
```
      fun: array([[5.3756907]])
 hess_inv: <3x3 LbfgsInvHessProduct with dtype=float64>
      jac: array([-1.95399252e-06,  1.77635684e-06, -2.48689958e-06])
  message: b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
     nfev: 36
      nit: 6
   status: 0
  success: True
        x: array([5.79779452e-01, 2.24441890e-01, 1.06125462e-06])
```

In [9]:
```
def gpr(xinit, yinit, xpred, ypred, out_sensor, noise_var, mean_func, kernel, tau
    """Gaussian process regression Algorithm

    Inputs
    -------
    xinit: (N, ) training inputs
    yinit: (N, ) training outputs
    xpred:  (M, ) locations at which to make predictions
    ypred:  (M, ) locations at which to make predictions
    noise_var: (N, ) noise at every training output
    mean_func: function to compute the prior mean
    kernel: covariance kernel

    Returns
    -------
    pred_mean : (M, ) predicted mean at prediction points
    pred_cov : (M, M) predicted covariance at the prediction points
    --
    """
    # constructing covariance matrix from initial datapoints given
    cov = build_covariance(xinit, xinit, yinit, yinit, kernel, tau, L)
    u, s, v = np.linalg.svd(cov)
    sqrtcov = np.dot(u, np.sqrt(np.diag(s)))

    # pseudoinverse is better conditioned
    invcov = np.linalg.pinv(cov + np.diag(noise_var))
    mean_func = np.mean(sen_out)*np.ones((n_init,1))
    mean_func = np.mean(sen_out)
    vec_pred = build_covariance(xpred, xinit, ypred, yinit, kernel, tau, L)
    #pred_mean = mean_func + np.dot(vec_pred, np.dot(invcov, out_sensor - mean_fu
    pred_mean = mean_func*np.ones(xpred.shape[0]) + np.matmul(vec_pred, np.matmul

    cov_predict_pre = build_covariance(xpred, xpred, ypred, ypred, kernel, tau, L
    cov_predict_up = np.dot(vec_pred, np.dot(invcov, vec_pred.T))
    pred_cov = cov_predict_pre - cov_predict_up

    return pred_mean, pred_cov
```

In [10]:
```
tau = hyperParameters.x[0]
L = hyperParameters.x[1]
sigma = hyperParameters.x[2]
mean_func = np.mean(sen_out)*np.ones((n_init,1))
#mean_func = lambda x: np.zeros((x.shape[0]))
xinit = np.array([0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9,-0.770,
yinit = np.array([0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9, -0.2
sen_out = np.array([3.39382006, 3.2073034, 3.39965035, 3.68810201, 2.96941623, 2.
sigma_opt = sigma * np.ones((xinit.shape[0]))
print(sen_out.shape)
mean_out, cov_out = gpr(xinit, yinit, xbar_new, ybar_new, sen_out, sigma_opt, mea
```

```
(18,)
```

In [11]:
```
fig, ax = plt.subplots(subplot_kw={"projection": "3d"}, figsize = (10,10))
# Plot the surface.
pred_mean = np.reshape(mean_out, (50,50))
#print(pred_mean)
mean = ax.plot_surface(xbar, ybar, pred_mean, cmap='plasma_r')
var = np.sqrt(np.diag(cov_out))
var_to_plot = np.reshape(var, (50,50))
#print(var_to_plot.shape)
#print(pred_mean.shape)
var_1 = ax.plot_surface(xbar, ybar, pred_mean + var_to_plot, cmap='Reds', alpha =
var_2 = ax.plot_surface(xbar, ybar, pred_mean - var_to_plot, cmap='Reds', alpha =
#
# Customize the z axis.
ax.set_zlim(2.5, 4)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.set_xlabel('X coordinate of location', fontsize = 13)
ax.set_ylabel('Y coordinate of location', fontsize = 13)
ax.set_zlabel('Sensor Output', fontsize = 13)
# A StrMethodFormatter is used automatically
ax.zaxis.set_major_formatter('{x:.02f}')
ax.set_title('Iteration 3: Mean Distribution and Standard Deviation', fontsize =


# Add a color bar which maps values to colors.
fig.colorbar(mean, shrink=0.25, aspect=10)

plt.show()
```


Iteration 3: Mean Distribution and Standard Deviation
Sensor Output
4.00
3.83
3.67
3.50
3.33
3.17
3.00
2.83
2.67
2.50
4.0
3.5
3.0
X coordinate of location
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
Y coordinate of location
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

In [15]:
```
# tau = 5.73767342e-01
# L = 2.10367359e-01
# sigma = 1.02150013e-06

def var_from_trace(points):

    x1,x2,x3,x4,y1,y2,y3,y4 = points
    xinit = np.array([0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9, -0.
    yinit = np.array([0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9,

    sigma_new = sigma * np.ones((xinit.shape[0]))
    # constructing covariance matrix from initial datapoints given
    cov = build_covariance(xinit, xinit, yinit, yinit, sqexp, tau, L)
    u, s, v = np.linalg.svd(cov)
    sqrtcov = np.dot(u, np.sqrt(np.diag(s)))

    # pseudoinverse is better conditioned
    invcov = np.linalg.pinv(cov + np.diag(sigma_new))
    vec_pred = build_covariance(xbar_new, xinit, ybar_new, yinit, sqexp, tau, L)

    cov_predict_pre = build_covariance(xbar_new, xbar_new, ybar_new, ybar_new, sq
    cov_predict_up = np.dot(vec_pred, np.dot(invcov, vec_pred.T))
    pred_cov = cov_predict_pre - cov_predict_up

    var_dist = np.trace(pred_cov)

    return var_dist
```

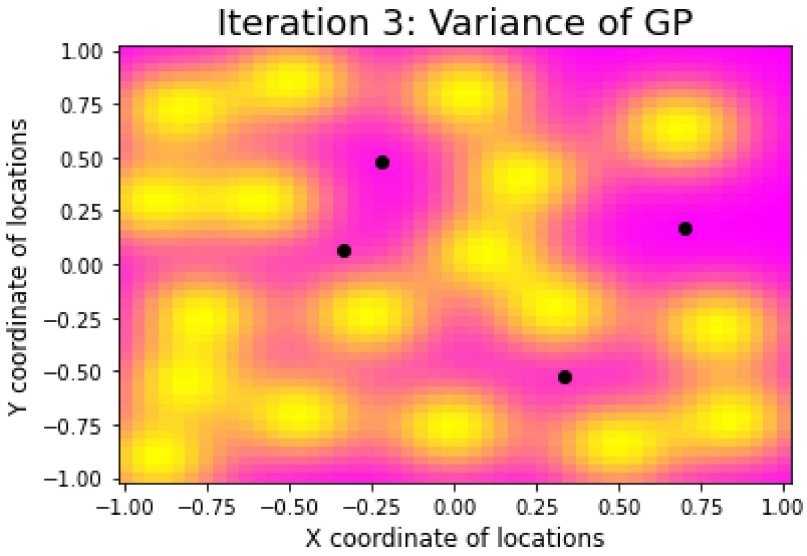
In [16]:
```
const = ((-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1))
results = scipy.optimize.minimize(var_from_trace, (0.835, -0.814, -0.830,  0.682,
print(results)
```

```
      fun: 99.51065605000997
 hess_inv: <8x8 LbfgsInvHessProduct with dtype=float64>
      jac: array([ 3.55271368e-05,  8.38440428e-05, -1.03739239e-04, -4.2632564
1e-06,
        0.00000000e+00, -1.22213351e-04,  4.68958206e-05,  1.32160949e-04])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 297
      nit: 23
   status: 0
  success: True
        x: array([ 0.33062188, -0.33954949, -0.22327942,  0.69817503, -0.529005
06,
        0.07033031,  0.48298352,  0.16810652])
```

In [17]:
```
var = np.diag(cov_out)
var_to_plot = np.reshape(var, (50,50))
print(var_to_plot)
c = plt.pcolormesh(xbar,ybar,var_to_plot, cmap = "spring_r")
x_values = [(results.x[0],results.x[1],results.x[2],results.x[3])]
y_values = [(results.x[4],results.x[5],results.x[6],results.x[7])]
plt.scatter(x_values, y_values, c = 'black')
plt.title('Iteration 3: Variance of GP', fontsize = 18)
plt.xlabel('X coordinate of locations', fontsize = 12)
plt.ylabel('Y coordinate of locations', fontsize = 12)
```

```
[[0.03444706 0.02399315 0.0185957  ... 0.09031155 0.09375728 0.09761833]
 [0.02477996 0.01306069 0.00700117 ... 0.07885072 0.08394578 0.08975088]
 [0.02013286 0.00773996 0.00125646 ... 0.06485196 0.07194113 0.08012329]
 ...
 [0.08226122 0.0742602  0.06713125 ... 0.10556633 0.10809398 0.10996824]
 [0.09140151 0.08575699 0.08065122 ... 0.10846808 0.11000815 0.11115009]
 [0.09880109 0.0950646  0.09161658 ... 0.11041251 0.11129079 0.11194202]]
```

Out[17]: Text(0, 0.5, 'Y coordinate of locations')



In [ ]:

In [15]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
from matplotlib import cm
from matplotlib.ticker import LinearLocator
```

In [16]:
```
%matplotlib inline
import numpy as np
import scipy.stats as stats
import scipy.special as scispec
import scipy.optimize
import matplotlib.pyplot as plt
import sympy
from sympy.utilities.lambdify import lambdify
from scipy import optimize
from scipy.optimize import minimize
```

In [17]:
```
def build_covariance(x, xp, y, yp, func, tau, L):
    """Build a covariance matrix

    Inputs
    -------
    x: (N) array of inputs
    xp: (M) array of inptus
    kern: a function mapping inputs to covariance

    Outputs
    -------
    cov: (N, M) covariance matrix
    """
    kern = func
    out = np.zeros((x.shape[0], xp.shape[0]))
    for jj in range(xp.shape[0]):
        out[:, jj] = kern(x, xp[jj], y, yp[jj], tau, L)
    return out
```

In [18]:
```
def sqexp(x, xp, y, yp, tau, L):
    """Squared exponential kernel (2 dimensional)

    Inputs
    ------
    x : (N), array of multiple inputs
    xp: float
    y : (N), array of multiple inputs
    yp: float

    Returns
    -------
    cov (N,) -- Covariance between each input at *x* and the function values at *
    """
    cov = tau**2 * np.exp(-1/2 * (x - xp)**2 / L**2)*tau**2 * np.exp(-1/2 * (y -
    return cov
```

In [19]:
```
# constructing gridspace
n_div = 50
x = np.linspace(-1, 1, n_div)
y = np.linspace(-1, 1, n_div)
xbar, ybar = np.meshgrid(x,y)
xbar_new = np.reshape(xbar,(n_div**2))
ybar_new = np.reshape(ybar,(n_div**2))
ybar_new = ybar_new.T
print(xbar_new.shape)
#print(xbar_new)
print(ybar_new.shape)
#print(ybar_new)
```

```
(2500,)
(2500,)
```

In [20]:
```
# tau = 0.4
# L = 0.4
# sigma = 0.01
n_init = 22
xinit = np.array([0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9, -0.771,
xinit = xinit.T
yinit = np.array([0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9, -0.2
yinit = yinit.T
sen_out = np.array([3.39382006, 3.2073034, 3.39965035, 3.68810201, 2.96941623, 2.
print((sen_out.T).shape)
sen_out_new = np.reshape(sen_out,(-1,1))
#mean_func = lambda x: np.zeros(((x.shape[0]),1))
#kernel = sqexp(xinit, xbar_new, yinit, ybar_new, tau, L)
#print(kernel)
print(sen_out_new.shape)
```

```
(22,)
(22, 1)
```

In [21]:
```
def max_marg_likelihood(des_var):
    tau, L, sigma = des_var
    cov = build_covariance(xinit, xinit, yinit, yinit, sqexp, tau, L)

    mean_func = np.mean(sen_out)*np.ones((n_init,1))

    subterm_1 = (sen_out_new - mean_func).T
    subterm_2 = np.linalg.pinv(cov + (sigma**2)*np.identity(xinit.shape[0]))
    subterm_3 = sen_out_new - mean_func

    term1 = np.dot(subterm_1,np.dot(subterm_2,subterm_3))
    term2 = np.log(np.linalg.det((cov + (sigma**2)*np.identity(xinit.shape[0]))))
    term3 = (n_init)*np.log(2*np.pi)

    mx_marg_lh = (1/2)*(term1 + term2 + term3)

    return mx_marg_lh
```

In [22]:
```
const = ((0,None),(0,None),(0,None))
#results = scipy.optimize.minimize(max_marg_likelihood, (0.65, 0.45, 0.01), bound
#results = scipy.optimize.minimize(max_marg_likelihood, (0.95, 0.5, 0.01), bounds
hyperParameters = scipy.optimize.minimize(max_marg_likelihood, (5.79779452e-01, 2
# results = scipy.optimize.minimize(max_marg_likelihood, (0.7, 0.4, 0.01), bounds
#results = scipy.optimize.minimize(max_marg_likelihood, (0.1, 0.45, 0.001), bound
# print(results)
print(hyperParameters)
```

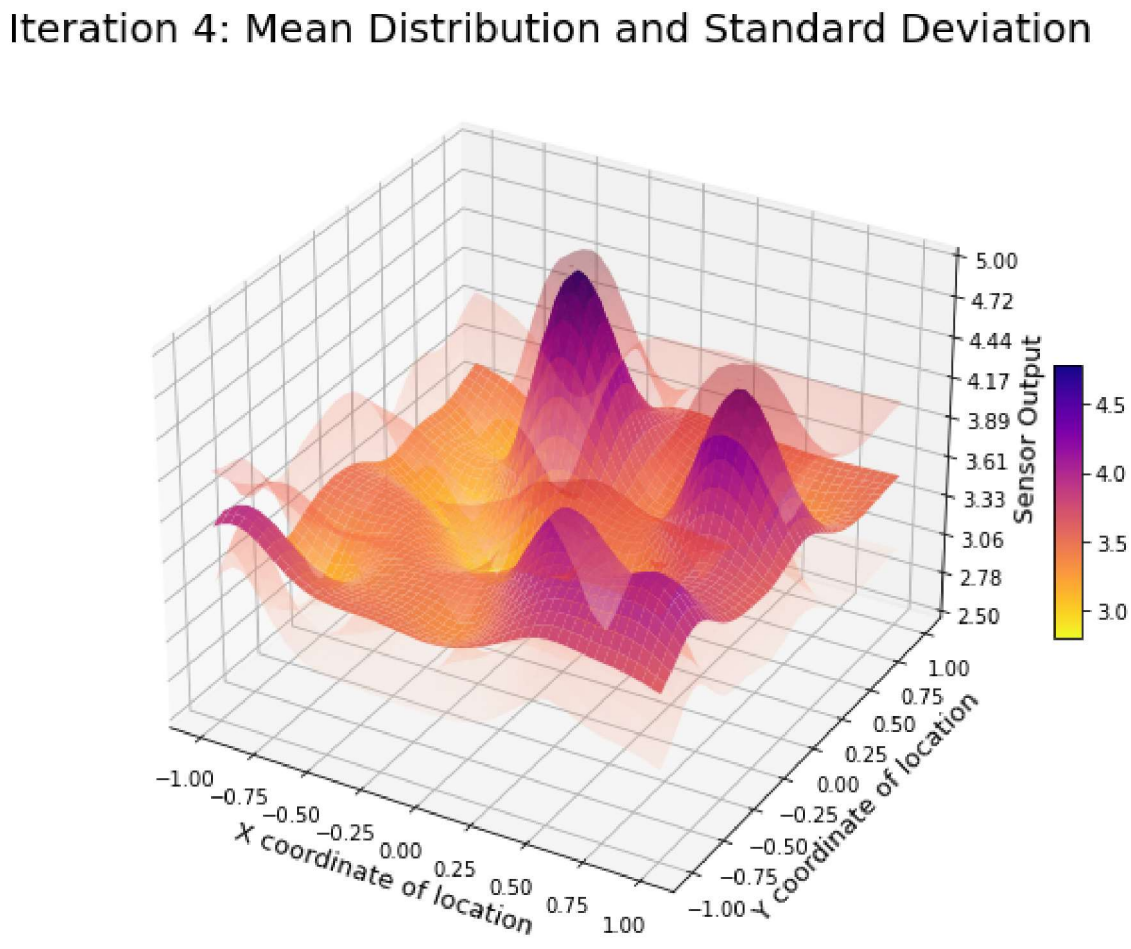
```
      fun: array([[16.78847233]])
 hess_inv: <3x3 LbfgsInvHessProduct with dtype=float64>
      jac: array([-1.49213975e-05,  1.84741111e-05,  1.42108547e-06])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 52
      nit: 10
   status: 0
  success: True
        x: array([7.21974093e-01, 1.61545364e-01, 1.56412591e-06])
```

In [23]:
```
def gpr(xinit, yinit, xpred, ypred, out_sensor, noise_var, mean_func, kernel, tau
    """Gaussian process regression Algorithm

    Inputs
    -------
    xinit: (N, ) training inputs
    yinit: (N, ) training outputs
    xpred:  (M, ) locations at which to make predictions
    ypred:  (M, ) locations at which to make predictions
    noise_var: (N, ) noise at every training output
    mean_func: function to compute the prior mean
    kernel: covariance kernel

    Returns
    -------
    pred_mean : (M, ) predicted mean at prediction points
    pred_cov : (M, M) predicted covariance at the prediction points
    --
    """
    # constructing covariance matrix from initial datapoints given
    cov = build_covariance(xinit, xinit, yinit, yinit, kernel, tau, L)
    u, s, v = np.linalg.svd(cov)
    sqrtcov = np.dot(u, np.sqrt(np.diag(s)))

    # pseudoinverse is better conditioned
    invcov = np.linalg.pinv(cov + np.diag(noise_var))
    mean_func = np.mean(sen_out)*np.ones((n_init,1))
    mean_func = np.mean(sen_out)
    vec_pred = build_covariance(xpred, xinit, ypred, yinit, kernel, tau, L)
    #pred_mean = mean_func + np.dot(vec_pred, np.dot(invcov, out_sensor - mean_fu
    pred_mean = mean_func*np.ones(xpred.shape[0]) + np.matmul(vec_pred, np.matmul

    cov_predict_pre = build_covariance(xpred, xpred, ypred, ypred, kernel, tau, L
    cov_predict_up = np.dot(vec_pred, np.dot(invcov, vec_pred.T))
    pred_cov = cov_predict_pre - cov_predict_up

    return pred_mean, pred_cov
```

In [34]:
```
tau = hyperParameters.x[0]
L = hyperParameters.x[1]
sigma = hyperParameters.x[2]
mean_func = np.mean(sen_out)*np.ones((n_init,1))
#mean_func = lambda x: np.zeros((x.shape[0]))
xinit = np.array([0.1, -0.9, 0.2, 0.8, -0.6, 0.3, 0.5, -0.5, -0.01, -0.9, -0.771,
yinit = np.array([0.05, 0.3, 0.4, -0.3, 0.3, -0.2, -0.84, 0.85, -0.76, -0.9, -0.2
sen_out = np.array([3.39382006, 3.2073034, 3.39965035, 3.68810201, 2.96941623, 2.
sigma_opt = sigma * np.ones((xinit.shape[0]))
print(sen_out.shape)
mean_out, cov_out = gpr(xinit, yinit, xbar_new, ybar_new, sen_out, sigma_opt, mea
x = np.max(mean_out)
print(x)
```

```
(22,)
4.799556294718048
```

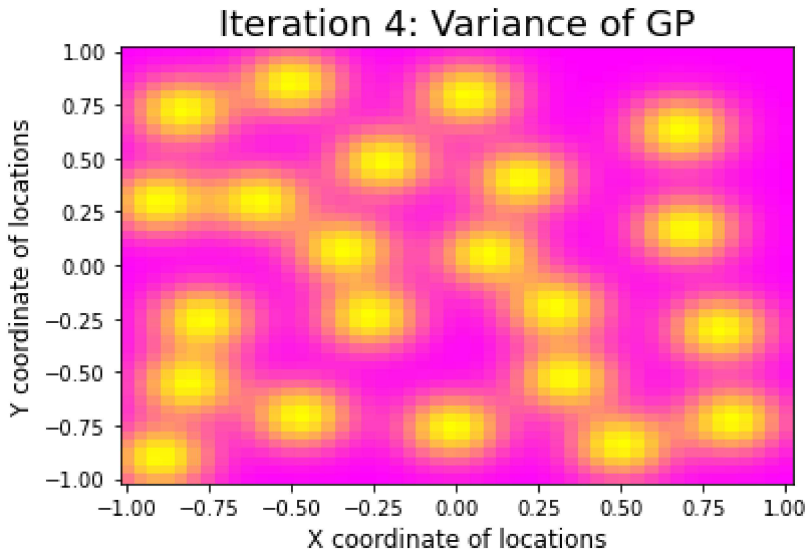
In [28]:
```
fig, ax = plt.subplots(subplot_kw={"projection": "3d"}, figsize = (10,10))
# Plot the surface.
pred_mean = np.reshape(mean_out, (50,50))
#print(pred_mean)
mean = ax.plot_surface(xbar, ybar, pred_mean, cmap='plasma_r')
var = np.sqrt(np.diag(cov_out))
var_to_plot = np.reshape(var, (50,50))
#print(var_to_plot.shape)
#print(pred_mean.shape)
var_1 = ax.plot_surface(xbar, ybar, pred_mean + var_to_plot, cmap='Reds', alpha =
var_2 = ax.plot_surface(xbar, ybar, pred_mean - var_to_plot, cmap='Reds', alpha =
#
# Customize the z axis.
ax.set_zlim(2.5, 5)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.set_xlabel('X coordinate of location', fontsize = 13)
ax.set_ylabel('Y coordinate of location', fontsize = 13)
ax.set_zlabel('Sensor Output', fontsize = 13)
# A StrMethodFormatter is used automatically
ax.zaxis.set_major_formatter('{x:.02f}')
ax.set_title('Iteration 4: Mean Distribution and Standard Deviation', fontsize =


# Add a color bar which maps values to colors.
fig.colorbar(mean, shrink=0.25, aspect=10)

plt.show()
```



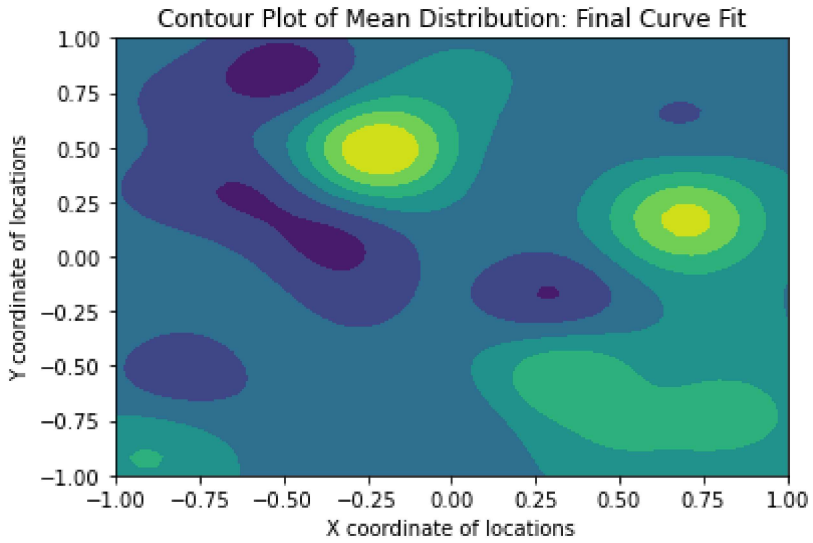
In [32]:
```
var = np.diag(cov_out)
var_to_plot = np.reshape(var, (50,50))
print(var_to_plot)
c = plt.pcolormesh(xbar,ybar,var_to_plot, cmap = "spring_r")
#x_values = [(results.x[0],results.x[1],results.x[2],results.x[3])]
#y_values = [(results.x[4],results.x[5],results.x[6],results.x[7])]
#plt.scatter(x_values, y_values, c = 'black')
plt.title('Iteration 4: Variance of GP', fontsize = 18)
plt.xlabel('X coordinate of locations', fontsize = 12)
plt.ylabel('Y coordinate of locations', fontsize = 12)
```

```
[[0.14482072 0.10900712 0.08809719 ... 0.25996256 0.26320941 0.26628638]
 [0.10919445 0.06335306 0.03660603 ... 0.24576008 0.25292058 0.2597272 ]
 [0.08859058 0.03694484 0.00679688 ... 0.22121828 0.23513868 0.2483936 ]
 ...
 [0.25111359 0.23889702 0.22568548 ... 0.27028795 0.27106646 0.27144905]
 [0.26125678 0.25506134 0.24834518 ... 0.2711562  0.27145534 0.27160235]
 [0.26703702 0.26427066 0.26125844 ... 0.27151478 0.27161594 0.27166566]]
```

Out[32]: Text(0, 0.5, 'Y coordinate of locations')



In [38]:
```
fig, ax = plt.subplots(1, 1)

ax.contourf(xbar, ybar, pred_mean)

ax.set_title('Contour Plot of Mean Distribution: Final Curve Fit')
ax.set_xlabel('X coordinate of locations')
ax.set_ylabel('Y coordinate of locations')

plt.show()
```



In [ ]: