# AE 6310-A: Assignment #4

Karl Roush

Due April 21, 2020; Submission up to April 28, 2020 for no penalty

## Contents

# Problem 1: Surrogate Modeling

We are given the following "black box" function defined on $x_i \in [-1,1]$

$$f(x_1, x_2) = {x_1}^2 - {x_2}^2 - \cos(\frac{\pi}{2}x_1)\cos(\frac{\pi}{2}x_2)$$

The DoE is also defined as follows.

$$x_1 = -1 + \frac{2(i-1)}{M-1}; \quad x_2 = -1 + \frac{2(j-1)}{M-1}$$

$$Number\ of\ points = M^2; \quad i,j = 1,\dots,M$$

We will be considering the cases of M=3 and M=5 which produce 9 and 25 points, respectively.

Table 1: M=3 case (N=9 points)

| Point | X1 | X2 | Point | X1 | X2 | Point | X1 | X2 |
|---|---|---|---|---|---|---|---|---|
| **1** | -1 | -1 | **4** | -1 | 0 | **7** | -1 | 1 |
| **2** | 0 | -1 | **5** | 0 | 0 | **8** | 0 | 1 |
| **3** | 1 | -1 | **6** | 1 | 0 | **9** | 1 | 1 |

Table 2: M=5 case (N=25 points)

| Point | X1 | X2 | Point | X1 | X2 | Point | X1 | X2 |
|---|---|---|---|---|---|---|---|---|
| **1** | -1 | -1 | **11** | -1 | 0 | **21** | -1 | 1 |
| **2** | -0.5 | -1 | **12** | -0.5 | 0 | **22** | -0.5 | 1 |
| **3** | 0 | -1 | **13** | 0 | 0 | **23** | 0 | 1 |
| **4** | 0.5 | -1 | **14** | 0.5 | 0 | **24** | 0.5 | 1 |
| **5** | 1 | -1 | **15** | 1 | 0 | **25** | 1 | 1 |
| **6** | -1 | -0.5 | **16** | -1 | 0.5 | | | |
| **7** | -0.5 | -0.5 | **17** | -0.5 | 0.5 | | | |
| **8** | 0 | -0.5 | **18** | 0 | 0.5 | | | |
| **9** | 0.5 | -0.5 | **19** | 0.5 | 0.5 | | | |
| **10** | 1 | -0.5 | **20** | 1 | 0.5 | | | |

## Part 1: Quadratic Basis Functions

We are provided the following set of six quadratic basis functions:

$$\phi = \{1,\ x_1,\ x_2,\ {x_1}^2,\ x_1x_2,\ {x_2}^2\}$$

Note that the subscript $\phi_m$ refers to the m index of the set of basis functions, *starting from an index of zero*. This is done since Python indices start at zero. For example, the fourth basis function is defined as m=3 seen below.

$$\phi_3 = {x_1}^2$$

### Part 1A: Finding the weights

Finding the weights associated with associated surrogate function (generated from the basis functions) is an unconstrained optimization problem, defined through the equation below.

$$\Phi^T \Phi w = \Phi^T f$$

Note that f represents the values of the black box function at our sample points. Furthermore, $\Phi$ is a matrix of the basis functions evaluated at a given sample point (rows= given sample point, columns= specific basis function). This is represented below.

$$\Phi = \begin{bmatrix} \phi_1(point\ 1) & \phi_2(point\ 1) & \cdots & \phi_m(point\ 1) \\ \phi_1(point\ 1) & \phi_2(point\ 2) & \cdots & \phi_m(point\ 2) \\ \vdots & \vdots & \vdots & \vdots \\ \phi_1(point\ N) & \phi_2(point\ N) & \cdots & \phi_m(point\ N) \end{bmatrix}$$

Therefore, the process to find the weights is as follows: calculate the black box function values at the sample points, build the $\Phi$ matrix by evaluating the basis functions at the sample points, and then solve the linear algebra problem discussed previously.

The calculated weights are summarized in the table below.

Table 3: Quadratic basis function weights

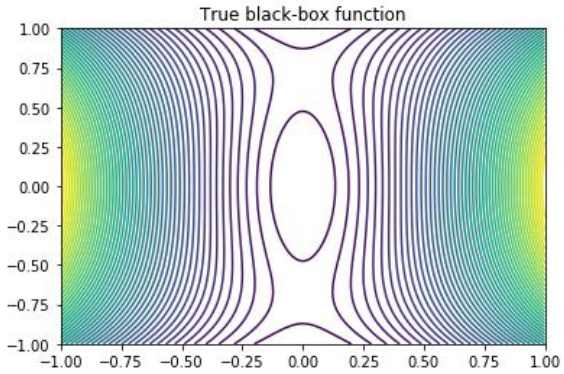
| Basis function index | Basis function | Weight (M=3) | Weight (M=5) |
|---|---|---|---|
| 0 | 1 | -0.5556 | -0.7041 |
| 1 | X1 | 0 | 8.8818e-18 |
| 2 | X2 | -6.2490e-34 | -2.9995e-34 |
| 3 | X1**2 | 1.3333 | 1.4710 |
| 4 | X1*X2 | 0 | -5.9990e-34 |
| 5 | X2**2 | -0.6667 | -0.5290 |

Directly from Python:

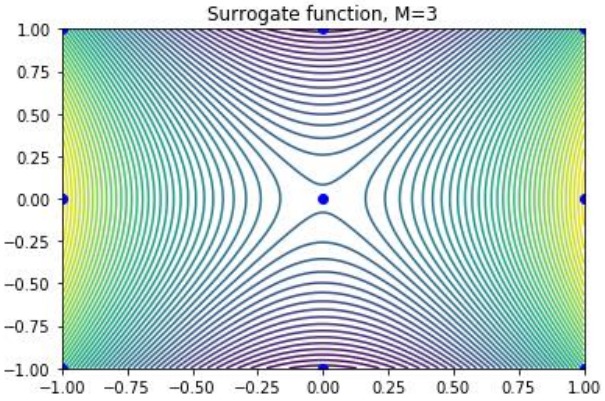
```
M=3 weights=  [-5.55555556e-01  0.00000000e+00 -6.24899909e-34  1.33333333e+00
  0.00000000e+00 -6.66666667e-01]


M=5 weights=  [-7.04145124e-01  8.88178420e-18 -2.99951957e-34  1.47100804e+00
 -5.99903913e-34 -5.28991961e-01]
```

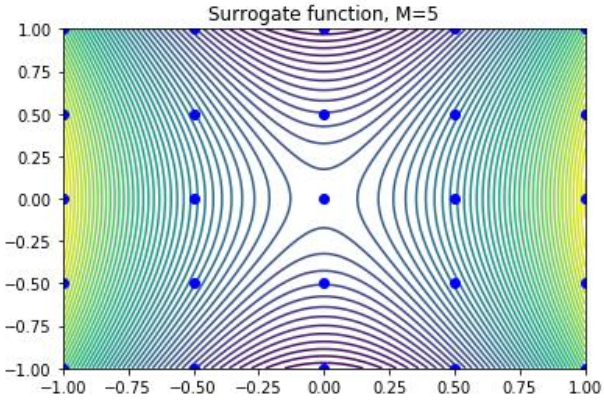
## Part 1B: Plotting the error

A contour plot of the true function f(x) is seen below.




The contour plots of the surrogate models are seen below.

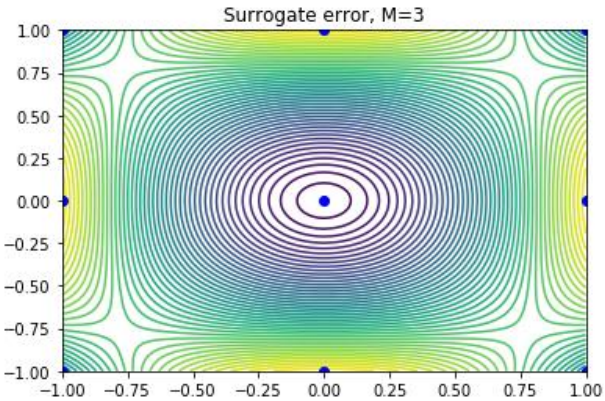


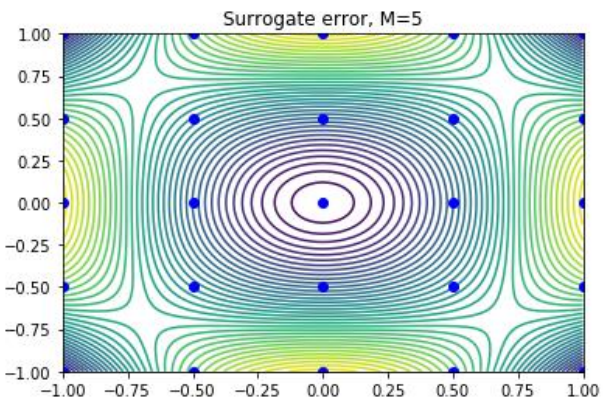




The error of the model is defined as the difference between the true function and the surrogate model :

$$\epsilon(x) = f(x) - \hat{f}(x)$$

Applying this concept to the two surrogate models yields the following contour plots for their error.



## Part 1C: Validation with 100 random points and $R^2$ values

In order to validate the model, 100 random points were chosen. The true function value and the surrogate function model were calculated at these points. The $R^2$ is then calculated as follows:

$$R^2 = 1 - \frac{SSE}{SST} = 1 - \frac{(y-\hat{y})^T(y-\hat{y})}{(y-\bar{y})^T(y-\bar{y})}$$

Since the 100 points are chosen at random, the $R^2$ value will vary slightly. This is because the surrogate model matches the true function in some places better than others. Therefore, if the random points are close to this "better match" the $R^2$ will be closer to 1. However, these 100 points are random so the $R^2$ will not be the same each time. The results from three sets of 100 random points are summarized in the table below.

Table 4: $R^2$ values for quadratic basis surrogate models

| Surrogate model | Test case | $R^2$ value |
|---|---|---|
| M= 3 (9 DoE points) | Initial DoE points | 0.87705 |
| | 100 random points, run 1 | 0.90632 |
| | 100 random points, run 2 | 0.88565 |
| | 100 random points, run 3 | 0.85990 |
| | ***Average of runs w/ random points*** | ***0.88396*** |
| M=5 (25 DoE points) | Initial DoE points | 0.94152 |
| | 100 random points, run 1 | 0.94326 |
| | 100 random points, run 2 | 0.92737 |
| | 100 random points, run 3 | 0.93553 |
| | ***Average of runs w/ random points*** | ***0.93539*** |

Note that as the number of points used to build the surrogate model increase, the $R^2$ value also increases. This indicates that having more points for the surrogate model increases its quality within the range of the data set.

## Part 2: Gaussian Radial Basis Function

The Gaussian radial basis function is defined below.

$$\phi(r) = e^{\frac{-r^2}{2{r_0}^2}}; \quad r_0 = 1$$

The only term that varies for the set of basis functions is r, which is defined as the 2-norm, seen below.

$$r = \left\|x_j - x_i\right\|_2 = \sqrt{(x_{j,1} - x_{i,1})^2 + \ldots + (x_{j,k} - x_{i,k})^2}$$

Combining the two equations from above allows us to create the surrogate model of the form:

$$\hat{f}(x_j) = \sum_i^n w_i \phi(\left\|x_j - x_i\right\|_2)$$

In both of the following plots, the original constraint boundary is marked by a red circle. As rho increases, the minimizer approaches the constrained minimizer from the infeasible space.

### Part 2A: Finding the weights

The process for finding the weights of the basis functions is essentially the same as in Part 1A: Finding the weights. However, in this case we are using an interpolation model so the number of basis functions is equal to the number of sample points (N=m=$M^2$).

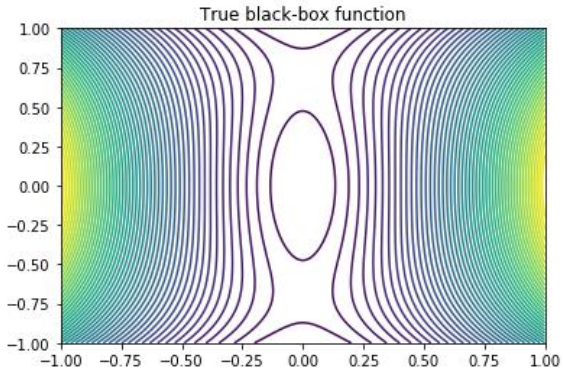
Table 5: Gaussian radial basis weights for M=3 case (N=9 points)

| $\phi_m$ | Weight | $\phi_m$ | Weight | $\phi_m$ | Weight |
|---|---|---|---|---|---|
| **0** | -2.3041 | **3** | 6.8156 | **6** | -2.3041 |
| **1** | 1.8103 | **4** | -8.0733 | **7** | 1.8103 |
| **2** | -2.3041 | **5** | 6.8156 | **8** | -2.3041 |

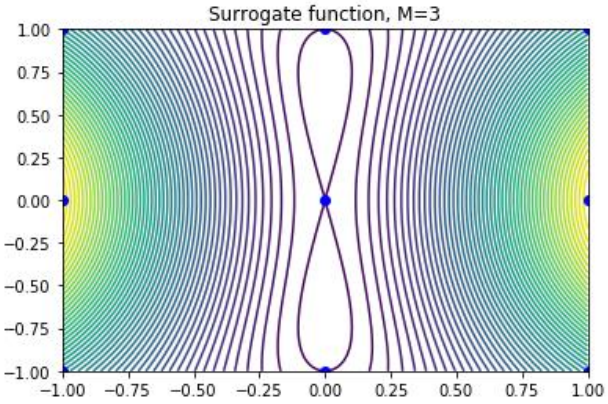
Table 6: Gaussian radial basis weights for M=5 case (N=25 points)

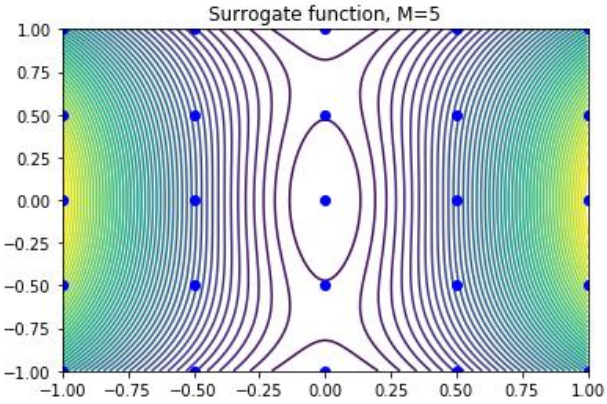
| $\phi_m$ | Weight | $\phi_m$ | Weight | $\phi_m$ | Weight |
|---|---|---|---|---|---|
| **0** | -41.7714 | **10** | -129.3368 | **20** | -41.7714 |
| **1** | 71.8282 | **11** | 226.0319 | **21** | 71.8282 |
| **2** | -86.3510 | **12** | -278.4274 | **22** | -86.3510 |
| **3** | 71.8282 | **13** | 226.0319 | **23** | 71.8282 |
| **4** | -41.7714 | **14** | -129.3368 | **24** | -41.7714 |
| **5** | 115.1166 | **15** | 115.1166 | | |
| **6** | -209.1645 | **16** | -209.1645 | | |
| **7** | 256.6154 | **17** | 256.6154 | | |
| **8** | -209.1645 | **18** | -209.1645 | | |
| **9** | 115.1166 | **19** | 115.1166 | | |

## Part 2B: Plotting the error




The contour plots of the surrogate models are seen below.

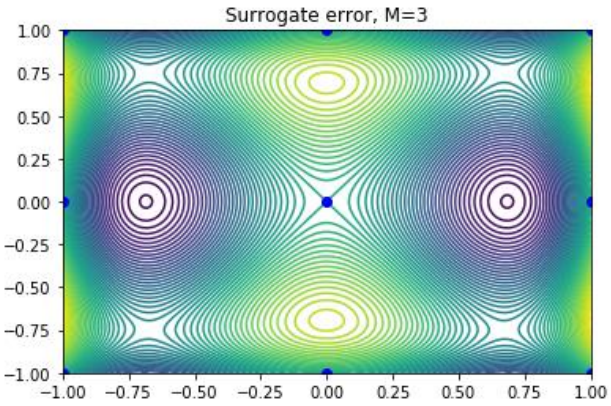


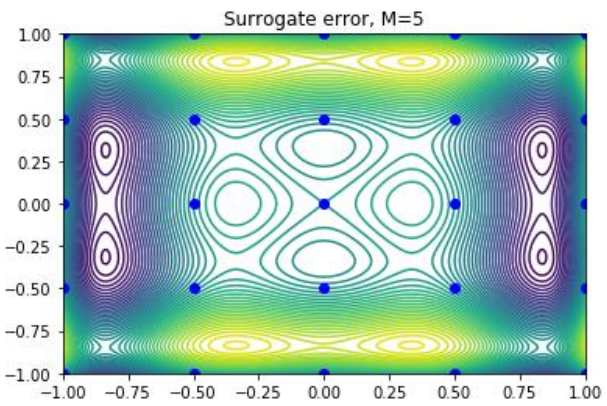




The error of the model is defined as the difference between the true function and the surrogate model :

$$\epsilon(x) = f(x) - \hat{f}(x)$$

Applying this concept to the two surrogate models yields the following contour plots for their error.



## Part 2C: Validation with 100 random points and $R^2$ values

Following the same process detailed in Part 1C: Validation with 100 random points and $R^2$ values, produces the results summarized in the table below.

Table 7: $R^2$ values for Gaussian radial basis surrogate models

| Surrogate model | Test case | $R^2$ value |
|---|---|---|
| M= 3 (9 DoE points) | Initial DoE points | 0.98485 |
| | 100 random points, run 1 | 0.98605 |
| | 100 random points, run 2 | 0.98602 |
| | 100 random points, run 3 | 0.98014 |
| | ***Average of runs w/ random points*** | ***0.98407*** |
| M=5 (25 DoE points) | Initial DoE points | 0.99988 |
| | 100 random points, run 1 | 0.99989 |
| | 100 random points, run 2 | 0.99985 |
| | 100 random points, run 3 | 0.99988 |
| | **Average of runs w/ random points** | **0.99987** |

Again, note that as the number of points used to build the surrogate model increase, the $R^2$ value also increases. However, as the number of DoE points increase, the run time of the program also significantly increases, so one should consider that when deciding on the number of basis functions.

Note that because this is an interpolating model, we would expect the $R^2$ value for the initial DoE points to be 1. Additionally, the sample points should all be on the same error contour (since the error should be approximately zero). Upon speaking with Prof. Kennedy in office hours, it was unclear what is causing this error- he directed me to continue with these results.

# Problem 2: Gradient and Gradient Free Optimizers

For this section, we are provided the following function:

$$f(x_1, x_2) = |x_1 x_2| + 0.1({x_1}^2 + {x_2}^2)$$

## Problem 2A: Minimizer

From inspection, the minimum of this function is located at [x1, x2]= [0,0].

For this section, I will be using the scipy.optimize.minimize function. This function offers several different methods. Two gradient-based methods offered are BFGS and L-BFGS. Two gradient-free methods offered are Powell and Nelder-Mead. These results with default tolerance= none are summarized in the table below.

Table 8: Default tolerance minimization by method

| Parameter | BFGS (Gradient) | Nelder-Mead (Gradient-free) | Powell (Gradient-free) |
|---|---|---|---|
| Iterations | 2 | 64 | 2 |
| Function evaluations | 12 | 119 | 37 |
| Minimum [x1, x2] | [-2.74055362e-09, -1.17159193e-09] | [4.69330301e-05, -1.90855003e-05] | [0., 0.] |

## Problem 2B: Gradient based optimizer tolerancing

These methods are then checked with varying tolerances. The gradient based optimizer does exit successfully, though all instances in the table for the BFGS method threw a "Desired error not necessarily achieved due to precision loss". However, the number of function evaluations for the gradient method always exceeded the number of function evaluations for the gradient-free methods, by a significant margin.

Table 9: Varying tolerance minimization by method

| Tolerance | Parameter | BFGS (Gradient) | Nelder-Mead (Gradient-free) | Powell (Gradient-free) |
|---|---|---|---|---|
| 1e-30 | Iterations | 2 | - | 2 |
| | Function evaluations | 408 | - | 59 |
| | Success | True | False, max function evals exceeded | True |
| 1e-20 | Iterations | 2 | - | 2 |
| | Function evaluations | 408 | - | 59 |
| | Success | True | False, max function evals exceeded | True |
| 1e-15 | Iterations | 2 | 169 | 2 |
| | Function evaluations | 408 | 319 | 59 |
| | Success | True | True | True |
| 1e-10 | Iterations | 2 | 125 | 2 |
| | Function evaluations | 408 | 234 | 51 |
| | Success | True | True | True |

## Problem 2C: Accuracy comparison

Referring to Table 8, the gradient based method (BFGS) was more accurate than the gradient-free method (Nelder-Mead).

## Problem 2D: Function evaluation cost comparison

As discussed in Problem 2B: Gradient based optimizer tolerancing, the number of function evaluations for the gradient method always exceeded the number of function evaluations for the gradient-free methods, by a significant margin. This is likely due to the finite difference approximation made for the gradient.

# Code snippets

## Problem 1, Part 1 (function, DoE, Quadratic Basis)

```python
def box_func(x):
    '''This is the black box function'''
    #col1= x1, col2=x2
    return x[0]**2 -x[1]**2 -math.cos(0.5*math.pi*x[0])*math.cos(0.5*math.pi*x[1])

def create_DoE_points(M):
    '''create the sample points'''
    DoE_points= [] #num points=M**2; col1= x1, col2=x2
    j=0
    while j<M:
        for i in range(M):
            x1= -1 + ((2*i)/(M-1))
            x2= -1 + ((2*j)/(M-1))
            DoE_points.append([x1,x2])
            # print('[%d, %d]'%(x1,x2))
            # print('i= %d, j= %d'%(i,j))
        j+=1
    DoE_points= np.array(DoE_points)
    return DoE_points

def phi_quadratic(xi):
    '''defines the basis functions'''
    #xi are the sample points
    #xi[0]= x1, xi[1]= x2
    phi0= 1
    phi1= xi[0]
    phi2= xi[1]
    phi3= xi[0]**2
    phi4= xi[0]*xi[1]
    phi5= xi[1]**2
    quad_basis= [phi0,phi1,phi2,phi3,phi4,phi5]
    return quad_basis
```

## Problem 1, Part 1 (Quadratic basis surrogate function)

```python
def surrogate_quad(xi, func):
    '''constructs the surrogate function from the function value at sample points
    and the basis function values at those points'''
    #xi are the sample points
    #xi[0]= x1, xi[1]= x2
    N= xi.shape[0] #number of sample points

    f = np.zeros(N) #black box function evaluated at sample points
    Phi = np.zeros((N, 6)) #6 quad basis funs w/ N rows (evaluated at N pts)

    for i in range(N): #calc values of black box func @sample pts
        f[i] = func(xi[i,:])
        for j in range(6): #6 quad basis
            # Evaluate the j-th basis function at the point xi[i,:]
            #technically calcs values of all basis funcs @ each pt
            basis_evals= phi_quadratic(xi[i,:])
            Phi[i,j] = basis_evals[j]

    #now we want to solve the unconstrained opt problem
    weights= np.linalg.solve(np.dot(Phi.T, Phi), np.dot(Phi.T, f))
    return weights

def eval_surrogate_quad(x, w):
    '''evualates the surrogate function at a desired point'''
    unweight_vals= phi_quadratic(x)
    fhat= np.dot(w,unweight_vals)

    return fhat
```

## Problem 1, Part 1 (Plots, Validation)

```python
def check100_pts(w,func):
    '''validates the model with 100 random points'''
    npts= 100
    N=100

    #adapted from provided radial_basis_function.py
    X = -1.0 + 2.0*np.random.uniform(size=(N, 1)) #between -1,1
    Y = -1.0 + 2.0*np.random.uniform(size=(N, 1))
    X, Y = np.meshgrid(X, Y)
    F = np.zeros((npts, npts))
    Fhat = np.zeros((npts, npts))

    for j in range(npts):
        for i in range(npts):
            xpt = np.array([X[i,j], Y[i,j]])
            F[i,j] = box_func(xpt)
            Fhat[i,j] = func(xpt, w)

    # Evaluate the R2 value (coefficient of determination)
    SSE = np.sum((F - Fhat)**2)
    SST = np.sum((F - np.average(F))**2)

    R2 = 1.0 - SSE/SST
    # R2= round(R2,5)
    print('R2, 100 random = ', R2)
```

```python
def error_contours(xi,w,M,func):
    '''plots the contours of the function, surrogate model, and error'''
    #adapted from provided radial_basis_function.p
    npts = 250
    X = np.linspace(-1, 1, npts)
    X, Y = np.meshgrid(X, X)
    F = np.zeros((npts, npts))
    Fhat = np.zeros((npts, npts))

    for j in range(npts):
        for i in range(npts):
            xpt = np.array([X[i,j], Y[i,j]])
            F[i,j] = box_func(xpt)
            Fhat[i,j] = func(xpt, w)

    # Evaluate the R2 value (coefficient of determination)
    SSE = np.sum((F - Fhat)**2)
    SST = np.sum((F - np.average(F))**2)

    R2 = 1.0 - SSE/SST
    # R2= round(R2,5)
    print('R2, sample pts = ', R2)

    plt.figure()
    plt.contour(X, Y, F, levels=50)
    plt.title('True black-box function')

    plt.figure()
    plt.contour(X, Y, Fhat, levels=50)
    plt.plot(xi[:,0], xi[:,1], 'ob')
    plt.title('Surrogate function, M=%d'%(M))

    plt.figure()
    plt.contour(X, Y, F - Fhat, levels=50)
    plt.plot(xi[:,0], xi[:,1], 'ob')
    plt.title('Surrogate error, M=%d'%(M))

    plt.show()
```

## Problem 1, Part 2 (Gaussian radial basis surrogate function)

```python
def phi_radialBasis(r):
    '''returns the radial basis function value, given r'''
    #e^(-r**2/(2*r0**2)); r0=1
    return math.exp(-0.5*(r**2))

def surrogate_radial(xi, func):
    '''creates radial basis surrogate function given the sample points and
    black box function'''
    #adapted from provided radial_basis_function.py
    N = xi.shape[0] #number of points

    f = np.zeros(N) #true function values at sample points
    Phi = np.zeros((N, N)) #basis function values at sample points

    # Set the values into f and Phi
    for i in range(N):
        f[i] = func(xi[i,:])

        # Place the basis function values in row i of
        # the Phi matrix
        for j in range(N):
            # Evaluate the j-th basis function at the point xi[i,:]
            r = np.sqrt(np.dot(xi[i,:] - xi[j,:], xi[i,:] - xi[j,:]))
            Phi[i,j] = phi_radialBasis(r)
    # Solve Phi^{T}*Phi*w = Phi^{T}*f, but because Phi is
    # square, we can solve Phi*w = f instead
    weights= np.linalg.solve(Phi, f)
    return weights

def eval_surrogate_radial(x, xi, w):
    """
    Evaluate the surrogate model at the specified design point.

    Args:
        x: The design point at which to evaluate the surrogate
        xi: The sample points
        w: The surrogate model weights

    Returns:
        The radial basis surrogate function value
    """
    #adapted from provided radial_basis_function.py
    # m = N in this case, since we are using an interpolating model
    N = len(w)
    fhat = 0.0
    for i in range(N):
        # r = ||x - x[i]||_{2}
        r = np.sqrt(np.dot(x - xi[i,:], x - xi[i,:]))
        fhat += w[i]*phi_radialBasis(r)

    return fhat
```

## Problem 1, Part 2 (Plots, Validation)

```python
def radial_graphCalc(xi,w,M):
    # Plot the true function and the black box function
    npts = 250
    X = np.linspace(-1, 1, npts)
    X, Y = np.meshgrid(X, X)
    F = np.zeros((npts, npts))
    Fhat = np.zeros((npts, npts))

    for j in range(npts):
        for i in range(npts):
            xpt = np.array([X[i,j], Y[i,j]])
            F[i,j] = box_func(xpt)
            Fhat[i,j] = eval_surrogate_radial(xpt, xi, w)

    # Evaluate the R2 value (coefficient of determination)
    SSE = np.sum((F - Fhat)**2)
    SST = np.sum((F - np.average(F))**2)

    R2 = 1.0 - SSE/SST
    # R2= round(R2,5)
    print('R2 sample pts = ', R2)

    plt.figure()
    plt.contour(X, Y, F, levels=50)
    plt.title('True black-box function')

    plt.figure()
    plt.contour(X, Y, Fhat, levels=50)
    plt.plot(xi[:,0], xi[:,1], 'ob')
    plt.title('Surrogate function, M=%d'%(M))

    plt.figure()
    plt.contour(X, Y, F - Fhat, levels=50)
    plt.plot(xi[:,0], xi[:,1], 'ob')
    plt.title('Surrogate error, M=%d'%(M))

    plt.show()
```

```python
def check100_pts_radial(w,func,xi):
    '''validates the model with 100 random points'''
    npts= 100
    N=100

    #adapted from provided radial_basis_function.py
    X = -1.0 + 2.0*np.random.uniform(size=(N, 1)) #between -1,1
    Y = -1.0 + 2.0*np.random.uniform(size=(N, 1))
    X, Y = np.meshgrid(X, Y)
    F = np.zeros((npts, npts))
    Fhat = np.zeros((npts, npts))

    for j in range(npts):
        for i in range(npts):
            xpt = np.array([X[i,j], Y[i,j]])
            F[i,j] = box_func(xpt)
            Fhat[i,j] = func(xpt, xi, w)

    # Evaluate the R2 value (coefficient of determination)
    SSE = np.sum((F - Fhat)**2)
    SST = np.sum((F - np.average(F))**2)

    R2 = 1.0 - SSE/SST
    # R2= round(R2,5)
    print('R2, 100 random = ', R2)
```

## Problem 2 (Function definition, tolerance evaluation)

```
14  def obj_func(x):
15      '''returns the value of the main function'''
16      #x[0]= x1, x[1]= x2
17      return abs(x[0]*x[1]) +0.1*(x[0]**2 +x[1]**2)
18
19  def checkTolerance(tolerance,x0):
20      print('\nTolerance= ', tolerance)
21      print('Gradient-based method (BFGS): ')
22      res_grad= sp.minimize(obj_func, x0, method= 'BFGS', tol=tolerance, options={'disp':True})
23      print('Gradient-free method (Nelder-Mead): ')
24      res_gradFree= sp.minimize(obj_func, x0, method= 'Nelder-Mead', tol= tolerance, options={'disp':True})
25      print('Gradient-free method (Powell): ')
26      res_gradFree= sp.minimize(obj_func, x0, method= 'Powell',tol=tolerance,options={'disp':True})
```

## Problem 2 (Running the different methods)

```
32      x0= [-1,1]
33      ## Return minimizer
34      print('-----PART A, find the minimizer-----')
35      print('Gradient-based method (BFGS): ')
36      res_grad= sp.minimize(obj_func, x0, method= 'BFGS', options={'disp':True})
37      print('Min at: ', res_grad.x)
38      print('\nGradient-free method (Nelder-Mead): ')
39      res_gradFree= sp.minimize(obj_func, x0, method= 'Nelder-Mead', options={'disp':True})
40      print('Min at: ', res_gradFree.x)
41      print('\nGradient-free method (Powell): ')
42      res_gradFree= sp.minimize(obj_func, x0, method= 'Powell',options={'disp':True})
43      print('Min at: ', res_gradFree.x)
44
45      ## Check if tighter tolerances modify behavior
46      print('\n-----PART B, check tolerancing behavior-----')
47      tolerance= 1e-30
48      checkTolerance(tolerance,x0)
49      tolerance= 1e-20
50      checkTolerance(tolerance,x0)
51      tolerance= 1e-15
52      checkTolerance(tolerance,x0)
53      tolerance= 1e-10
54      checkTolerance(tolerance,x0)
```