Studienprojekt

Kristoffer Schnieders

Jan 08, 2021

CONTENTS:

CHAPTER ONE

GRID POINTS

Here we show what kind of grids are used for the different quadratures.

Further more, there are two options, how many points are used for the probabilistic quadratures. (one or three points)

These options are compared afterwards.

```
[1]: import matplotlib.pyplot as plt
     import os
     os.chdir("..")
     from Methodes_Studienproject.Studienprojekt_Smolyak_qmc_one_point import *
```

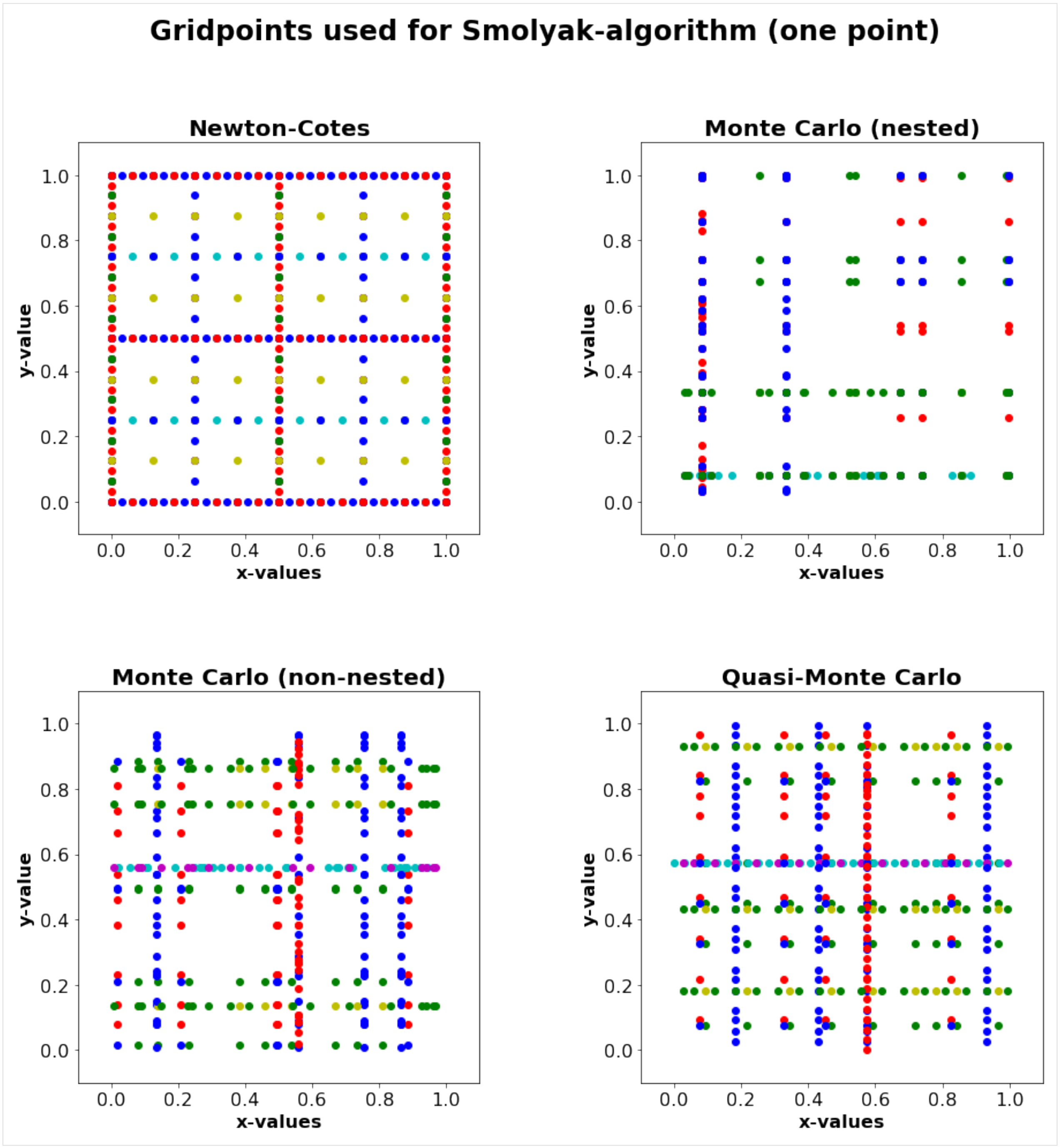
1.1 One point for q = 1

```
[2]: def grid_points(quadrature: str, q: int, a=0,b=1):
         list_of_points = []
         dim = 2

         # Because the points used for the deterministic quadratures are not changed, it␣
     ↪is faster to call them
         # at the beginning before starting the Smolyak alg.
         if "Carlo" in quadrature:
             q = q + 1

         # For every one dimensional degree of approx.  a weights and points vector
         for i in range(1, (q - dim + 2)):

             # One option for Newton-Cotes and Trapezoidal quadrature, because
             if quadrature == "Newton-Cotes" or quadrature == "Trapezoidal":
                 points, weights =one_dim_trapezoidal(i, a, b)

             if quadrature ==  "Monte Carlo (nested)" and i >1:
                 points, weights = monte_carlo_quad(i-1, a, b)
                 points = np.concatenate([list_of_points[-1], points])


             if quadrature ==  "Monte Carlo (non-nested)"or (quadrature == "Monte Carlo␣
     ↪(nested)" and i == 1)   :
                 points, weights = monte_carlo_quad(i, a, b)

             if quadrature ==  "Quasi-Monte Carlo":
                 points, weights = qmc_quad(i, a, b)
```

(continues on next page)

(continued from previous page)

```
        list_of_points.append(points)

    # Now we get the combinations of one-dimensional degrees of approx. in Q(q,d)
    rng = list(range(q)) * dim
    rng = [x + 1 for x in rng]

    possible_combinations = list(set(i for i in itt.permutations(rng, dim) if (q -␣
↪dim) < sum(i) < q + 1))
    number_approx = len(possible_combinations)
    meshgrid_list_x = []
    meshgrid_list_y = []
    for i in range(number_approx):

        # gridpoints and the weight vectors are put into separate lists to make it␣
↪easier to
        # make a meshgrid
        current_tuple = possible_combinations[i]
        current_points = []

        for j in range(len(current_tuple)):
            current_points.append(list_of_points[current_tuple[j] - 1])

        # meshgrid yields a np.array with all coordinates needed for alg
        # coordinate and respective weight could be found in the same places in the␣
↪array
        meshgrid_points = np.array(np.meshgrid(*current_points))
        meshgrid_list_x.append(meshgrid_points[0])
        meshgrid_list_y.append(meshgrid_points[1])

    return [meshgrid_list_x, meshgrid_list_y]

def make_plot_of_grids():
    option_list = ["Newton-Cotes",
                   "Monte Carlo (nested)",
                   "Monte Carlo (non-nested)",
                   "Quasi-Monte Carlo"]
    fig = plt.figure(figsize=(15,15))
    plt.subplots_adjust(hspace=0.4, wspace=0.4)

    for i in range(len(option_list)):
        grid_list_x, grid_list_y = grid_points(option_list[i], 6)
        plt.subplot(2,2,i+1)
        modified_scatter_plot(grid_list_x,grid_list_y,title=option_list[i], input_
↪list=True)
        plt.xticks(fontsize=16)
        plt.yticks(fontsize=16)


    fig.suptitle("Gridpoints used for Smolyak-algorithm (one point)", fontsize=24,
↪fontweight="bold")
    plt.show()

make_plot_of_grids()
```


Gridpoints used for Smolyak-algorithm (one point)
Newton-Cotes
Monte Carlo (nested)
Monte Carlo (non-nested)
Quasi-Monte Carlo
y-value
x-values
0.0
0.2
0.4
0.6
0.8
1.0

1.1.1 Here the grid points used for the approximation of the integral of a 2 dimensional function with the level of approximation q = 6 is shown, if one point is used for the one is used for probabilistic quadratures and q = 1. Apart form the upper left grid, the other grids change a little every time they are generated.

1.2 Three points for q = 1

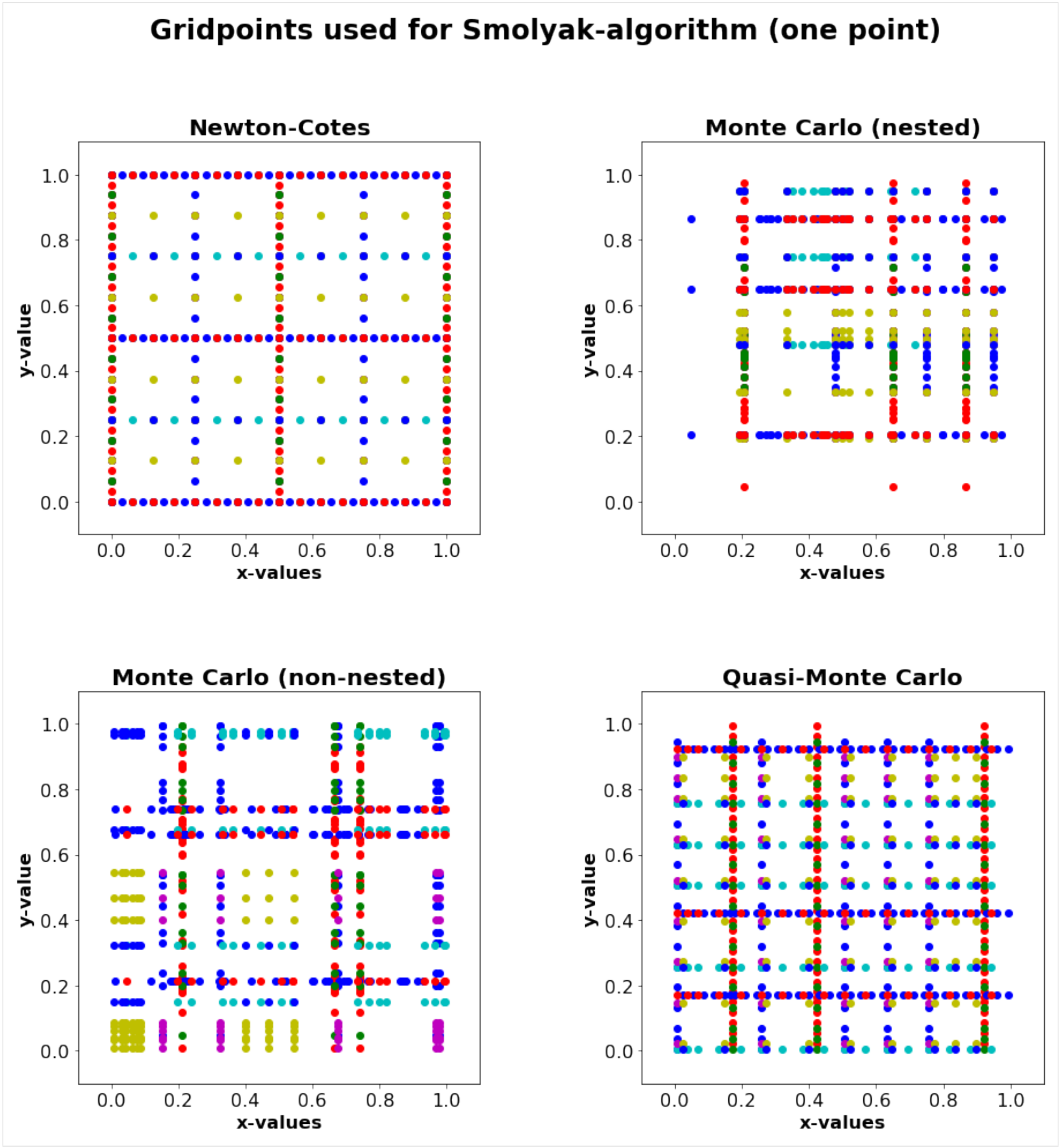
```
[3]: from Methodes_Studienproject.Studienprojekt_Smolyak_qmc_three_points import *
```

```
[4]: def grid_points(quadrature: str, q: int, a=0,b=1):
         list_of_points = []
         dim = 2
         # Now for the  probabilistic quadratures, only one point is used for the first␣
     ↪one-dimensional
         # degree of approximation. For this we add one to q, because otherwise the number␣
     ↪of points for
         # the probabilistic quadratures would be much  smaller.


         # Now we apply the quadrature chosen by user
         # !! If quadratures is changed or extended in interface, please options here !!

         # Because the points used for the deterministic quadratures are not changed, it␣
     ↪is faster to call them
         # at the beginning before starting the Smolyak alg.


         # For every one dimensional degree of approx.  a weights and points vector
         for i in range(1, (q - dim + 2)):

             # One option for Newton-Cotes and Trapezodial quadrature, because
             if quadrature == "Newton-Cotes" or quadrature == "Trapezoidal":
                 points, weights =one_dim_trapezoidal(i, a, b)

             if quadrature ==  "Monte Carlo (nested)" and i >1:
                 points, weights = monte_carlo_quad(i-1, a, b)
                 points = np.concatenate([list_of_points[-1], points])


             if quadrature ==  "Monte Carlo (non-nested)"or (quadrature == "Monte Carlo␣
     ↪(nested)" and i == 1)  :
                 points, weights = monte_carlo_quad(i, a, b)

             if quadrature ==  "Quasi-Monte Carlo":
                 points, weights = qmc_quad(i, a, b)

             list_of_points.append(points)

         # Now we get the combinations of one-dimensional degrees of approx. in Q(q,d)
         rng = list(range(q)) * dim
         rng = [x + 1 for x in rng]

         possible_combinations = list(set(i for i in itt.permutations(rng, dim) if (q -␣
     ↪dim) < sum(i) < q + 1))
```

(continues on next page)

(continued from previous page)

```
    number_approx = len(possible_combinations)
    meshgrid_list_x = []
    meshgrid_list_y = []
    for i in range(number_approx):

        # the next two steps could be combined in one line.
        # what is done is, that first the tuple at position i is taken from the list␣
↪of the set I.
        # Then the regarding vectors are taken from the list with the weight and␣
↪points vectors and
        # are combined to a d dimensional point set

        # gridpoints and the weight vectors are put into separate lists to make it␣
↪easier to
        # make a meshgrid
        current_tuple = possible_combinations[i]
        current_points = []

        for j in range(len(current_tuple)):
            current_points.append(list_of_points[current_tuple[j] - 1])

        # meshgrid yields a np.array with all coordinates needed for alg
        # coordinate and respective weight could be found in the same places in the␣
↪array
        meshgrid_points = np.array(np.meshgrid(*current_points))
        meshgrid_list_x.append(meshgrid_points[0])
        meshgrid_list_y.append(meshgrid_points[1])

    return [meshgrid_list_x, meshgrid_list_y]

def make_plot_of_grids():
    option_list = ["Newton-Cotes",
                   "Monte Carlo (nested)",
                   "Monte Carlo (non-nested)",
                   "Quasi-Monte Carlo"]
    fig = plt.figure(figsize=(15,15))
    plt.subplots_adjust(hspace=0.4, wspace=0.4)

    for i in range(len(option_list)):
        grid_list_x, grid_list_y = grid_points(option_list[i], 6)
        plt.subplot(2,2,i+1)
        modified_scatter_plot(grid_list_x,grid_list_y,title=option_list[i], input_
↪list=True)
        plt.xticks(fontsize=16)
        plt.yticks(fontsize=16)


    fig.suptitle("Gridpoints used for Smolyak-algorithm (one point)", fontsize=24,
↪fontweight="bold")
    plt.show()

make_plot_of_grids()
```



Here the grid points used for the approximation of the integral of a 2 dimensional function with the level of approximation q = 6 is shown, if one point is used for the one is used for probabilistic quadratures and q = 1. Apart form the upper left grid, the other grids change a little every time they are generated.

```
[ ]:
```

CHAPTER

TWO

PROOF OF PRINCIPLE

2.1 One point for q = 1

We want to show that the approximations are working. For this we use an easy differential function. The result of the integral is 1. We only take low degrees of evaluation, because we only want to give a proof of principle. Quantitative evaluations comparisons are made afterwards for on a statistical basis.

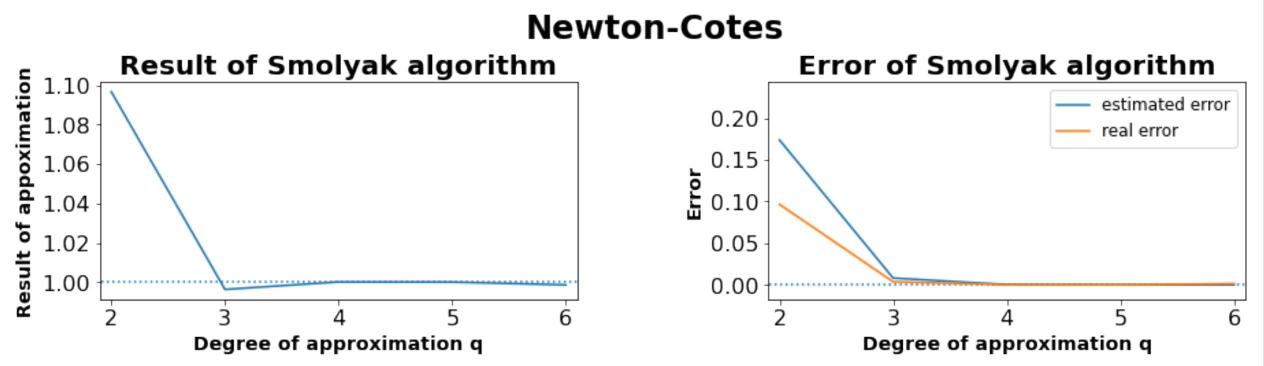
```
[1]: import matplotlib.pyplot as plt
     import time as time
     import pandas as bearcats
     import matplotlib.colors as colors

     import os
     os.chdir("..")
     from Methodes_Studienproject.Studienprojekt_Smolyak_qmc_one_point import *
```

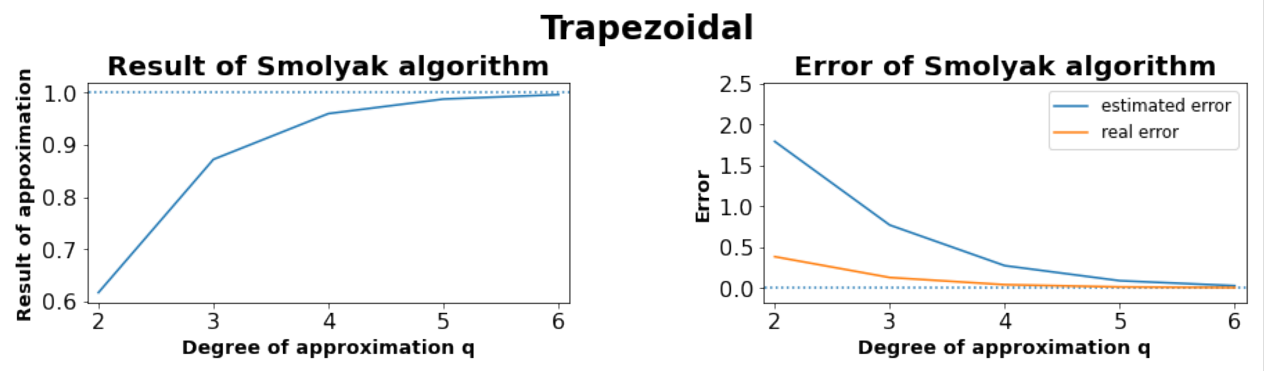
```
[2]: function_string = "pi**2/4 *sin(pi*x)*sin(pi*y)"
     variables_string = "(x,y)"
     option_list = ["Newton-Cotes",
                    "Trapezoidal",
                    "Monte Carlo (nested)",
                    "Monte Carlo (non-nested)",
                    "Quasi-Monte Carlo"]
     result_list = []

     for i in range(len(option_list)):
         results_quad = []
         for j in  range(2,7):
             results_quad.append(controller_smolyak(function_string, variables_string,␣
     ↪option_list[i], j))
         result_list.append(results_quad)

     for i in range(5):
         result = [result_list[i][j][0] for j in range(5) ]
         error  = [result_list[i][j][1] for j in range(5) ]
         cost   = [result_list[i][j][2] for j in range(5) ]
         fig = plt.figure(figsize=(15,7))
         plt.subplots_adjust(hspace=0.4, wspace=0.4)

         plt.subplot(2,2,1)
         plt.plot(list(range(2,7)),result)
         plt.hlines(1,0,10, linestyles= "dotted")
```

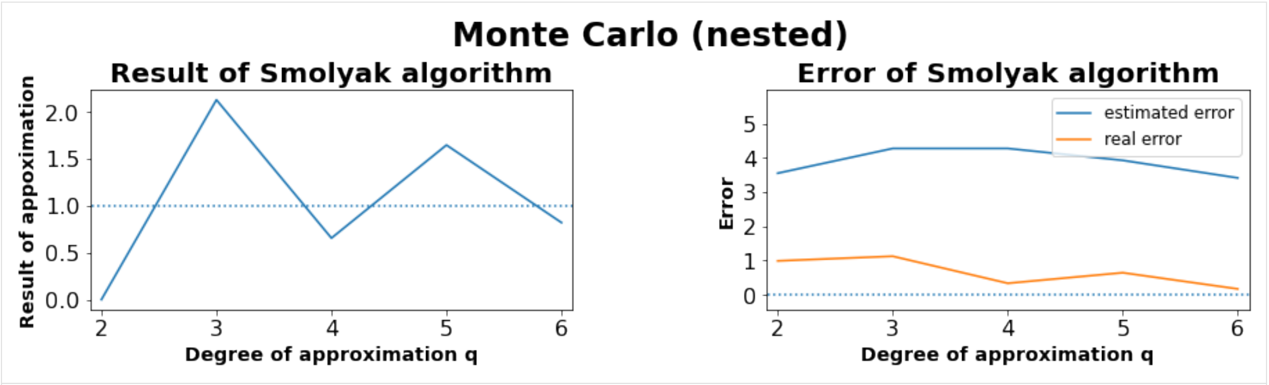
(continues on next page)

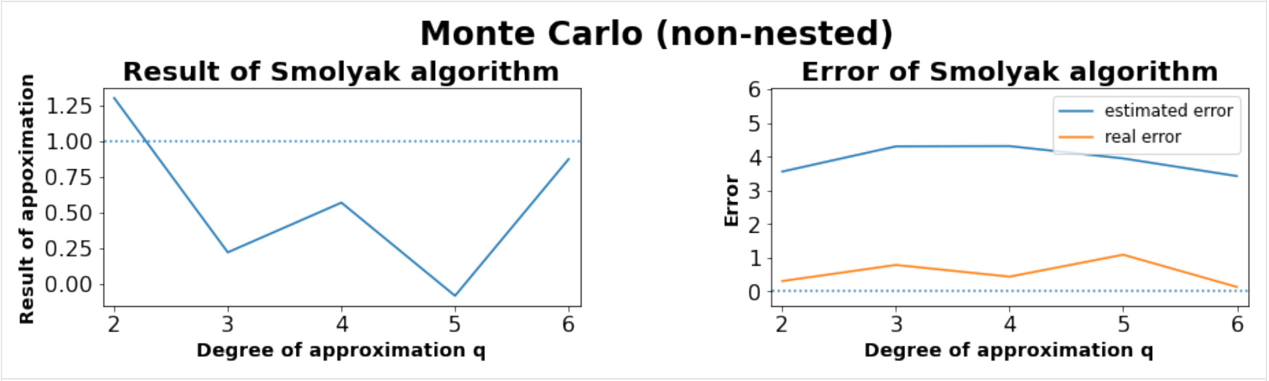
(continued from previous page)

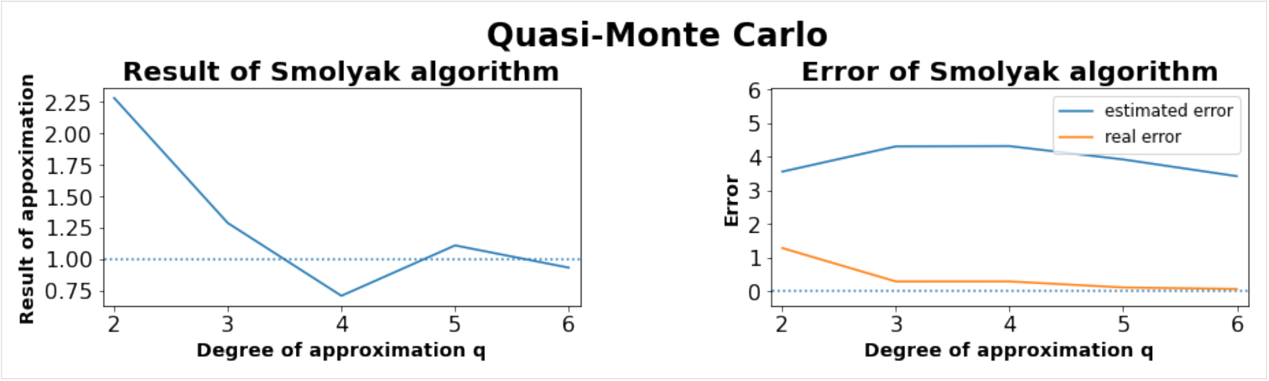
```
    plt.xlim(1.9,6.1)
    plt.xlabel("Degree of approximation q", fontsize = 14,fontweight = "bold")
    plt.ylabel("Result of appoximation",fontsize = 14,fontweight = "bold")
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)
    plt.xticks(range(2,7))
    plt.title("Result of Smolyak algorithm", fontsize = 20, fontweight = "bold")

    plt.subplot(2,2,2)
    plt.plot(list(range(2,7)),error, label= "estimated error")
    plt.plot(list(range(2,7)),[abs(x-1) for x in result], label= "real error")
    plt.hlines(0,0,10, linestyles= "dotted")
    plt.xlim(1.9,6.1)

    plt.xlabel("Degree of approximation q", fontsize=14,fontweight = "bold")
    plt.ylabel("Error", fontsize=14,fontweight = "bold")
    plt.title("Error of Smolyak algorithm",fontsize = 20,  fontweight="bold")
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)
    plt.xticks(range(2,7))
    plt.ylim(-max(error)*0.1,max(error)*1.4)
    plt.legend(fontsize=12, loc=1)
    plt.suptitle(option_list[i],fontsize=24, fontweight="bold")
    plt.show()
```











The result of the approximation and the error each should converge towards the dotted line. The estimated error for the probabilistic quadratures is not really significant, because the central limit theorem could not be applied here. The probabilistic quadratures for q_i = 1 only uses one point. For this reason these quadratures are less exact.

2.2 Three points for q = 1.

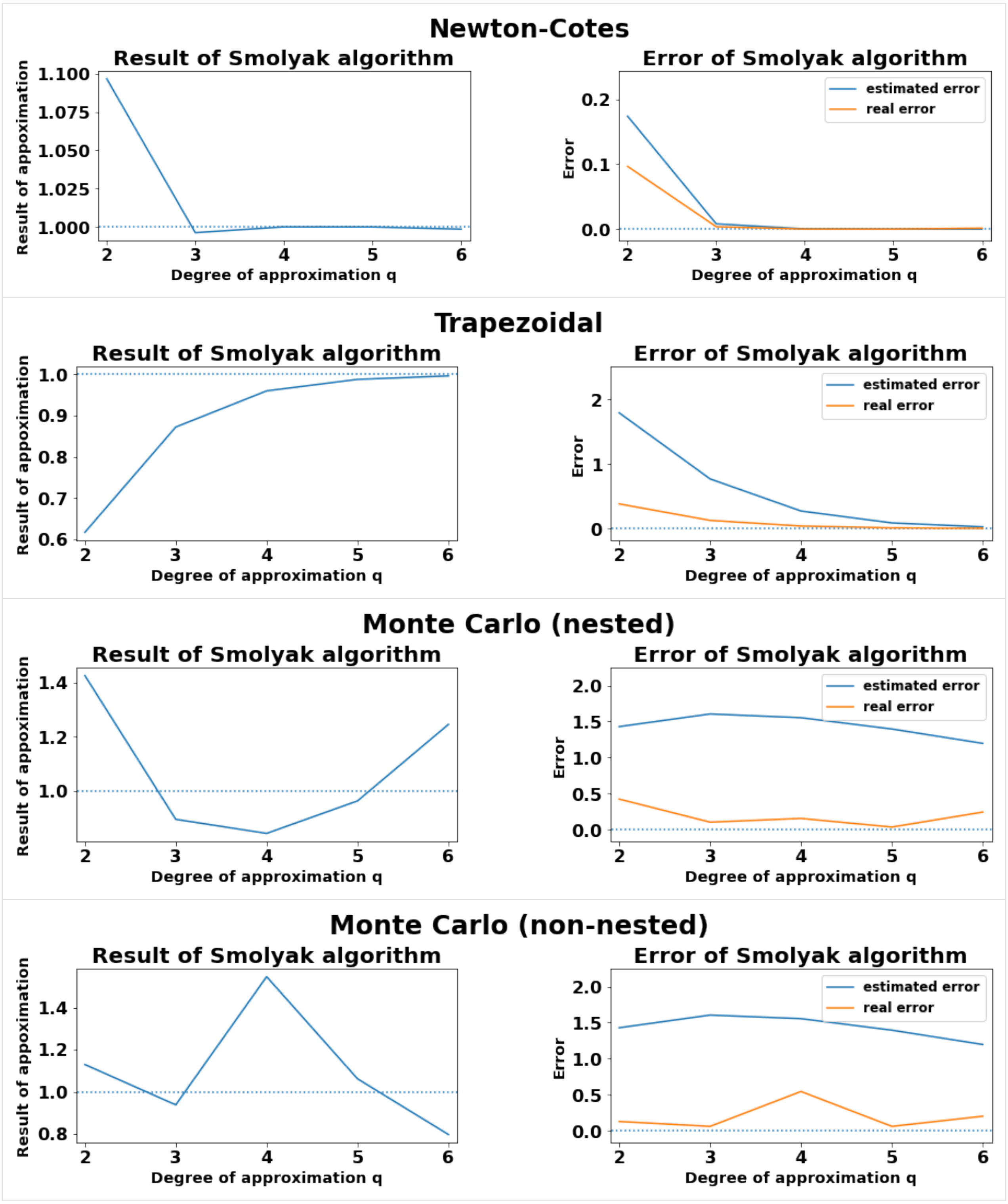
```
[11]: import matplotlib.pyplot as plt
      import time as time
      from Methodes_Studienproject.Studienprojekt_Smolyak_qmc_three_points import *
```

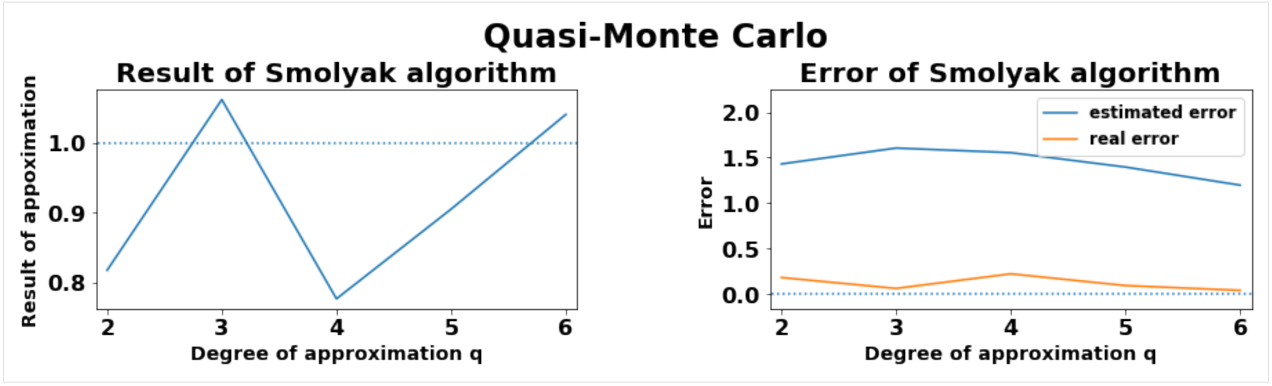
```
[12]: function_string = "pi**2/4 *sin(pi*x)*sin(pi*y)"
      variables_string = "(x,y)"
      option_list = ["Newton-Cotes",
                     "Trapezoidal",
```

(continues on next page)

(continued from previous page)

```
                "Monte Carlo (nested)",
                "Monte Carlo (non-nested)",
                "Quasi-Monte Carlo"]
result_list = []

for i in range(len(option_list)):
    results_quad = []
    for j in  range(2,7):
        results_quad.append(controller_smolyak(function_string, variables_string,␣
↪option_list[i], j))
    result_list.append(results_quad)

for i in range(5):
    result = [result_list[i][j][0] for j in range(5) ]
    error  = [result_list[i][j][1] for j in range(5) ]
    cost   = [result_list[i][j][2] for j in range(5) ]
    fig = plt.figure(figsize=(15,7))
    plt.subplots_adjust(hspace=0.4, wspace=0.4)

    plt.subplot(2,2,1)
    plt.plot(list(range(2,7)),result)
    plt.hlines(1,0,10, linestyles= "dotted")
    plt.xlim(1.9,6.1)
    plt.xlabel("Degree of approximation q", fontsize = 14,fontweight = "bold")
    plt.ylabel("Result of appoximation",fontsize = 14,fontweight = "bold")
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)
    plt.xticks(range(2,7))
    plt.title("Result of Smolyak algorithm", fontsize = 20, fontweight = "bold")

    plt.subplot(2,2,2)
    plt.plot(list(range(2,7)),error, label= "estimated error")
    plt.plot(list(range(2,7)),[abs(x-1) for x in result], label= "real error")
    plt.hlines(0,0,10, linestyles= "dotted")
    plt.xlim(1.9,6.1)

    plt.xlabel("Degree of approximation q", fontsize=14,fontweight = "bold")
    plt.ylabel("Error", fontsize=14,fontweight = "bold")
    plt.title("Error of Smolyak algorithm",fontsize = 20,  fontweight="bold")
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)
    plt.xticks(range(2,7))
    plt.ylim(-max(error)*0.1,max(error)*1.4)
    plt.legend(fontsize=12, loc=1)
    plt.suptitle(option_list[i],fontsize=24, fontweight="bold")
    plt.show()
```


Newton-Cotes
Result of Smolyak algorithm
Error of Smolyak algorithm
Trapezoidal
Result of Smolyak algorithm
Error of Smolyak algorithm
Monte Carlo (nested)
Result of Smolyak algorithm
Error of Smolyak algorithm
Monte Carlo (non-nested)
Result of Smolyak algorithm
Error of Smolyak algorithm
Result of appoximation
Error
Degree of approximation q
estimated error
real error



The result of the approximation and the error each should converge towards the dotted line. The estimated error for the probabilistic quadratures is not really significant, because the central limit theorem could not be applied here. The probabilistic quadratures for q_i = 1 here use three points.

2.3 Higher dimension

2.3.1 Approximation of constant

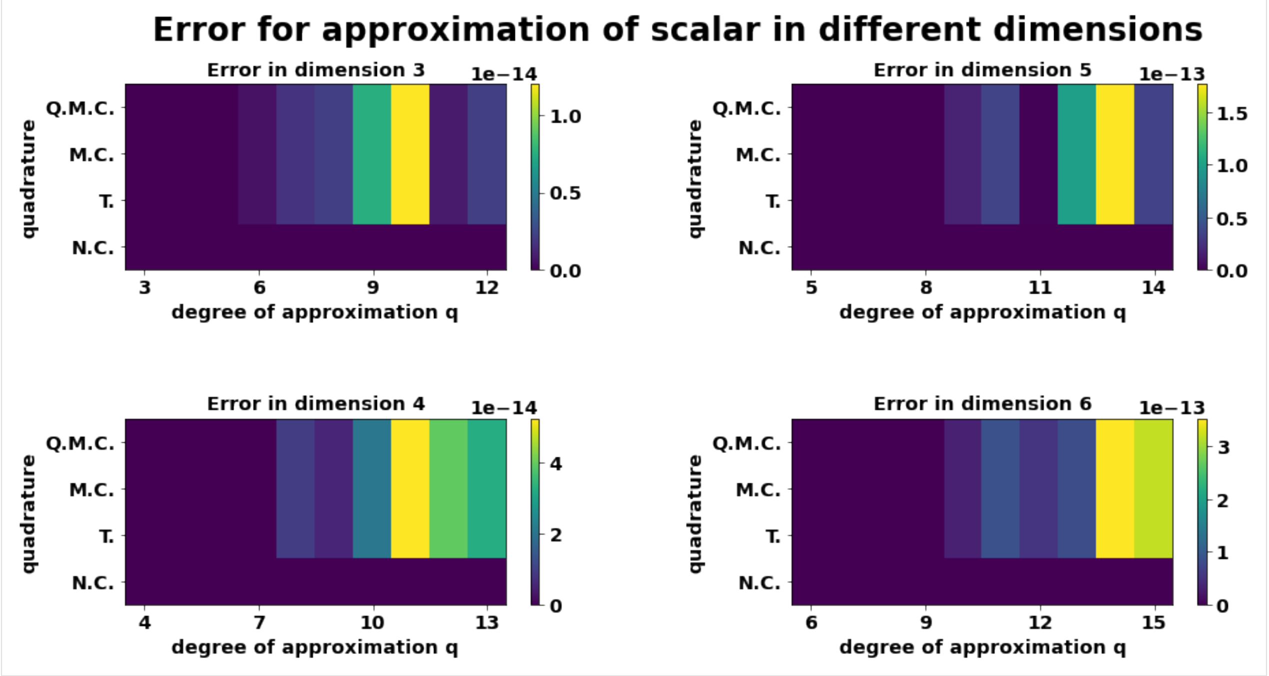
In the end, we want to show that the algorithm is working correct in different dimension. For this we first calculate the integral $f_d(\vec{x}) = 1$, to see, if the Smolyak algorithm is working well. Here the only interesting point is the difference of the result to 1. This should be sufficiently small to be explainable by the machine epsilon. After this we calculated the integral of $f_d(\vec{x}) = 2^d \cdot \prod_{i=0}^{d} x_i$ on the interval $[0,1]^d$. This integral is approximated exact by the trapezoidal quadrature. Apart from that an impression of the error for the non-deterministic quadratures.

The data is loaded, because the calculation takes about 10 minutes. The calculations are repeated 10 times for q = d + 10.

[13]:
```
color_str = [ "fuchsia", "navy", "limegreen", "red"]

plt.rc('font', size=14, weight="bold")
os.chdir("Data")
approx_simple_highdim_fct = bearcats.read_pickle("integral_over_one.pkl")
np.mean(approx_simple_highdim_fct["2, results"],axis=2).shape
fig, ax = plt.subplots(2,2,figsize=(15,7))


for dim in range(3,7):
    error = abs(1 - np.mean(approx_simple_highdim_fct[str(dim) + ", results"],axis=2))


    # Time plot

    plt.subplots_adjust(hspace=0.8, wspace=0.4)

    mat = ax[(dim-3)%2, int((dim-3)/2)].pcolor(error)
    fig.colorbar(mat, ax=ax[(dim-3)%2, int((dim-3)/2)])

    ax[(dim-3)%2, int((dim-3)/2)].set_ylabel("quadrature",fontsize=14, fontweight=
↪"bold")
    ax[(dim-3)%2, int((dim-3)/2)].set_yticks([0.5,1.5,2.5,3.5])
```
(continues on next page)

(continued from previous page)

```
    ax[(dim-3)%2, int((dim-3)/2)].set_yticklabels(["N.C.","T.", "M.C.", "Q.M.C."],
↪fontsize=14, fontweight="bold")
    ax[(dim-3)%2, int((dim-3)/2)].set_xlabel("degree of approximation q",fontsize=14,␣
↪fontweight="bold")
    ax[(dim-3)%2, int((dim-3)/2)].set_xticks([0.5,3.5,6.5,9.5])
    ax[(dim-3)%2, int((dim-3)/2)].set_xticklabels([dim,dim+3,dim+6,dim+9],fontsize=14,
↪ fontweight="bold")
    ax[(dim-3)%2, int((dim-3)/2)].set_title("Error in dimension "+ str(dim),
↪fontsize=14, fontweight="bold")

fig.suptitle("Error for approximation of scalar in different dimensions\n",␣
↪fontsize=24,fontweight="bold")

plt.show()
```



It turns out that the error is in the suspected range of the machine epsilon. Furthermore, it is not systematically increasing with q. This means that the different summands used in the Smolyak algorithm are summed up in a proper way. The calculation was repeated 25 times and every time the error stayed the same.

The bottom line is 0, because the Newton-Cotes algorithm could not be applied for any one-dimensional q bigger 5. Up to that point no error occurred.

2.3.2 Approximation of simple analytical function

Next we show the results of the approximation.

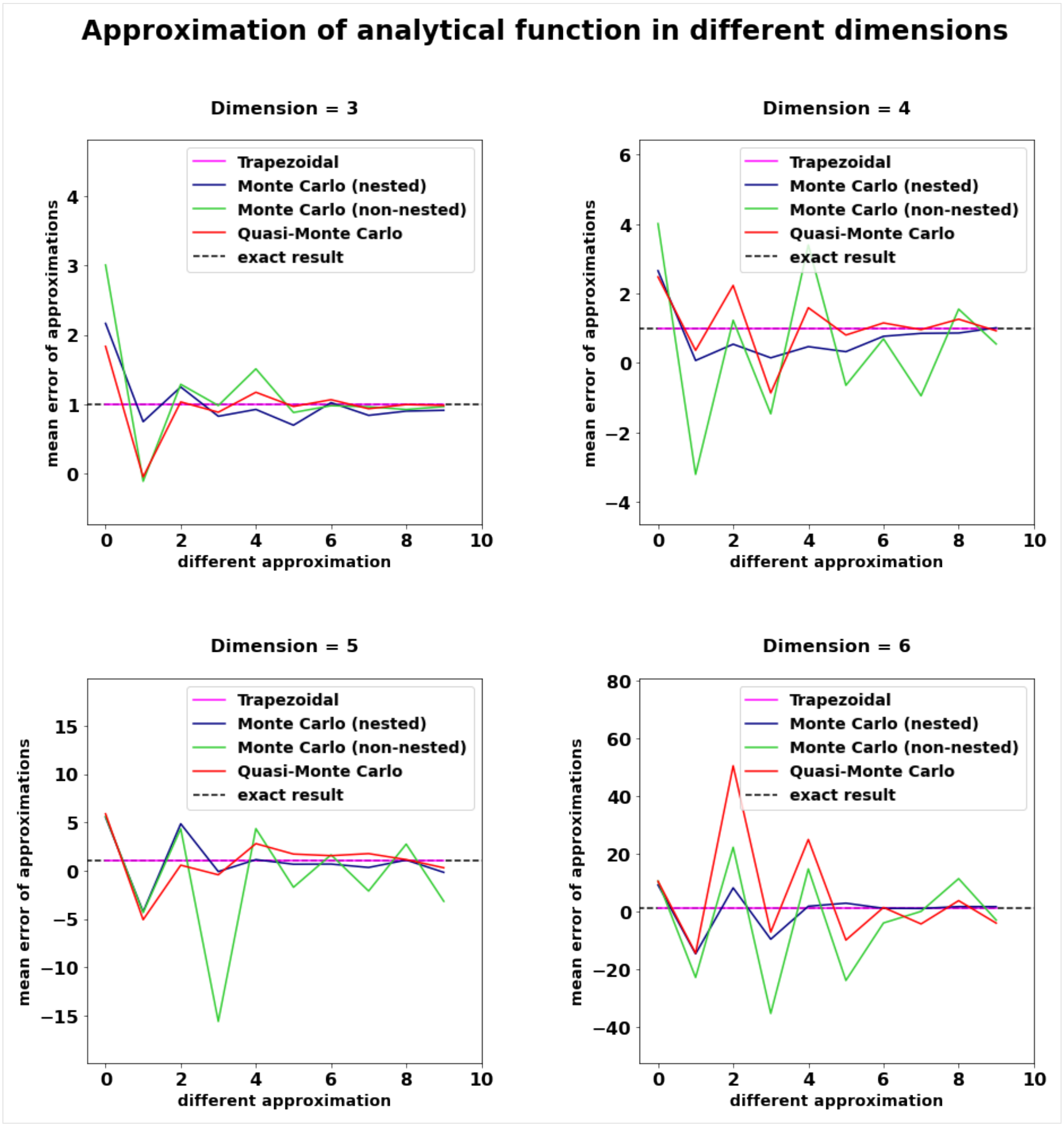
First we show the average of 25 calculation. As mentioned the deterministic quadratures are exact for this functions.

[14]:
```
color_str = [ "fuchsia", "navy", "limegreen", "red"]


approx_simple_highdim_fct = bearcats.read_pickle("integral_over_polynomial.pkl")
fig = plt.figure(figsize=(15,15))
```

(continues on next page)

(continued from previous page)

```
for dim in range(3,7):
    approx = np.mean(approx_simple_highdim_fct[str(dim) + ", results"], axis=2)


    # Time plot

    plt.subplots_adjust(hspace=0.4, wspace=0.4)
    plt.subplot(2,2,dim-2)

    for k_1 in range( approx.shape[0]):
        plt.plot((approx[k_1,:]), color = color_str[k_1], label=option_list[k_1+1])

    plt.xlim(-0.5,10,5)
    plt.hlines(1, xmin=-1, xmax = 11, linestyles="--", color= "black", label="exact␣
↪result")
    plt.legend(loc=1)
    plt.ylabel("mean error of approximations",fontsize = 14,fontweight = "bold")
    plt.xlabel("different approximation",fontsize = 14,fontweight = "bold")
    plt.ylim(np.max(approx)- abs(np.max(approx)-np.min(approx))*1.2, np.max([np.
↪max(approx)*1.6,abs(np.max(approx)- abs(np.max(approx)-np.min(approx))*1.2)]))
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)
    plt.title("Dimension = " + str(dim)+"\n", fontsize = 16,fontweight = "bold")
fig.suptitle("Approximation of analytical function in different dimensions",␣
↪fontsize=24,fontweight="bold")

plt.show()
```



The deterministic quadratures are exact. The non deterministic quadratures tend to be relatively inaccurate especially in higher dimensions

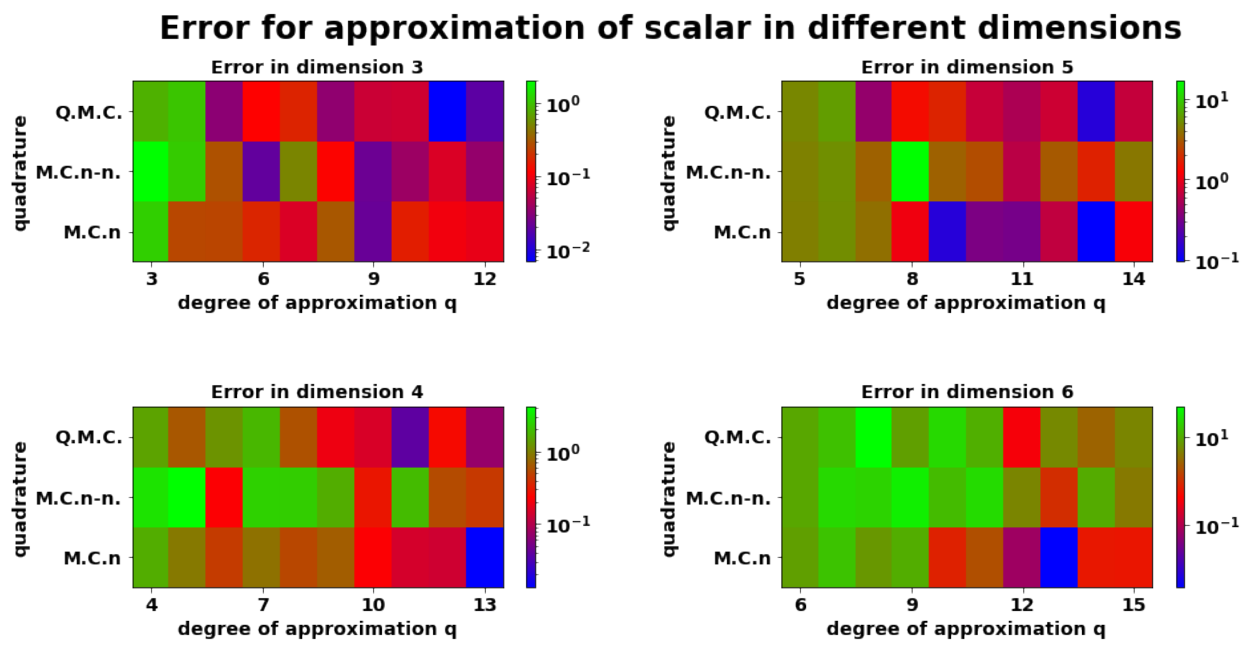
Now the errors of the non-deterministic approximations are shown in a logarithmic scale.

```
[15]: plt.rc('font', size=14, weight="bold")
      approx_simple_highdim_fct = bearcats.read_pickle("integral_over_polynomial.pkl")


      fig, ax = plt.subplots(2,2,figsize=(15,7))
```

(continues on next page)

(continued from previous page)

```
for dim in range(3,7):
    error = abs(1 - np.mean(approx_simple_highdim_fct[str(dim) + ", results"][1:,:],
↪axis=2))


    # Time plot

    plt.subplots_adjust(hspace=0.8, wspace=0.4)

    mat = ax[(dim-3)%2, int((dim-3)/2)].pcolor(error,norm=colors.LogNorm(vmin=error.
↪min(), vmax=error.max()),
                    cmap='brg')
    fig.colorbar(mat, ax=ax[(dim-3)%2, int((dim-3)/2)])

    ax[(dim-3)%2, int((dim-3)/2)].set_ylabel("quadrature",fontsize=14, fontweight=
↪"bold")
    ax[(dim-3)%2, int((dim-3)/2)].set_yticks([0.5,1.5,2.5])
    ax[(dim-3)%2, int((dim-3)/2)].set_yticklabels(["M.C.n", "M.C.n-n.", "Q.M.C."],
↪fontsize=14, fontweight="bold")
    ax[(dim-3)%2, int((dim-3)/2)].set_xlabel("degree of approximation q",fontsize=14,␣
↪fontweight="bold")
    ax[(dim-3)%2, int((dim-3)/2)].set_xticks([0.5,3.5,6.5,9.5])
    ax[(dim-3)%2, int((dim-3)/2)].set_xticklabels([dim,dim+3,dim+6,dim+9],fontsize=14,
↪ fontweight="bold")
    ax[(dim-3)%2, int((dim-3)/2)].set_title("Error in dimension "+ str(dim),
↪fontsize=14, fontweight="bold")


fig.suptitle("Error for approximation of scalar in different dimensions\n",␣
↪fontsize=24,fontweight="bold")
plt.show()
```



```
[1]: '''
import Methodes_Studienproject.Studienprojekt_Smolyak_qmc_one_point as Studieproject_
↪one
import numpy as np
import time
import pandas as bearcats

# for proof of principle
variables = ["(x_1)"]
#function = ["1"]
functions = ["2 * x_1"]
option_list = ["Trapezoidal",
               "Monte Carlo (nested)",
               "Monte Carlo (non-nested)",

               "Quasi-Monte Carlo"]
repetions = 25
results = {}
for dim in range(2,7):
    print(dim)
    variables.append(variables[-1][:-1]+", x_"+ str(dim)+")")
    functions.append(functions[-1]+"* 2 * x_"+str(dim))
    # functions.append(functions[-1])
    results_one_dim= np.empty((4,10,repetions,3))
    run_times = np.empty((4))
    for k_1 in range(len(option_list)):

        start = time.time()
        for k_3  in range(dim,dim+10):
            print(k_3)
            for k_2 in range(repetions):
                results_one_dim[k_1, k_3-dim, k_2, :] = Studieproject_one.controller_
↪smolyak(functions[-1], variables[-1], option_list[k_1], k_3)
            run_times[k_1] = (time.time()-start)/25
    new_results = { str(dim)+ ", results": results_one_dim[:,:,:,0], str(dim)+ ", time
↪": run_times}
    results.update(new_results)
bearcats.to_pickle(results,"integral_over_polynomial.pkl")
# bearcats.to_pickle(results,"integral_over_one.pkl")
'''
print("If you want to generate data, remove quotation marks.")
```

```
If you want to generate data, remove quotation marks.
```

```
[ ]:
```

CHAPTER THREE

APPROXIMATE π

The way not not steady functions could be used in the project is illustrated in a well known basic example for the use of the Monte Carlo method. This is the approximation of pi. Here the it is used that the area of the (unit) circle is proportional to pi. Hence it is possible to approximate π by setting all points in the circle to 1 and all others to 0. Then the average of these value converges to π. Because we restrict our self to the interval $[0, 1]^n$, we have

$$\pi = 4 \cdot A(2, q)$$

and

$$\pi = 6 \cdot A(3, q),$$

with $A(n, q)$ being the result of the Smolyak algorithm in n dimensions and the degree of approximation q. Because the function is not derivable, it is not possible to estimate the error for the algorithm for the deterministic quadratures. We will just compare them to the value of π used by numpy.

```
[6]: import os
os.chdir("..")


import Methodes_Studienproject.Studienprojekt_Smolyak_qmc_one_point as Studieproject_
↪one
import matplotlib.pyplot as plt
import sympy as sp
import numpy as np
import itertools as itt
```

3.1 The functions used

```
[2]: # Functions receiving 2 or 3 matrices of the the dimension 2 or 3. Each matirce␣
↪contains the coordinate in
# the component x, y or z. The function returns a martice of the shape of the input␣
↪components with 1 if
# the 2-norm of the coordinate is smaller or equal 1 and 0 otherwise.
def circle(X_1, X_2):
    save_mat = np.ones(X_1.shape)

    for index_vec, vectors in enumerate(X_1):
        for index_col, x_1 in enumerate(vectors):
```

(continues on next page)

(continued from previous page)

```
            if x_1**2 + X_2[index_vec, index_col]**2 > 1:
                save_mat[index_vec, index_col] = 0
    return save_mat

def ball(X_1, X_2, X_3):
    save_mat = np.ones(X_1.shape)
    for index_mat, matrix in enumerate(X_1):
        for index_vec, vectors in enumerate(matrix):
            for index_col, x_1 in enumerate(vectors):

                if x_1**2 + X_2[index_mat, index_vec, index_col]**2 + X_3[index_mat,␣
↪index_vec, index_col]**2 > 1:
                    save_mat[index_mat, index_vec, index_col] = 0
    return save_mat
color_str = [ "fuchsia", "navy", "limegreen", "red"]
variables_string_rewrite = "(x_1, x_2, x_3)"
variables_string_rewrite = variables_string_rewrite.replace(" ", "")
variables = variables_string_rewrite.strip(')(').split(',')
variables_string_rewrite = variables_string_rewrite.strip(')(').split(',')

if len(variables) == 0:
    raise Exception("There at least has to be one variable.")

for f in range(len(variables)):
    variables[f] = sp.Symbol("x_" + str(f))

option_list = ["Trapezoidal",
               "Monte Carlo (nested)",
               "Monte Carlo (non-nested)",
               "Quasi-Monte Carlo"]

# Approximation of pi with different q and degrees of approx.
lower_bound, upper_bound = [8,15]
approx_pi_circle = np.empty([len(option_list), upper_bound - lower_bound])
for index_option, option in enumerate(option_list):
    for index_q, degree_of_approx in enumerate(range(lower_bound, upper_bound)):
        approx_pi_circle[index_option, index_q] = 4 * Studieproject_one.controller_
↪smolyak(circle, variables[:2], option, degree_of_approx, function_given=True, no_
↪error=True)[0]

approx_pi_ball = np.empty([len(option_list), upper_bound - lower_bound])
for index_option, option in enumerate(option_list):
    for index_q, degree_of_approx in enumerate(range(lower_bound, upper_bound)):
        approx_pi_ball[index_option, index_q] = 6 * Studieproject_one.controller_
↪smolyak(ball, variables, option, degree_of_approx, function_given=True, no_
↪error=True)[0]
```

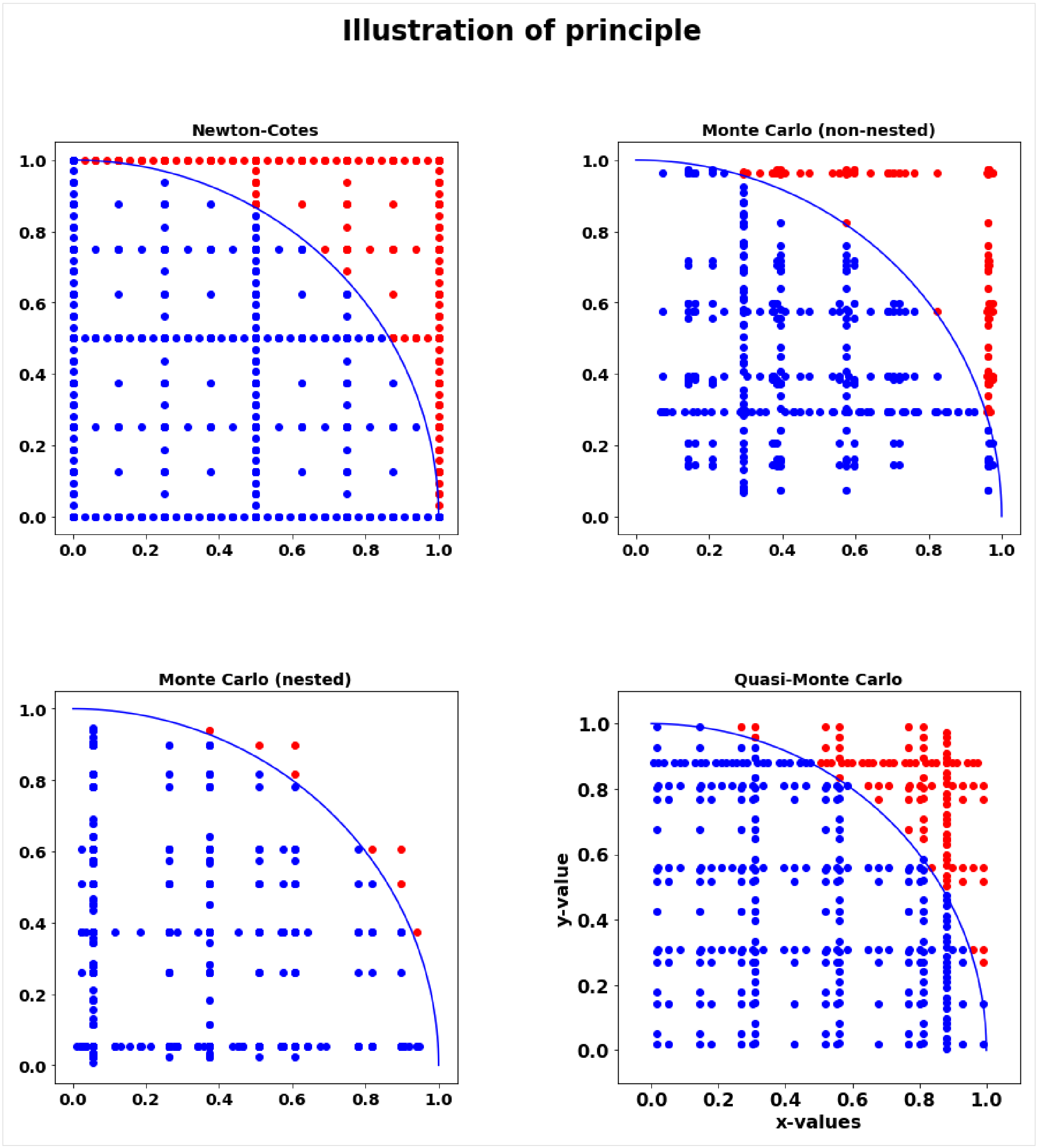
3.2 Illustration of the principle of approximation for 2 dimensions

```
[3]: plt.rc('font', size=14, weight="bold")
def grid_points(quadrature: str, q: int, a=0,b=1):
    list_of_points = []
    dim = 2

    # Because the points used for the deterministic quadratures are not changed, it␣
↪is faster to call them
    # at the beginning before starting the Smolyak alg.
    if "Carlo" in quadrature:
        q = q + 1

    # For every one dimensional degree of approx.  a weights and points vector
    for i in range(1, (q - dim + 2)):

        # One option for Newton-Cotes and Trapezoidal quadrature, because
        if quadrature == "Newton-Cotes" or quadrature == "Trapezoidal":
            points, weights = Studieproject_one.one_dim_trapezoidal(i, a, b)

        if quadrature ==  "Monte Carlo (nested)" and i >1:
            points, weights = Studieproject_one.monte_carlo_quad(i-1, a, b)
            points = np.concatenate([list_of_points[-1], points])


        if quadrature ==  "Monte Carlo (non-nested)"or (quadrature == "Monte Carlo␣
↪(nested)" and i == 1)   :
            points, weights = Studieproject_one.monte_carlo_quad(i, a, b)

        if quadrature ==  "Quasi-Monte Carlo":
            points, weights = Studieproject_one.qmc_quad(i, a, b)

        list_of_points.append(points)

    # Now we get the combinations of one-dimensional degrees of approx. in Q(q,d)
    rng = list(range(q)) * dim
    rng = [x + 1 for x in rng]

    possible_combinations = list(set(i for i in itt.permutations(rng, dim) if (q -␣
↪dim) < sum(i) < q + 1))
    number_approx = len(possible_combinations)
    meshgrid_list_x = []
    meshgrid_list_y = []
    for i in range(number_approx):

        # gridpoints and the weight vectors are put into separate lists to make it␣
↪easier to
        # make a meshgrid
        current_tuple = possible_combinations[i]
        current_points = []

        for j in range(len(current_tuple)):
            current_points.append(list_of_points[current_tuple[j] - 1])

        # meshgrid yields a np.array with all coordinates needed for alg
        # coordinate and respective weight could be found in the same places in the␣
↪array
```

(continues on next page)

(continued from previous page)

```
        meshgrid_points = np.array(np.meshgrid(*current_points))
        meshgrid_list_x.append(meshgrid_points[0])
        meshgrid_list_y.append(meshgrid_points[1])

    return [meshgrid_list_x, meshgrid_list_y]

def make_plot_of_grids():
    a = 0
    b = 1
    option_list = ["Newton-Cotes",
                   "Monte Carlo (nested)",
                   "Monte Carlo (non-nested)",
                   "Quasi-Monte Carlo"]
    fig, ax = plt.subplots(2,2,figsize=(15,15))
    plt.subplots_adjust(hspace=0.4, wspace=0.4)

    for i in range(len(option_list)):
        x, y = grid_points(option_list[i], 6)
        circle_pi = list()
        for index_x, _ in enumerate(x):

            circle_pi.append(circle(x[index_x], y[index_x]))
            ax[i%2,int(i/2)].scatter(x[index_x][circle_pi[-1]==1], y[index_x][circle_
↪pi[-1]==1], color="blue",label="1")
            ax[i%2,int(i/2)].scatter(x[index_x][circle_pi[-1]==0], y[index_x][circle_
↪pi[-1]==0], color="red",label="0")
            ax[i%2,int(i/2)].set_title(option_list[i],fontsize=14, fontweight="bold")

        # print circle
        x_circle = np.linspace(0,np.pi / 2,100)
        r_x = np.cos(x_circle)
        r_y = np.sin(x_circle)
        ax[i%2,int(i/2)].plot(r_x,r_y,color="blue")

    # Some cosmetic changes in plot
    plt.xlabel("x-values", fontsize=16, fontweight="bold")
    plt.ylabel("y-value", fontsize=16, fontweight="bold")
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)
    plt.xlim(a - (b - a) / 10, b + (b - a) / 10)
    plt.ylim(a - (b - a) / 10, b + (b - a) / 10)
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)



    fig.suptitle("Illustration of principle", fontsize=24,fontweight="bold")
    plt.show()

make_plot_of_grids()
```



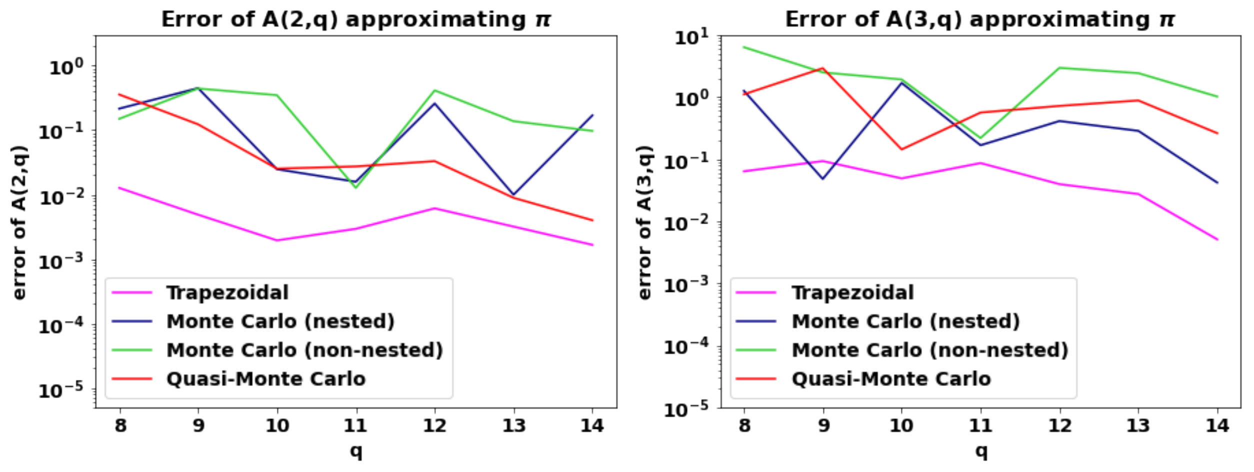
The blue points coincides with the points with value 1 and the red ones with the value 0.

```
[4]: fig, ax = plt.subplots(1,2,figsize=(15,5))

     for index_option, option in enumerate(option_list):
         ax[0].plot(range(lower_bound, upper_bound), abs(np.pi-approx_pi_circle[index_
     ↪option, :]), color = color_str[index_option],
                  label=option)
     ax[0].set_yscale("log")
```

(continues on next page)

(continued from previous page)

```
ax[0].set_ylim(0.000005,3)
ax[0].set_ylabel("error of A(2,q)",fontsize=14, fontweight="bold")
ax[0].set_xlabel("q",fontsize=14, fontweight="bold")
ax[0].set_xticks(list(range(lower_bound, upper_bound)))
ax[0].set_xticklabels(list(range(lower_bound, upper_bound)),fontsize=14, fontweight=
↪"bold")
ax[0].set_title(r"Error of A(2,q) approximating $\pi$",fontsize=16, fontweight="bold")
ax[0].legend()


for index_option, option in enumerate(option_list):
    ax[1].plot(range(lower_bound, upper_bound), abs(np.pi-approx_pi_ball[index_option,
↪ :]), color = color_str[index_option],
             label=option)
ax[1].set_yscale("log")
ax[1].set_ylabel("error of A(3,q)",fontsize=14, fontweight="bold")
ax[1].set_xlabel("q",fontsize=14, fontweight="bold")
ax[1].set_xticks(list(range(lower_bound, upper_bound)))
ax[1].set_xticklabels(list(range(lower_bound, upper_bound)),fontsize=14, fontweight=
↪"bold")
ax[1].set_title(r"Error of A(3,q) approximating $\pi$",fontsize=16, fontweight="bold")
ax[1].legend()
ax[1].set_ylim(0.00001,10)
plt.show()
```



```
[ ]:
```

CHAPTER

FOUR

STATISTICAL EVALUATION FOR ERROR FOR PROBABILISTIC QUADRATURES

In this script we want to evaluate statistical data. For this we load data, which was calculated and saved in a .cvs/ .pkt file, because generating file takes some hours.

```
[3]: import matplotlib.pyplot as plt

     import os
     os.chdir("..")
     from Methodes_Studienproject.Studienprojekt_Smolyak_qmc_one_point import *
```

4.1 First approximation of distribution

4.1.1 One point for q = 1

First we want to give a feeling for the distribution of the results of the Smolyak algorithm using different quadratures and the effect of higher degrees of approximation.

We only show the first degrees of approximation, because otherwise nothing could be seen, due to the scaling of the plot.

```
[11]: try:
          os.chdir("Data")
      except:
          pass
      results_one = load_stat_data("stat_approach_2_to_20_result_13_11_one.csv")
      results_three = load_stat_data("stat_approach_2_to_20_result_13_11_three.csv")
```

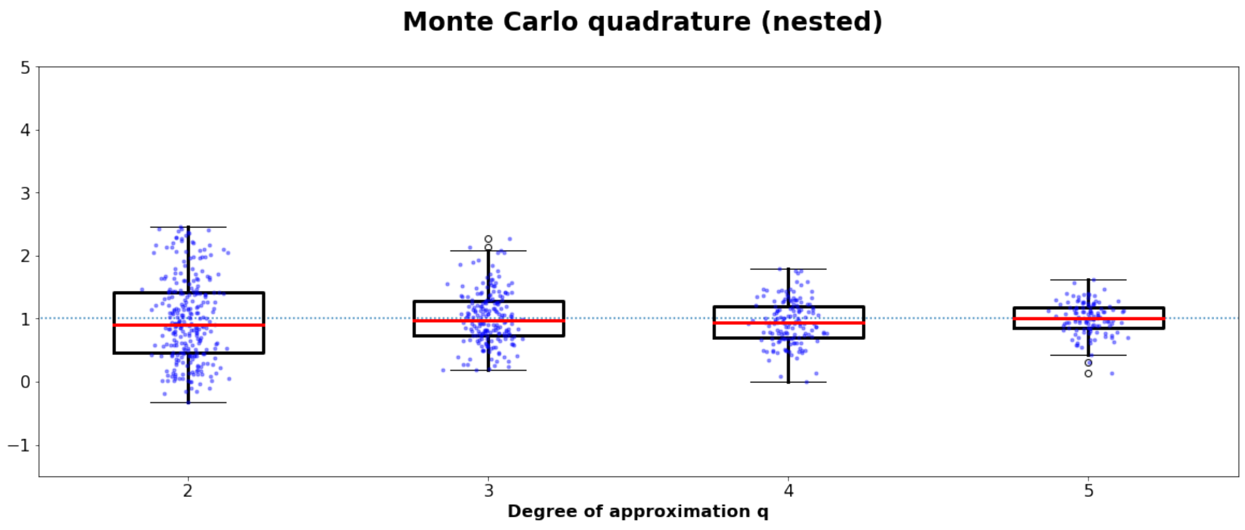
```
[12]: title_string = ["One point for q = 1 on the one dimensional level \n\n Monte Carlo␣
      ↪quadrature (nested)\n", "Monte Carlo quadrature (non-nested)\n",
                      "Quasi-Monte Carlo quadrature \n"]

      results = results_one
      for k_1 in range(len(results[0])):
          fig = plt.figure(figsize=(20,7))
          ax = plt.axes()
          boxprops = dict(linewidth=3.0, color='black')
          whiskerprops = dict(linestyle='-',linewidth=3.0, color='black')
          medianprops = dict(linestyle='-',linewidth=3.0, color='red')
          for k_2 in range(len(results[0][0])-14):
              y =results[0][k_1][k_2]
```

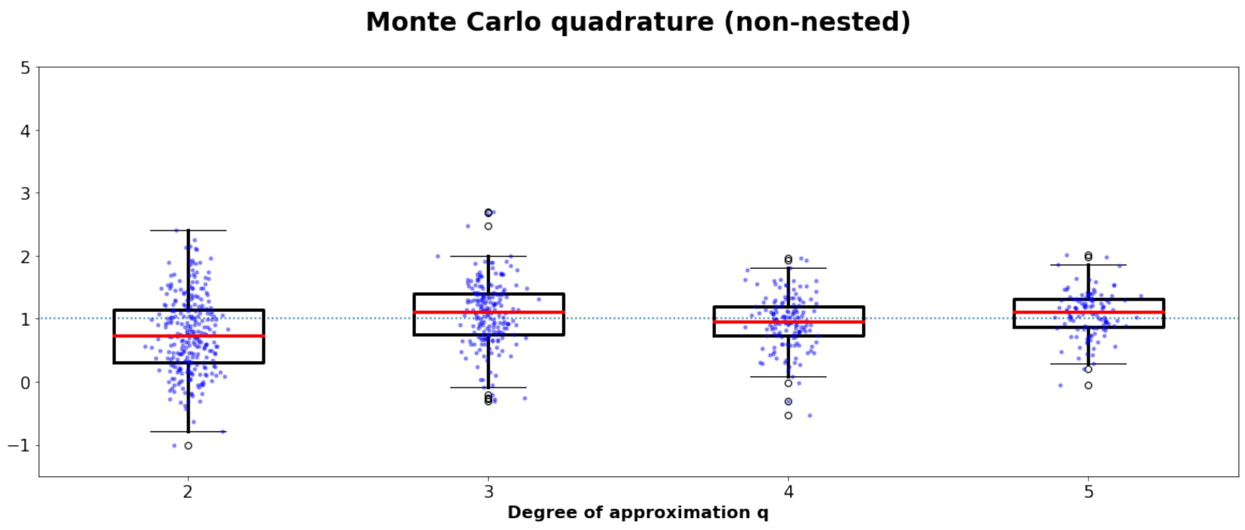
(continues on next page)

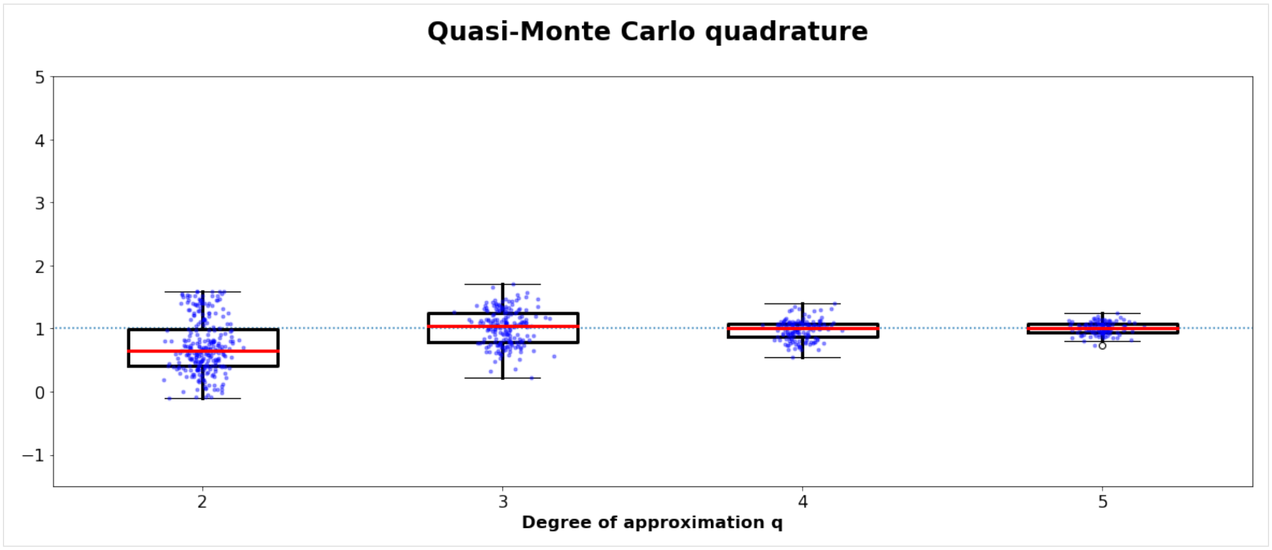
(continued from previous page)

```
        bp = plt.boxplot(y, positions = [k_2 + 1], widths = 0.5, boxprops=boxprops,
                        whiskerprops = whiskerprops, medianprops = medianprops)
        x = np.random.normal(k_2+1, 0.05, size=len(y))
        plt.plot(x, y, 'b.', alpha=0.4)
    plt.hlines(1,0,10, linestyles= "dotted")
    plt.xlim((1.5,  5.5))
    plt.ylim((-1.5,5))
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)
    plt.title(title_string[k_1], fontsize = 24, fontweight ="bold")
    plt.xlabel("Degree of approximation q", fontsize= 16, fontweight = "bold")
    plt.show()
```

One point for q = 1 on the one dimensional level







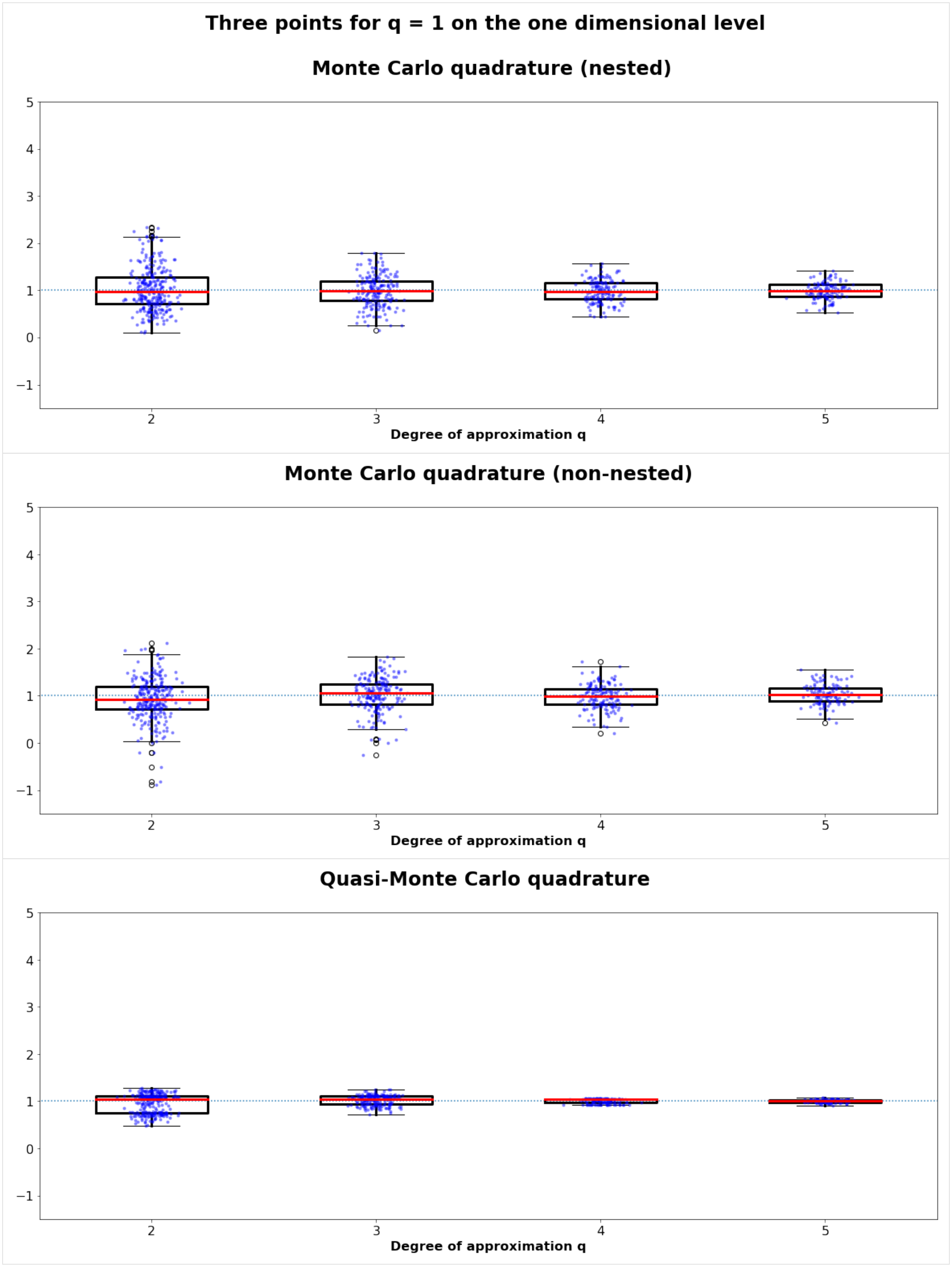
Boxplot of at least 200 calculations of the function $\pi^2/4 \cdot sin(\pi \cdot x) \cdot sin(\pi \cdot y)$ using probabilistic quadratures . Here q = 2 in implies that we the integral is approximated by a point evaluation. Thus, the high variance for low q is expected.

By taking a look at the the variance of the result and comparing it to the error estimation, we get a more quantitative impression about the goodness of the approximation

4.1.2 Three points

For comparison we also show the result, if for q = 1 on the one dimensional level we use 3 points.

```
[13]: title_string = ["Three points for q = 1 on the one dimensional level \n\n Monte Carlo␣
      ↪quadrature (nested)\n", "Monte Carlo quadrature (non-nested)\n",
                      "Quasi-Monte Carlo quadrature \n"]

      results = results_three
      for k_1 in range(len(results[0])):
          fig = plt.figure(figsize=(20,7))
          ax = plt.axes()
          boxprops = dict(linewidth=3.0, color='black')
          whiskerprops = dict(linestyle='-',linewidth=3.0, color='black')
          medianprops = dict(linestyle='-',linewidth=3.0, color='red')
          for k_2 in range(len(results[0][0])-14):
              y =results[0][k_1][k_2]
              bp = plt.boxplot(y, positions = [k_2 + 1], widths = 0.5, boxprops=boxprops,
                              whiskerprops = whiskerprops, medianprops = medianprops)
              x = np.random.normal(k_2+1, 0.05, size=len(y))
              plt.plot(x, y, 'b.', alpha=0.4)
          plt.hlines(1,0,10, linestyles= "dotted")
          plt.xlim((1.5,  5.5))
          plt.ylim((-1.5,5))
          plt.xticks(fontsize=16)
          plt.yticks(fontsize=16)
          plt.title(title_string[k_1], fontsize = 24, fontweight ="bold")
          plt.xlabel("Degree of approximation q", fontsize= 16, fontweight = "bold")
          plt.show()
```


Three points for q = 1 on the one dimensional level
Monte Carlo quadrature (nested)
Monte Carlo quadrature (non-nested)
Quasi-Monte Carlo quadrature
Degree of approximation q
5
4
3
2
1
0
−1

The approximation in this case is much more exact. A comparison of the runtime and the smaller error will be done later on.

4.2 Statistical parameters of distribution of results

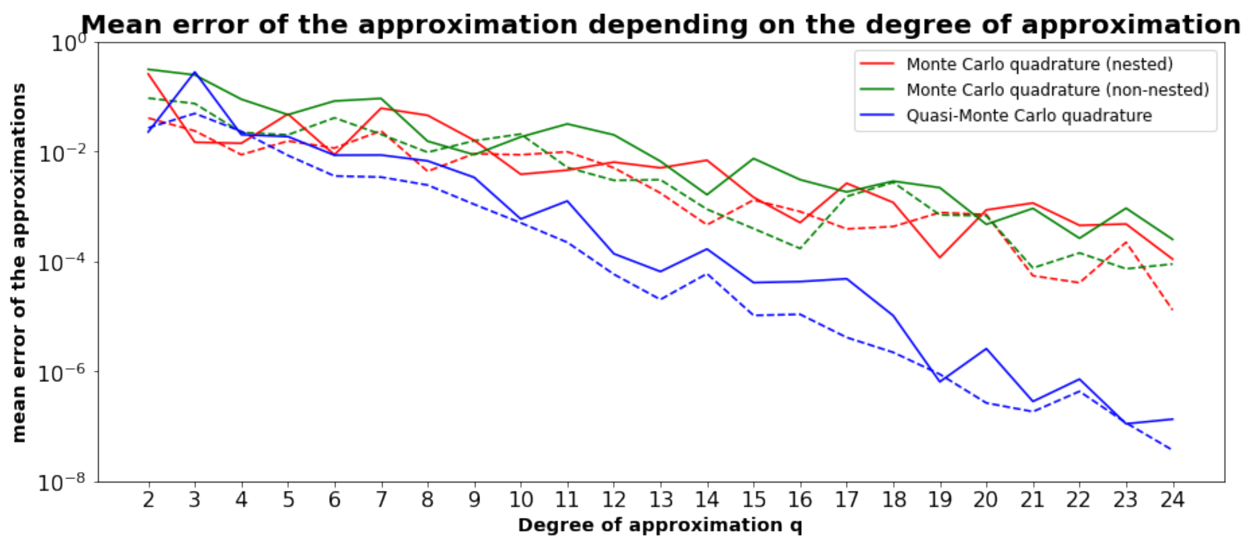
[14]:
```
# Calculation of mean an variance of the result and estimated error of approximation
mean_result = np.ones((2,3,len(results[0][0])))
sigma_result = np.ones((2,3,len(results[0][0])))

mean_error = np.ones((2,3,len(results[0][0])))
sigma_error = np.ones((2,3,len(results[0][0])))

runtime = np.ones((2,3,len(results[0][0])))
runtime_sig = np.ones((2,3,len(results[0][0])))


for k_3 in range(2):
    if k_3 == 0:
        results = results_one
    else:
        results = results_three

    for k_1 in range(3):
        for k_2 in range(len(results[0][0])):
            mean_result[k_3][k_1][k_2] = np.mean(results[0][k_1][k_2])
            mean_error[k_3][k_1][k_2] = np.mean(results[1][k_1][k_2])

            # We again choose the 95 % level
            sigma_result[k_3][k_1][k_2] = 2 * np.sqrt(np.var(results[0][k_1][k_2]))
            sigma_error[k_3][k_1][k_2] = 2 * np.sqrt(np.var(results[1][k_1][k_2]))

            runtime[k_3][k_1][k_2] = np.mean(results[2][k_1][k_2])
            runtime_sig[k_3][k_1][k_2] = np.sqrt(np.var(results[2][k_1][k_2]))


# What can be found in the different dimensions of the matrices: shape: (2,3,23):
# x: x= 0 q_i uses one point, x=1 q_i uses three points;
# y: y=0 Monte Carlo (nested), y=1 Monte Carlo (non-nested), y=2 Quasi-Monte Carlo␣
↪(nested)
# z: element related to z = q-2
```

[15]:
```
from sklearn import datasets, linear_model
color_code = ["r-", "g-", "b-", "r--", "g--", "b--","r:", "g:", "b:"]
legend_label = ["Monte Carlo quadrature (nested)", "Monte Carlo quadrature (non-
↪nested)",
                "Quasi-Monte Carlo quadrature"]

fig = plt.figure(figsize=(15,6))

for k_1 in range(6):

    # solution not beautiful. Result of afterwards implementing algorithm with one␣
↪point used for q_i = 1.
    if k_1 < 3:
        k_2 = 1
```

(continues on next page)

(continued from previous page)

```
            k_1 = k_1 % 3
        else:
            k_2 = 0
            k_1 = k_1 % 3

        mean_error_real = abs(mean_result[k_2][k_1]-1)
        mean_sigma_real = sigma_result[k_2][k_1]
#     plt.subplots_adjust(hspace=0.4, wspace=0.4)

        degree_of_app = len(results[0][0])
        if k_2 == 0:
            plt.plot(list(range(2,2+degree_of_app)),mean_error_real,color_code[k_1],␣
↪label=legend_label[k_1])
        else:
            plt.plot(list(range(2,2+degree_of_app)),mean_error_real,color_code[k_1+3*k_2])

plt.ylim(min(abs(mean_result[1][2]-1)*0.5),max(abs(mean_result[0][0]-1)*1.5))
plt.yscale("log")
plt.ylim([1e-8,1])
plt.legend(fontsize=12, loc=1)
plt.xlabel("Degree of approximation q", fontsize = 14,fontweight = "bold")
plt.ylabel("mean error of the approximations",fontsize = 14,fontweight = "bold")
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.xticks(range(2,len(results[0][0])+2))
plt.title("Mean error of the approximation depending on the degree of approximation",
          fontsize = 20, fontweight = "bold")
plt.show()
```



This graphic shows the mean of the real error of the different approximations. The solid lines are related to the approximations using one point for q_i = 1, while the dotted lines were calculated for q_i = 3. The points for q near 24 are statistically speaking less significant because about 25 approximations were calculated.

```
[16]: fig = plt.figure(figsize=(15,6))
      for k_1 in range(6):
```

(continues on next page)

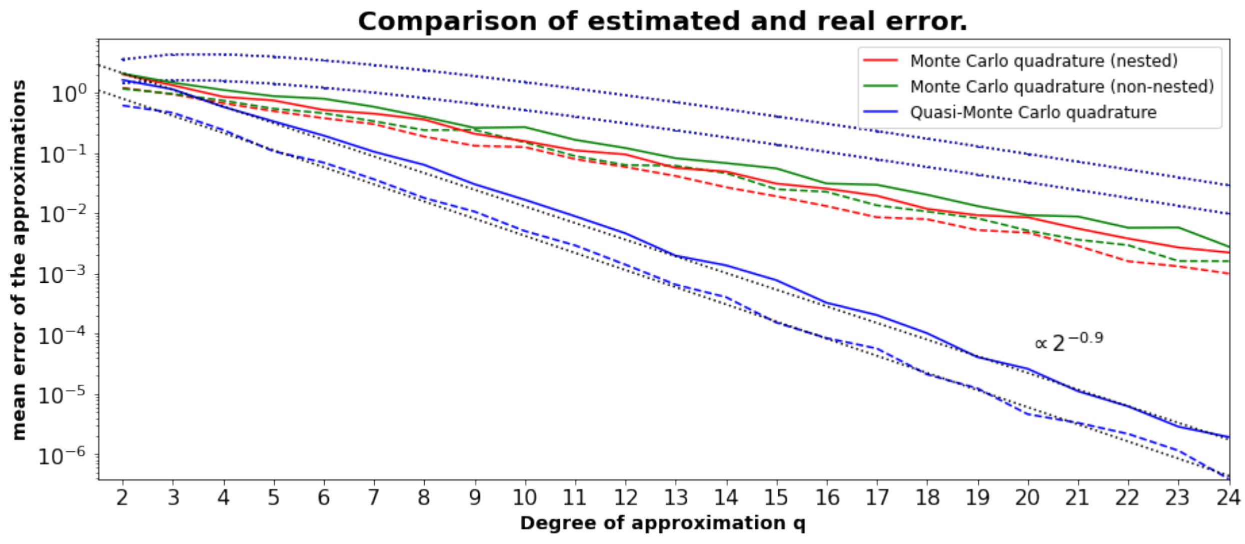
(continued from previous page)

```
    # solution not beautiful. Result of afterwards implementing algorithm with one␣
↪point used for q_i = 1.
    if k_1 < 3:
        k_2 = 1
        k_1 = k_1 % 3
    else:
        k_2 = 0
        k_1 = k_1 % 3
for k_1 in range(6):
        # solution not beautiful. Result of afterwards implementing algorithm with␣
↪one point used for q_i = 1.
    if k_1 < 3:
        k_2 = 1
        k_1 = k_1 % 3
    else:
        k_2 = 0
        k_1 = k_1 % 3

    mean_error_est  = mean_error[k_2][k_1]

    s_mean = sigma_result[k_2][k_1]
    s_error  = sigma_error[k_2][k_1]
    mean_error_real = abs(mean_result[k_2][k_1]-1)
    mean_sigma_real = sigma_result[k_2][k_1]


    degree_of_app = len(results[0][0])

    plt.errorbar(list(range(2,2+degree_of_app)), mean_error_est, s_error,ecolor=color_
↪code[k_1][0],
                 color=(0,0,0,0),marker="_",capsize=0.5)
    plt.plot(list(range(2,2+degree_of_app)),mean_error_est, color_code[k_1+6])
    if k_2 == 0:
        plt.plot(list(range(2,2+degree_of_app)),np.add(mean_error_real,s_mean),color_
↪code[k_1+ k_2 * 3],label=legend_label[k_1])
    else:
        plt.plot(list(range(2,2+degree_of_app)),np.add(mean_error_real,s_mean),color_
↪code[k_1+ k_2 *3])

    if k_1 == 2:
        regression_data_x = [[k_1] for k_1 in list(range(2,2+degree_of_app))]
        regression_data_y = [math.log2(k_1) for k_1 in np.add(mean_error_real,s_mean)]
        regr = linear_model.LinearRegression()
        regr.fit(regression_data_x , regression_data_y)
        regr.score(regression_data_x , regression_data_y)
        exponent = regr.coef_

        regression_data_x = [[k_1] for k_1 in list(range(2,2+degree_of_app))]
        regression_data_y = [math.log2(k_1) for k_1 in np.add(mean_error_real,s_mean)]
        regr = linear_model.LinearRegression()
        regr.fit(regression_data_x , regression_data_y)
        regr.score(regression_data_x , regression_data_y)
        exponent = regr.coef_

        regression_y = [np.add(mean_error_real,s_mean)[0]*1.3*2**(exponent[0]*(x -2))␣
↪ for x in list(range(0,20+degree_of_app))]
        plt.plot(list(range(0,20+degree_of_app)),regression_y, "k:")
```

(continues on next page)

(continued from previous page)

```
        if k_2 == 1:
            plt.annotate("$\propto 2^{-0.9}$", (20,regression_y[19]),xytext=(20,
↪5*10**(-5)), fontweight="bold",fontsize=16)

plt.ylim(min(abs(np.add(mean_error_real,s_mean))*0.2),max(abs(mean_error[1][2])*5))
plt.xlim(1.5,degree_of_app+0.5)
plt.yscale("log")
plt.legend(fontsize=12, loc=1)
plt.xlabel("Degree of approximation q", fontsize = 14,fontweight = "bold")
plt.ylabel("mean error of the approximations",fontsize = 14,fontweight = "bold")
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.xticks(range(2,len(results[0][0])+2))
plt.title("Comparison of estimated and real error.",
          fontsize = 20, fontweight = "bold")

plt.show()
```



The dotted lines are the error estimations for the different quadratures. The upper one is related to the calculations using one point for q_i = 1. The lower lines show the error within a confidence level of 95 %.”. The real error bound nearly decreases proportional to 2 ** (-1), if the Quasi-Monte Carlo quadrature is used. The exponent is varying a little with respect to the generated data used.

```
[18]: fig = plt.figure(figsize=(15,6))

      for k_1 in range(6):

          # solution not beautiful. Result of afterwards implementing algorithm with one␣
      ↪point used for q_i = 1.
          if k_1 < 3:
              k_2 = 1
              k_1 = k_1 % 3
          else:
              k_2 = 0
              k_1 = k_1 % 3
      for k_1 in range(6):
              # solution not beautiful. Result of afterwards implementing algorithm with␣
      ↪one point used for q_i = 1.
```

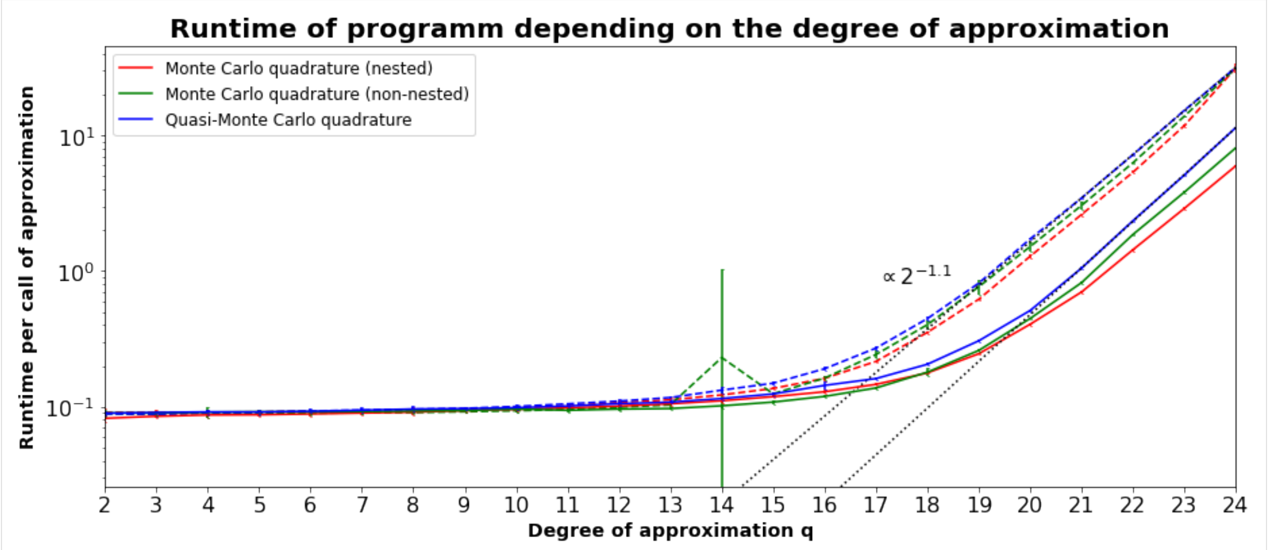
(continues on next page)

(continued from previous page)

```
    if k_1 < 3:
        k_2 = 1
        k_1 = k_1 % 3
    else:
        k_2 = 0
        k_1 = k_1 % 3
    runtime_mean = runtime[k_2][k_1]
    runtime_sigma = runtime_sig[k_2][k_1]

    degree_of_app = len(runtime_mean)

    if k_2 == 0:
        plt.plot(list(range(2,2+degree_of_app)),runtime_mean,color_code[k_1],␣
↪label=legend_label[k_1])
    else:
        plt.plot(list(range(2,2+degree_of_app)),runtime_mean,color_code[k_1 + 3])
    plt.errorbar(list(range(2,2+degree_of_app)), runtime_mean, runtime_sigma,
↪ecolor=color_code[k_1][0],
                 color=(0,0,0,0),marker="_",capsize=0.5)
    if k_1 == 2:


        regression_data_x = [[k_1] for k_1 in list(range(2,2+degree_of_app))]
        regression_data_y = [math.log2(k_1) for k_1 in runtime_mean]
        regr = linear_model.LinearRegression()
        regr.fit(regression_data_x , regression_data_y)
        regr.score(regression_data_x , regression_data_y)
        exponent = regr.coef_


        regression_data_x = [[k_1] for k_1 in list(range(21,2+degree_of_app))]
        regression_data_y = [math.log2(k_1) for k_1 in runtime_mean[19:degree_of_app]]
        regr = linear_model.LinearRegression()
        regr.fit(regression_data_x , regression_data_y)
        regr.score(regression_data_x , regression_data_y)
        exponent = regr.coef_
        regression_y = [2** (regr.intercept_ + exponent*x) for x in list(range(0,
↪20+degree_of_app))]
        plt.plot(list(range(0,20+degree_of_app)),regression_y, "k:")

        if k_2 == 1:
            plt.annotate("$\propto 2^{-1.1}$", (20,regression_y[19]),xytext=(17,
↪regression_y[19]), fontweight="bold",fontsize=16)

plt.xlim(2,24)
plt.ylim(min(abs(runtime[0][2]-1)*0.5),max(abs(runtime[1][2]-1)*1.5))
plt.yscale("log")
plt.legend(fontsize=12, loc=2)
plt.xlabel("Degree of approximation q", fontsize = 14,fontweight = "bold")
plt.ylabel("Runtime per call of approximation",fontsize = 14,fontweight = "bold")
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.xticks(range(2,len(results[0][0])+2))
plt.title("Runtime of programm depending on the degree of approximation",
          fontsize = 20, fontweight = "bold")
plt.show()
```

(continues on next page)

(continued from previous page)



It could be seen that the difference between the three quadratures is minor for this example. The variance of the runtime is relatively low. The observed runtime did not change fundamentally for changes in the implementation, such that this might be a related to the algorithm itself.

4.2.1 Code used for generation of evaluated data.

```
[1]: '''
import Methodes_Studienproject.Studienprojekt_Smolyak_qmc_one_point as Studieproject_
↪one

Studieproject_one.calculate_stat_data("pi^2/4 sin(pi*x)sin(pi*y)", "(x,y)", ["Monte␣
↪Carlo (nested)","Monte Carlo (non-nested)","Quasi-Monte Carlo"]
   , "stat_approach_2_to_20_result_1_12_one.csv",
                                    [2, 25], repetitions=2 ** 10, damping=True,␣
↪damping_exp=1.2)
'''

'''
import Methodes_Studienproject.Studienprojekt_Smolyak_qmc_three_points as␣
↪Studieproject_three

Studieproject_three.calculate_stat_data("pi^2/4 sin(pi*x)sin(pi*y)", "(x,y)", ["Monte␣
↪Carlo (nested)", "Monte Carlo (non-nested)","Quasi-Monte Carlo"]
    , "stat_approach_2_to_20_result_1_12_three.csv", [2, 23], repetitions=2 ** 10,␣
↪damping=True,
                      damping_exp=1.2)
'''
print("Approximation of analytical function in 2 dimensions. (Three points)")
```

```
Approximation of analytical function in 2 dimensions. (Three points)
```

```
[ ]:
```

CHAPTER

FIVE

EXAMPLE 15.2.6: PERIODIC FUNCTIONS (MODIFIED)

This is an easy example of a function fulfilling those conditions that make it possible to apply the results of example 15.2.6. In the example the classical Wiener measure is used. This measure sofar is not implemented in this project, such that the results of of this section do not hold for these examples. Nevertheless, the results are compared with the results using the Lebesgue measure. If the use of the Wiener measure is implemented, the example easily could be modified.

In example 15.2.6. in one dimension it is assumed that you have a r-times continuously differentiable function with period β. Then the norm of this space $F_1 = \tilde{C}^r([0,\beta])$ could be written as

$$||f||_{F_1} = |f(0)| + \max_{t\in[0,\beta]} |f^{(r)}(t)|$$

.

If we now assume $f^{(i)}(0) = f^{(i)}(\beta) = 0$ for all $i \in 1, 2, \ldots, r$, we have for the d dimensional case

$$||f||_{F_d} = \max_{t\in[0,\beta]^d} |f^{(r,r,\ldots,r)}(t)|$$

.

Using the information $N_i(f) = [f(\frac{\beta}{m_i+1}), f(\frac{2\cdot\beta}{m_i+1}), \ldots, f(\frac{m_i\cdot\beta}{m_i+1})]$ we use the algorithm

$$U_i(f) = \frac{\beta}{m_i} \sum_{j_1}^{m_i} f\left(\frac{j\beta}{m_1+1}\right)$$

to approximate the integral. This coincides with the open trapezoidal quadrature. The rest of the properties of the approximation are explained afterwards.

First we take a quick look at the function

$$f_1(x,t) = \min(x,t),$$

which is in $\tilde{C}^0([0,1])$. It is possible to calculate that the

$$\int_{[0,1]^2} f_1(\vec{x})d\vec{x} = \frac{1}{3}$$

holds. The real cost of the reaching of a specific error is compared with the estimated cost in the rear part of the section.

After this we take a look at some easy functions $\tilde{C}^{r-1}([0,1])$, with $r > 0$.

Here the one dimensional function we chose is

$$f_r(x) = \left(\prod_{k=1}^{\lfloor r/2\rfloor} \frac{2k-1}{2k}\right) \cdot \sin^r(2\pi \cdot x),$$

which especially is in $\tilde{C}^{r-1}([0,1])$, with $f(0) = f(1) = 1$. The result of the integral $:nbsphinx-math:*int{[0,1]}{d}}:nbsphinx-math:`prod`{d}*{k=1} 2 ·‘ : $nbsphinx - math : sin$‘^2(2 : $nbsphinx - math$: *pi `:nbsphinx-math:cdotx_k)d:nbsphinx-math:vec*$\{x\}$‘$couldbedetermineanalyticallytobe1forevenr$ and $d \in \mathbb{N}$ and 0 for uneven r and $d \in \mathbb{N}$.

[1]:
```
import os
os.chdir("..")


import Methodes_Studienproject.Studienprojekt_Smolyak_qmc_one_point as Studieproject_
↪one
import matplotlib.pyplot as plt
import pandas as bearcats
import scipy
import sympy as sp
import numpy as np
import itertools as itt
import math
from sklearn import datasets, linear_model
plt.rc('font', size=14, weight="bold")
```

5.1 Example on page 358

First we show the calculate the approximation of $f_1(x,t)$ and the approximation given on page 358.

[11]:
```
from IPython.display import display, Math

# other properties of the example
r = 0
beta = 1
q = 2
dim = 2
m_i = 2 ** q - 1

# Specific error properties of periodic functions of this kind
C_r = np.sqrt(abs(scipy.special.bernoulli(2+2*r)[-1])/math.factorial(2*r+2))
e_one_a =  C_r * beta ** ((2*r+3)/2)
e_one_b = 2 ** (-(r + 1))

# Cost and error esimation for the one dimensional error
B, C, D, F_0, F = [e_one_a, e_one_a, e_one_b, 1, 2]

# calculation of the approximation and the error and cost estimation
# of the example
option = "Trapezoidal"
deg_approx = 18
def max_fun(x, t):
# implementation of the function given on page 358 for input like used in
# implementation of algorithm.
    return np.min(np.array([x,t]), axis=0)

variables = []
for f in range(dim):
    variables.append(sp.Symbol("x_" + str(f)))
```

(continues on next page)

(continued from previous page)

```
result_max_fun = np.empty((deg_approx,2))
cost_and_err_estim = np.empty((20,2))
for eps_ind, epsilon in enumerate([(1/2) ** (i+3) for i in range(20)]):
    m_i = 2 ** q - 1
    cost_and_err_estim[eps_ind,:] = Studieproject_one.fast_eps_cost(dim, B, C, D, F_0,
↪ F, epsilon)[0::6]

for q in range(dim, dim + deg_approx):
    result_max_fun[q-2, :] = Studieproject_one.controller_smolyak(max_fun, variables,␣
↪option, q, function_given=True, no_error=True, example_2=True)[0:3:2]

result_max_fun[:,0] = abs(result_max_fun[:,0] -1/3)
```

We here want to show that the function calculating the estimation works properly.

```
[12]: estim_cost_error =Studieproject_one.fast_eps_cost(dim, B, C, D, F_0, F, 1/8)
meaning =[r"cost(A_{\epsilon}(d))", r"\alpha", r"\alpha_0", r"\alpha_1", r"\alpha_2",␣
↪r"\alpha_3"]

display(Math(r" $We chose $\epsilon"+r"$ = $ \frac{1}{8}"))
print("This leads to:")
for index, var in enumerate(meaning):
    display(Math(var+"$ = $" +str(round(estim_cost_error[index],4))))
    print("")
display(Math(" $This coincides with the values diplayed on the pages 368 and 369. For␣
↪bigger values of  $ \epsilon, $ the cost-estimation is not a number. In this␣
↪reagion the estimation will be seen to be bigger than actual values.$ "))
```

We chose $\epsilon = \frac{1}{8}$

```
This leads to:
```

$cost(A_{\epsilon}(d)) = 0.4141$

$\alpha = 1.0$

$\alpha_0 = 3.3902$

$\alpha_1 = -0.5762$

$\alpha_2 = 2.7098$

$\alpha_3 = 1.5$

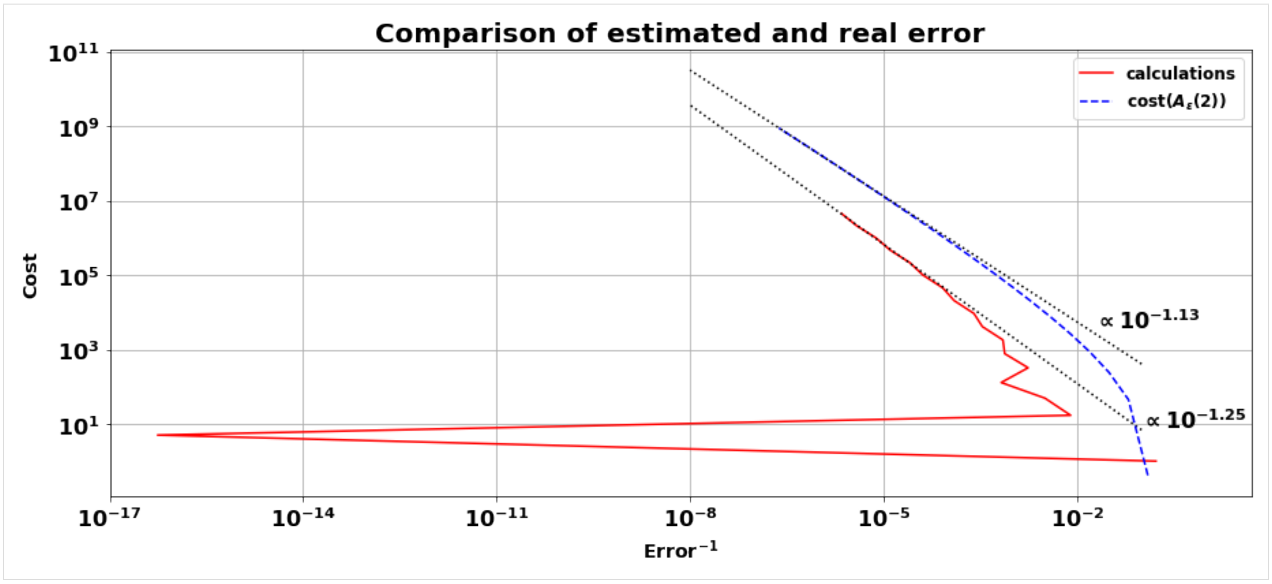
This coincides with the values diplayed on the pages 368 and 369. For bigger values of ϵ, the cost-estimation is not a number. In this reagion the estimation will be seen to be bigger than actual values.

```
[36]: fig = plt.figure(figsize=(15,6))
plt.plot( abs(result_max_fun[:,0]), result_max_fun[:,1], color="red", label=
↪"calculations")
regression_data_x = [[k_1] for k_1 in np.log(abs(result_max_fun[-5:,0]))]
```

(continues on next page)

(continued from previous page)

```
regression_data_y = [[k_2] for k_2 in np.log(abs(result_max_fun[-5:,1]))]
regr = linear_model.LinearRegression()
regr.fit(regression_data_x , regression_data_y)
regr.score(regression_data_x , regression_data_y)
exponent = regr.coef_

regression_y = [np.e**(regr.intercept_ + exponent[0][0] * np.log(x)) for x in np.
↪linspace(10**(-1),10**(-8),30)]
plt.plot(np.linspace(10**(-1),10**(-8),30),regression_y, "k:")
plt.annotate("$\propto 10^{-1.25}$", (np.linspace(10**(-1),10**(-8),30)[2],regression_
↪y[1]),xytext=(np.linspace(10**(-1),10**(-8),30)[2],regression_y[1]*1.1), fontweight=
↪"bold",fontsize=16)

plt.plot( [(1/2) ** (i+3) for i in range(20)], cost_and_err_estim[:,0], color="blue",␣
↪linestyle="--",label="cost($A_{\epsilon}(2)$)")
regression_data_x = [[k_1] for k_1 in np.log10([(2) ** (-i-3) for i in range(15,20)])]
regression_data_y = [[k_2] for k_2 in np.log10(cost_and_err_estim[-5:,0])]
regr = linear_model.LinearRegression()
regr.fit(regression_data_x , regression_data_y)
regr.score(regression_data_x , regression_data_y)
exponent = regr.coef_

regression_y = [10**(regr.intercept_ + exponent[0][0] * np.log10(x)) for x in np.
↪linspace(10**(-1),10**(-8),30)]
plt.plot(np.linspace(10**(-1),10**(-8),30),regression_y, "k:")
plt.annotate("$\propto 10^{-1.13}$", (np.linspace(10**(-1),10**(-8),30)[25],
↪regression_y[25]),xytext=(np.linspace(10**(-1),10**(-8),30)[25]*1.3,regression_
↪y[25]), fontweight="bold",fontsize=16)

plt.xlim(10**(-17), 5)
plt.yscale("log")
plt.xscale("log")
plt.grid(axis="both")
plt.legend(fontsize=12)
plt.ylabel("Cost", fontsize = 14,fontweight = "bold")
plt.xlabel(r"Error$^{-1}$",fontsize = 14,fontweight = "bold")
plt.yticks(fontsize=16,fontweight="bold")
plt.xticks(fontsize=16,fontweight="bold")
plt.title("Comparison of estimated and real error",
          fontsize = 20, fontweight = "bold")
plt.show()
```



The error scales in a surprisingly nice way, such that the error bound holds for the function $f_1(\vec{x})$ for almost every point. We once again remind the reader that the error bound could not be applied to the calculation without changing the measure used in this project.

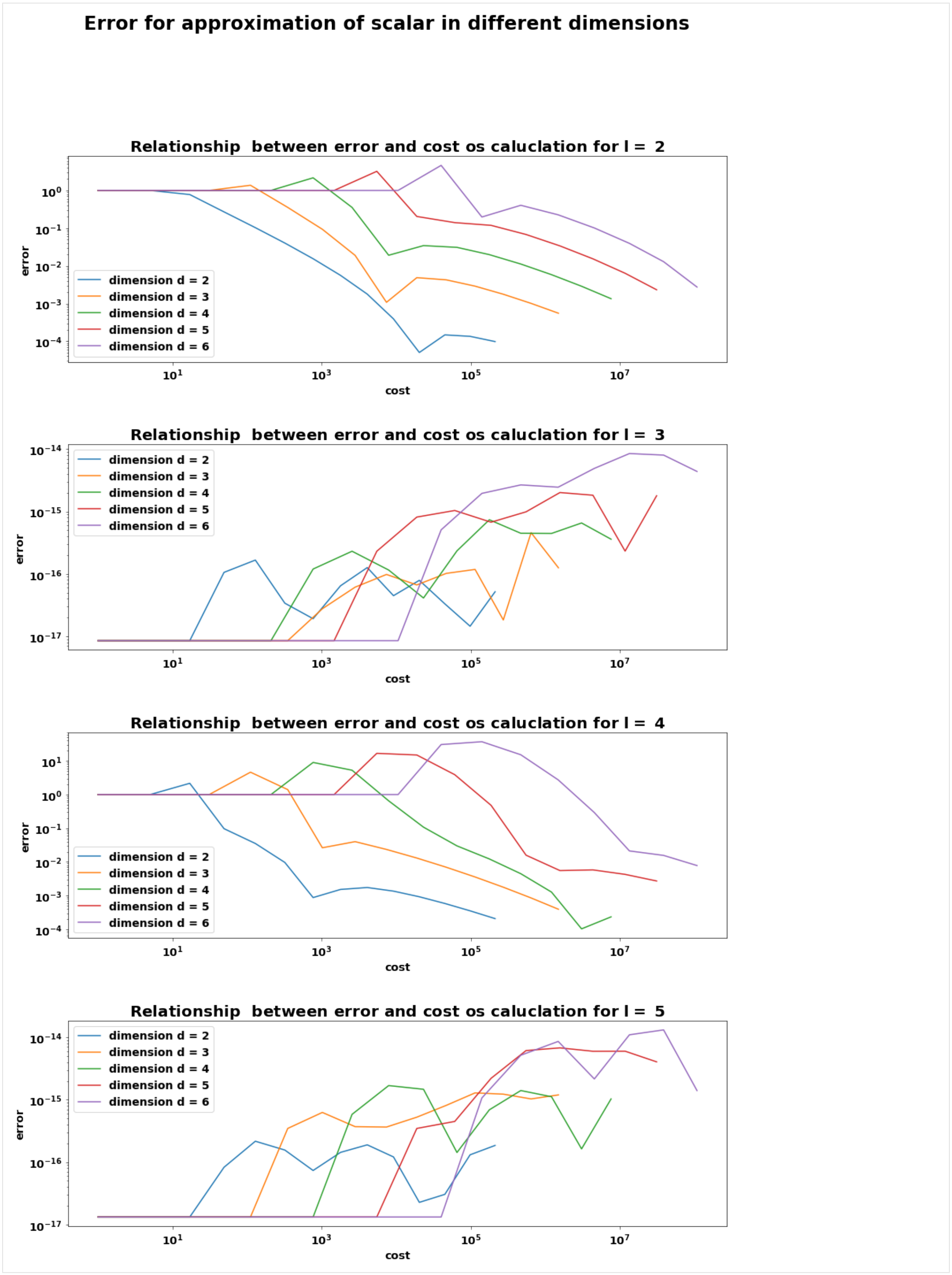
5.2 Example 2: Functions in $f \in \tilde{C}^r([0,1])$ with $r > 0$ and higher dimensions

[6]:
```
# We from now on use l instead of r to make clear that the sofar the results of␣
↪section 15.2.6 could not be applied.

beta = 1


try:
    os.chdir("Data")
except:
    pass

color_str = ["navy", "crimson", "forestgreen", "cornflowerblue", "darkorange"]

# Structure of data: [r, dim,q,:]
result_per_func = bearcats.read_pickle("approx_per_function.pkl")
result_per_func = result_per_func['results']#[:-1,:,:,:]

fig, ax = plt.subplots(result_per_func.shape[0],figsize=(15,25))


for l in range(result_per_func.shape[0]):
    result_per_func_l = result_per_func[l,:,:,:]
    l_real = 0

    ''' These calculations are not made because the estimations are not met.
    In further implementation it might be possible to use them
```

(continues on next page)

(continued from previous page)

```
    # Specific error properties of periodic functions of this kind
    C_r = np.sqrt(abs(scipy.special.bernoulli(2+2*r_real)[-1])/math.factorial(2*r_
↪real+2))
    e_one_a =  C_r * beta ** ((2*r_real+3)/2)
    e_one_b = 2 ** (-(r_real + 1))

    # Cost and error esimation for the one dimensional error
    B, C, D, F_0, F = [e_one_a, e_one_a, e_one_b, 1, 2]
    '''

    for dim in range(result_per_func_l.shape[0]):
        dim_real = dim+2
        # We want to take a look at the error, not the value of the approximation.
        if l%2==0:
            error = abs(result_per_func_l[dim,:,0]-1)
        else:
             error = abs(result_per_func_l[dim,:,0])+np.max(abs(result_per_func_l[:,:,
↪0]))/1000

        cost  = result_per_func_l[dim,:,2]

        '''
        cost_estim = []
        epsilons = [(1/2)**(20*l+k_1) for k_1 in range(error.shape[0])]
        for eps in epsilons:
            cost_estim.append(Studieproject_one.fast_eps_cost(dim_real, B, C, D, F_0,␣
↪F, eps)[0])
        '''

        plt.subplots_adjust(hspace=0.4, wspace=0.4)
        ax[l].plot(cost, error, label="dimension d = "+ str(dim+2))

    ax[l].set_xscale("log")
    ax[l].set_yscale("log")
    ax[l].legend()
    ax[l].set_title(r"Relationship  between error and cost os caluclation for $\mathbf
↪{l} = $ " + str(l+2),fontsize=20, fontweight="bold")
    ax[l].set_xlabel("cost",fontsize=14, fontweight="bold")
    ax[l].set_ylabel("error",fontsize=14, fontweight="bold")

fig.suptitle("Error for approximation of scalar in different dimensions", fontsize=24,
↪fontweight="bold")

plt.show()
```


Error for approximation of scalar in different dimensions
Relationship between error and cost os caluclation for l = 2
Relationship between error and cost os caluclation for l = 3
Relationship between error and cost os caluclation for l = 4
Relationship between error and cost os caluclation for l = 5
error
cost
dimension d = 2
dimension d = 3
dimension d = 4
dimension d = 5
dimension d = 6
10^1
10^3
10^5
10^7
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-14}
10^{-15}
10^{-16}
10^{-17}

For l odd the results were shifted, because for small q the error is 0. Because of the symmetry of the function and the points used for the calculation the only source of an inexact approximation is the machine error.

For l even it could easily be seen that the results of section 15.2.6 could not be applied, because the the approximation would need to have the tendency to get more precise with increasing r. This could not be observed.

```
[2]: '''
option = "Trapezoidal"
deg_approx = 15
max_dim = 6
r_max = 5

results_example_2= np.empty((r_max-1 , max_dim-1, deg_approx, 3))
for r in range(2, r_max+1):
    print("  ")
    norm_factor = 1
    variables = ["(x_1)"]
    for k in range(1,int(np.ceil(r/2))+1):
        norm_factor = norm_factor * (2*k/(2*k-1))

    functions = [str(norm_factor) + " sin(x_"+str(1)+"*2*pi)** "+str(r)]



    for dim in range(2,max_dim+1):

        variables.append(variables[-1][:-1]+", x_"+ str(dim)+")")
        functions.append(functions[-1]+  " * " + str(norm_factor) + "* sin(x_
↪"+str(dim)+"*2*pi)** " + str(r))

        for q in range(dim,dim+deg_approx):
            print(str(r) + "  " +str(dim)+ "   " + str(q))
            results_example_2[r-2,dim-2, q - dim, :] = Studieproject_one.controller_
↪smolyak(functions[-1], variables[-1], option, q, example_2=True)


results_example_2 = { "results": results_example_2}

bearcats.to_pickle(results_example_2,"approx_per_function.pkl")

print("If you want to generate data, remove quotation marks.")
'''
print("Remove quotes to generate data.")
```

```
Remove quotes to generate data.
```

```
[ ]:
```

CHAPTER

SIX

EXAMPLE 15.3.8 PERTUBATED COULOMB POTENTIAL (PCP)

We want to implement the example 15.3.8 in the book Tractability of Multivariate Problems Vol 2 for 2 particles The function itself is

$$f(x_1, x_2, \dots, x_l) = \sum_{1 \le i < j \le l} \left(\frac{1}{\sqrt{||x_i - x_j||_2^2 + \alpha}} \right)$$

Where x_i, i = 1,2,…,l are three dimensional vectors. We first choose l = 2. Using the results of the example we calculate the function by integrating

$$\tilde{f}_\alpha(z_1, z_2, \dots, z_{3l}) = \left(\prod_{j=1}^{6-1} \left(j + \frac{1}{2} \right) \right) \cdot \frac{(z_1 - z_4)^4 (z_2 - z_5)^4 (z_3 - z_6)^4}{((z_1 - z_4)^2 + (z_2 - z_5)^2 + (z_3 - z_6)^2 + \alpha)^{13}}$$

over the interval [0,1]^6. For higher dimension the algorithm gets extremely slow.

```
[3]: import matplotlib.pyplot as plt
     import matplotlib.colors as colors
     import time as time
     import numpy as np
     import pandas as bearcats
     import scipy.integrate
     import sympy

     import os
     os.chdir("..")
     import Methodes_Studienproject.Studienprojekt_Smolyak_qmc_one_point as Studieproject_
     ↪one
     import Methodes_Studienproject.Studienprojekt_Smolyak_qmc_three_points as␣
     ↪Studienprojekt_three
```

6.1 Results of approximations

We calculated the integral for $\alpha = 10^{-i}$ for in $i = 0, \dots, 5$ using different quadratures up to a degree of approximation of up to 23, depending on the runtime of the algorithm. Furthermore it was possible to calculate the integral for $i = 0$ and $i = 1$ in an acceptable amount of time and with a sufficient accuracy. On this basis we also show a lower bound of the integral for the smaller α which was estimated on page 393 of the same book. First we show the results of the approximation.

```
[26]: try:
          os.chdir("Data")
      except:
          pass
      # Settings needed for plotting of results.
      option_list = ["Newton-Cotes",
                     "Trapezoidal",
                     "Monte Carlo (non-nested)",
                      "Quasi-Monte Carlo"]
      color_str = [ "fuchsia", "navy", "limegreen", "red"]
      alpha = [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]

      # Load the result the integration of the function calculated by scipy methode nquad
      result_scipy_calc_basic = bearcats.read_pickle("integral_Coulomb_alpha_
       ↪"+str(alpha[0])+".pkl")


      for k_1 in range(len(alpha)):
          # Load single approximation of the PCP different alphas and a wider range of␣
       ↪degree of approximation q.
          data = bearcats.read_pickle("test_higher_dim_approx_alpha_"+ str(alpha[k_1]) +".
       ↪pkl")

          # Integral could only be  calculated relative exactly by the nquad for the first␣
       ↪2 alphas.
          if k_1 < 2:
              result_scipy_calc = bearcats.read_pickle("integral_Coulomb_alpha_
       ↪"+str(alpha[k_1])+".pkl")
          # For alpha < 1 the PCP was approximated using the trapezoidal quadrature and q =␣
       ↪19. Higher approximations
          # would easily have taken about 5 hours.
          if k_1  >0:
              single_approx_trap = bearcats.read_pickle("test_higher_dim_approx_one_alpha_"␣
       ↪+ str(alpha[k_1]) + "_only_trap.pkl").Trapezoidal[0]

          fig = plt.figure(figsize=(15,12))
          plt.subplots_adjust(hspace=0.4, wspace=0.4)

          #  First figure showing the absolut value of the approx in a logarithmical scale
          plt.subplot(2,1,1)


          for k_2 in range(len(option_list)):

              approx = data[option_list[k_2]][0][:17,0]
              error = data[option_list[k_2]][0][:17,1]

              end_of_approx = np.where(approx==0)[0][0]

              if end_of_approx ==0:
                  end_of_approx = np.where(approx==0)[0][-1]+1

              if k_2 < 2:
                  plt.plot(list(range(6,6+end_of_approx)), abs(approx[:end_of_approx]),␣
       ↪color = color_str[k_2], label=option_list[k_2])

              else:
```

(continues on next page)

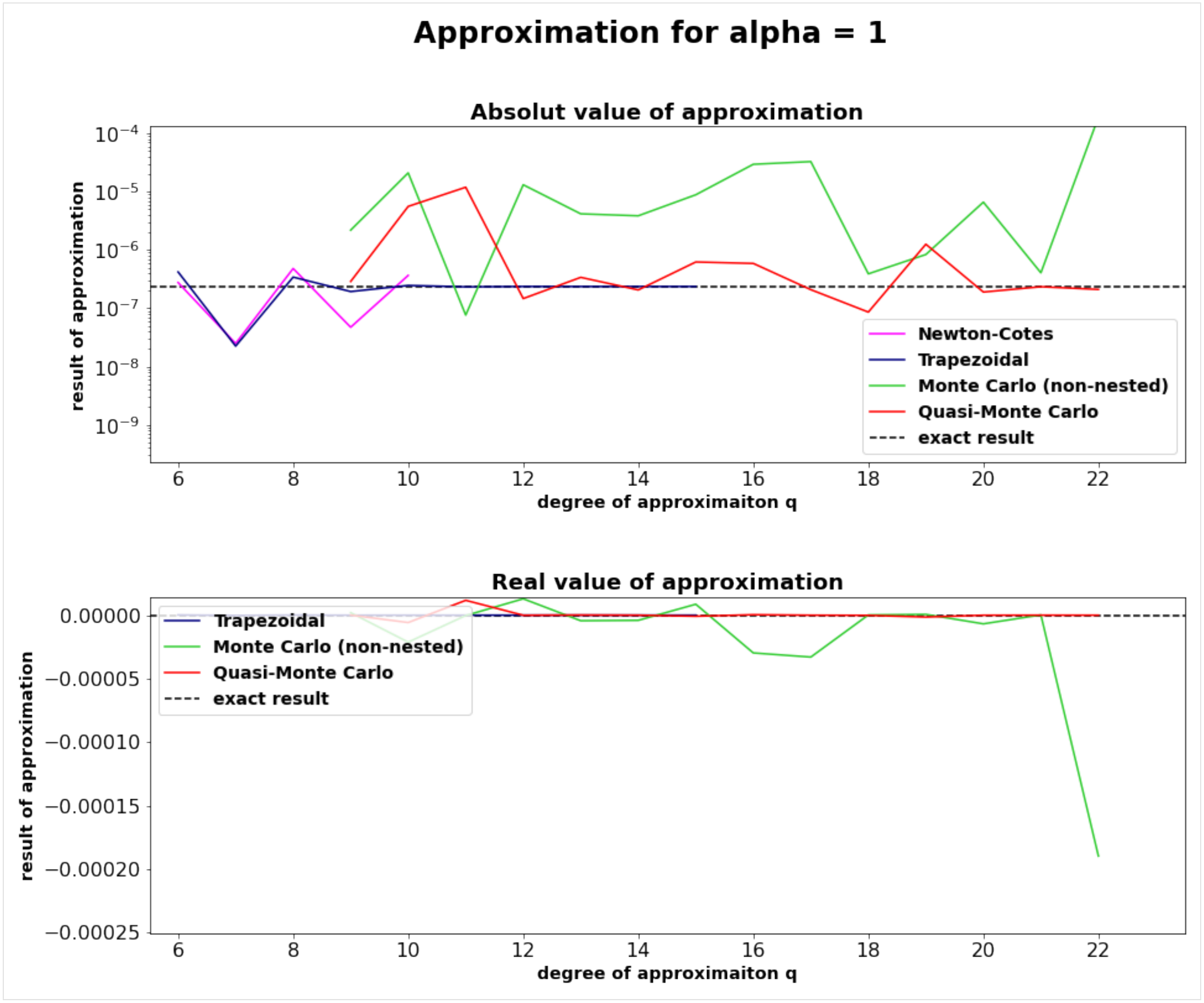
(continued from previous page)

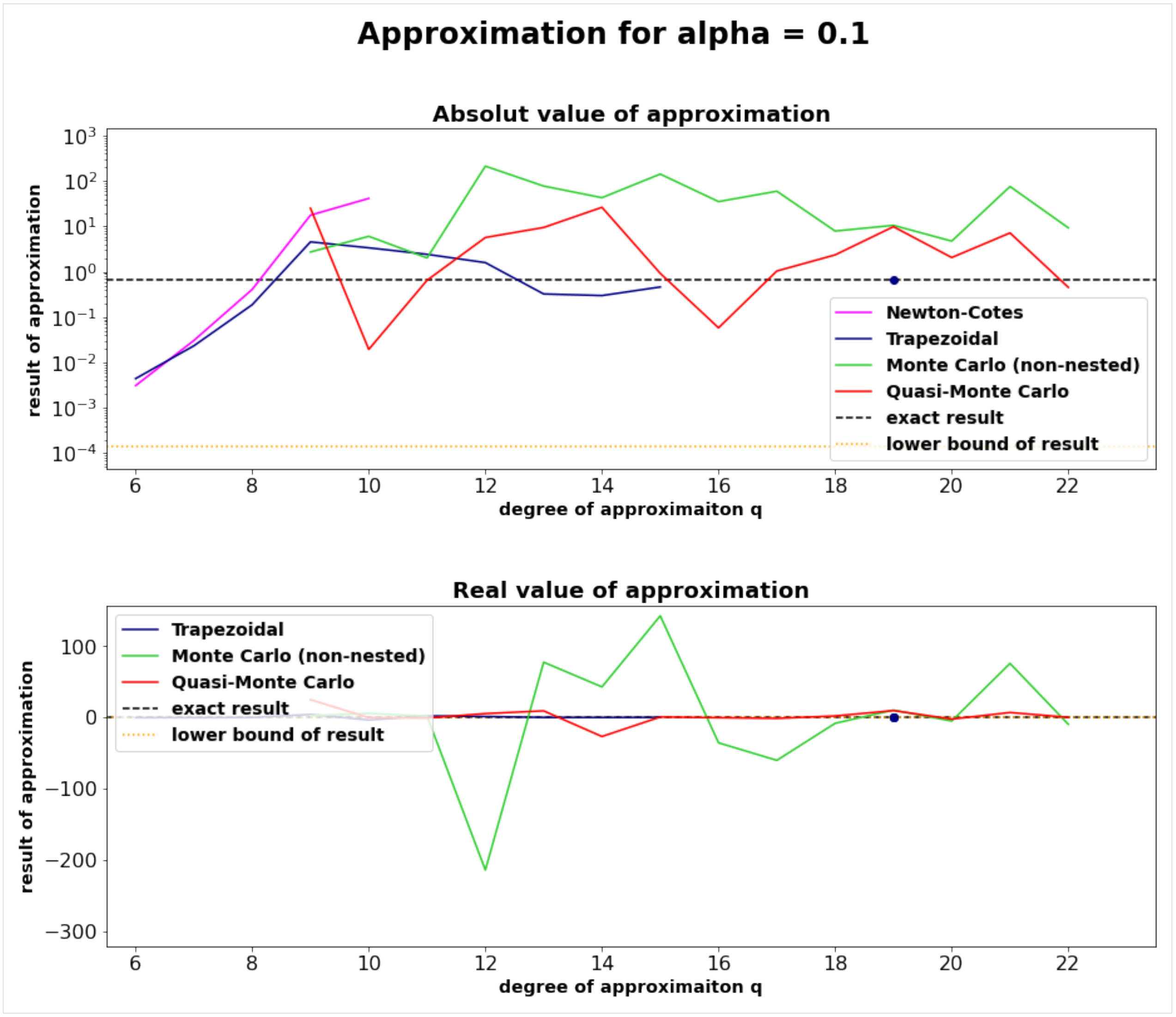
```
            plt.plot(list(range(6+end_of_approx,6+17)), abs(approx[end_of_approx:]),␣
↪color = color_str[k_2], label=option_list[k_2])

    if k_1 >0:
        plt.plot(6+13, abs(single_approx_trap[0]), color="navy", marker="o")

    if k_1 <2:
        plt.hlines(result_scipy_calc[0], xmin =  5.5, xmax= 6.5+17, linestyles="--",␣
↪color= "black", label="exact result")

    if k_1 > 0:
        plt.hlines(result_scipy_calc_basic[0]*6*(0.1**(-2))**int(np.log10(alpha[0]/
↪alpha[k_1])), xmin =  5.5, xmax= 6.5+17, linestyles=":", color= "orange", label=
↪"lower bound of result")

    plt.xlim(5.5,6.5+17)
    plt.ylim(np.min(abs(data[option_list[1]][0][np.where(data[option_list[1]][0][:,0]!
↪=0)][:,0]))
                 /100,np.max(data[option_list[2]][0][:,0])*10)
    plt.legend(loc=4, prop={"size":14, "weight":"bold"})
    plt.ylabel("result of approximation",fontsize = 14,fontweight = "bold")
    plt.xlabel("degree of approximaiton q",fontsize = 14,fontweight = "bold")
    plt.yscale("log")
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)
    plt.title("Absolut value of approximation" , fontsize=18,fontweight="bold")

    # Second subplot showing the absolut values.
    plt.subplot(2,1,2)

    for k_2 in range(1, len(option_list)):
        approx = data[option_list[k_2]][0][:17,0]
        error = data[option_list[k_2]][0][:17,1]
        end_of_approx = np.where(approx==0)[0][0]

        if end_of_approx ==0:
            end_of_approx = np.where(approx==0)[0][-1]+1

        if k_2 < 2:
            plt.plot(list(range(6,6+end_of_approx)), approx[:end_of_approx], color =␣
↪color_str[k_2], label=option_list[k_2])

        else:
            plt.plot(list(range(6+end_of_approx,6+17)), approx[end_of_approx:], color␣
↪= color_str[k_2], label=option_list[k_2])

        if k_1 >0:
            plt.plot(6+13, single_approx_trap[0], color="navy", marker="o")

    plt.xlim(5.5,6.5+17)
    plt.ylim(np.min([np.max(data[option_list[2]][0][np.where(data[option_list[2]][0][:
↪17,0]!=0),0])- abs(
        np.min(data[option_list[2]][0][np.where(data[option_list[2]][0][:17,0]!=0),
↪0])-np.max(data[option_list[2]][0][
            np.where(data[option_list[2]][0][:17,0]!=0),0]))*1.3,0]),np.max([np.
↪max(data[option_list[2]][0][:17,0])*1.1,np.max(data[option_list[3]][0][:17,0])*1.
↪1]))
```

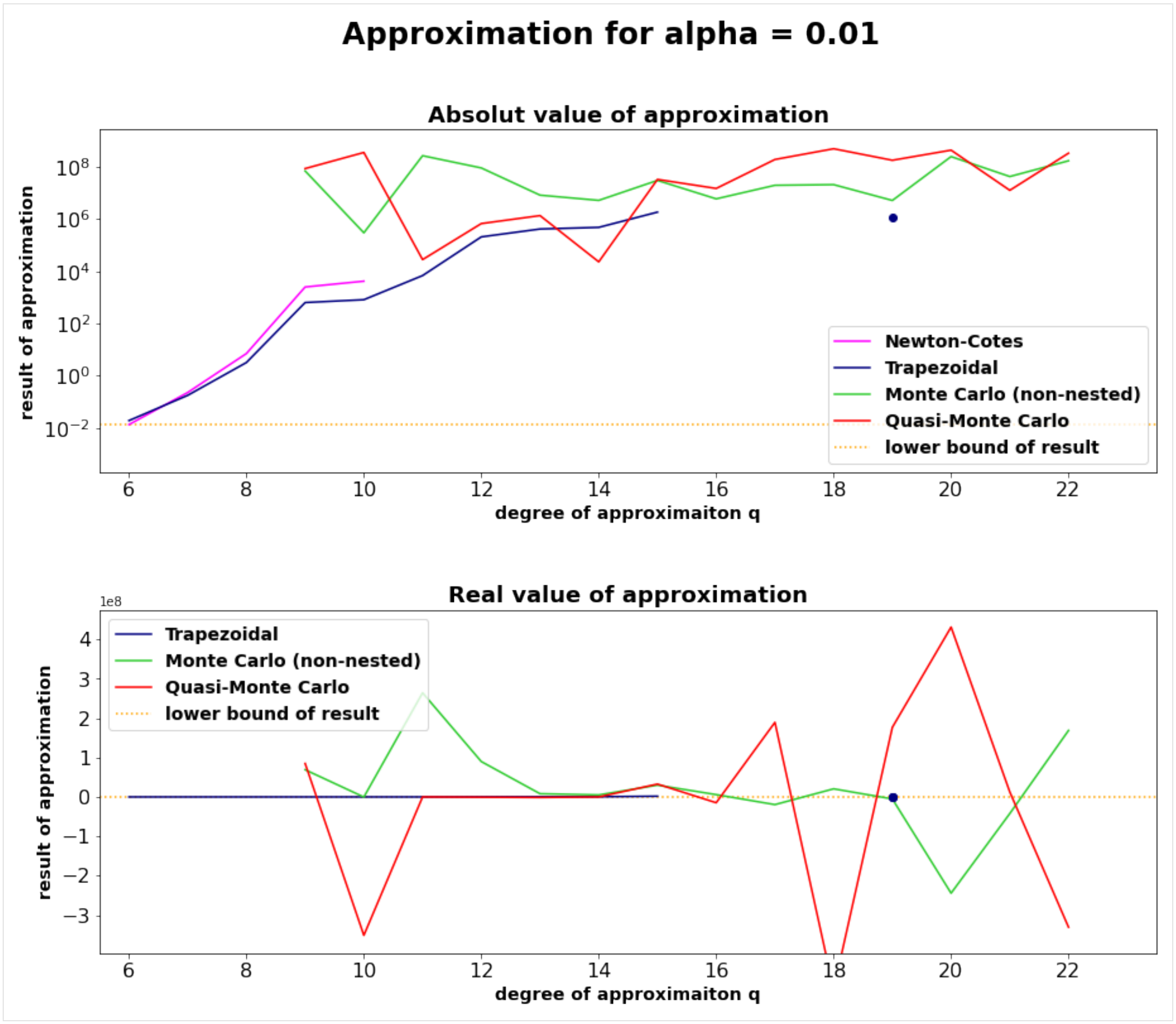
(continues on next page)

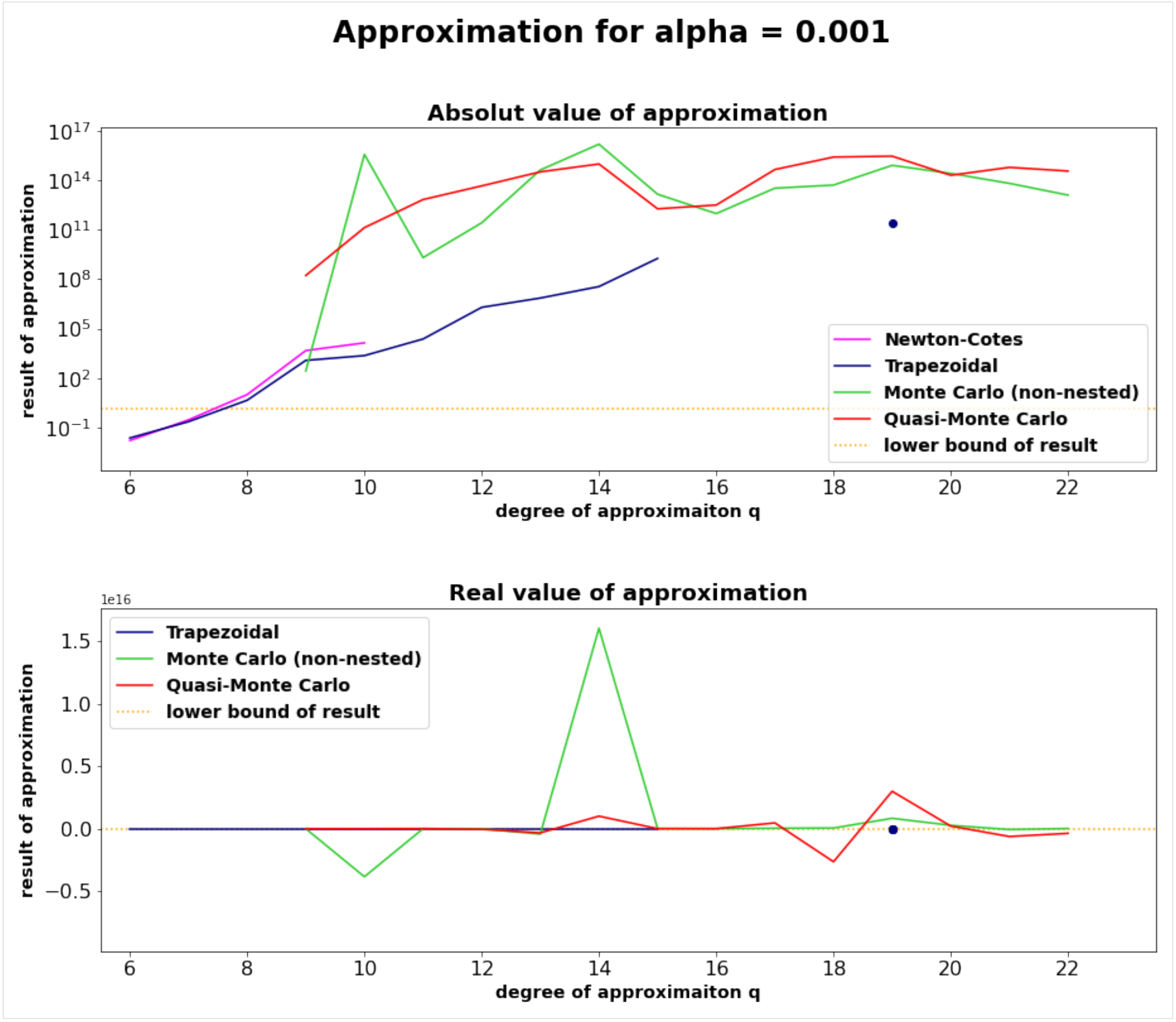
(continued from previous page)

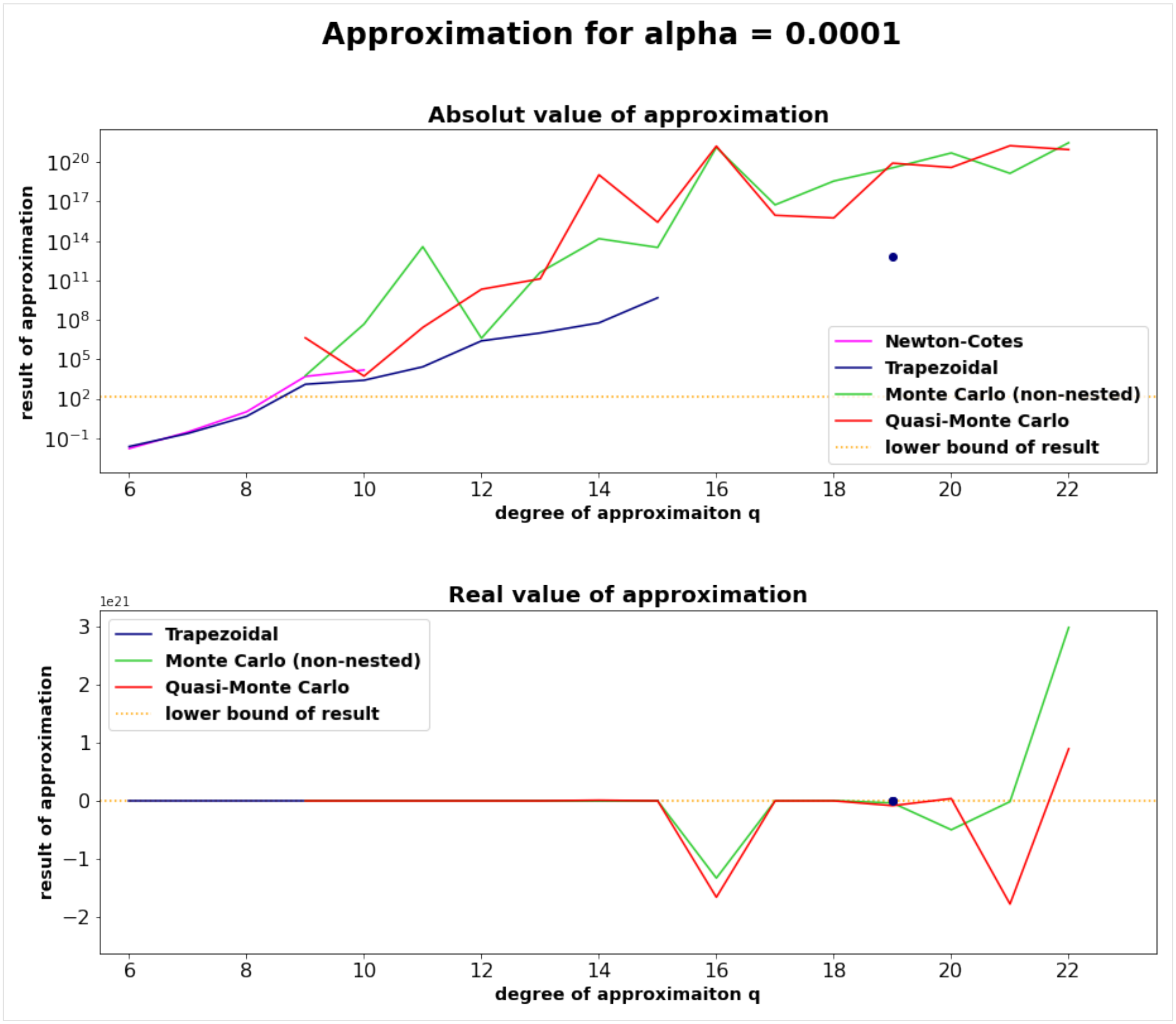
```
    if k_1 < 2:
        plt.hlines(result_scipy_calc[0], xmin =  5.5, xmax= 6.5+17, linestyles="--",␣
↪color= "black", label="exact result")

    if k_1 > 0:
        plt.hlines(result_scipy_calc_basic[0]*6*(0.1**(-2))**int(np.log10(alpha[0]/
↪alpha[k_1])), xmin =  5.5, xmax= 6.5+17, linestyles=":", color= "orange", label=
↪"lower bound of result")

    plt.legend(loc=2, prop={"size":14, "weight":"bold"})
    plt.ylabel("result of approximation",fontsize = 14,fontweight = "bold")
    plt.xlabel("degree of approximaiton q",fontsize = 14,fontweight = "bold")
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)
    plt.ticklabel_format(style="sci", axis="both")
    plt.title("Real value of approximation", fontsize=18,fontweight="bold")

    fig.suptitle("Approximation for alpha = " + str(alpha[k_1]), fontsize=24,
↪fontweight="bold")



    plt.show()
```

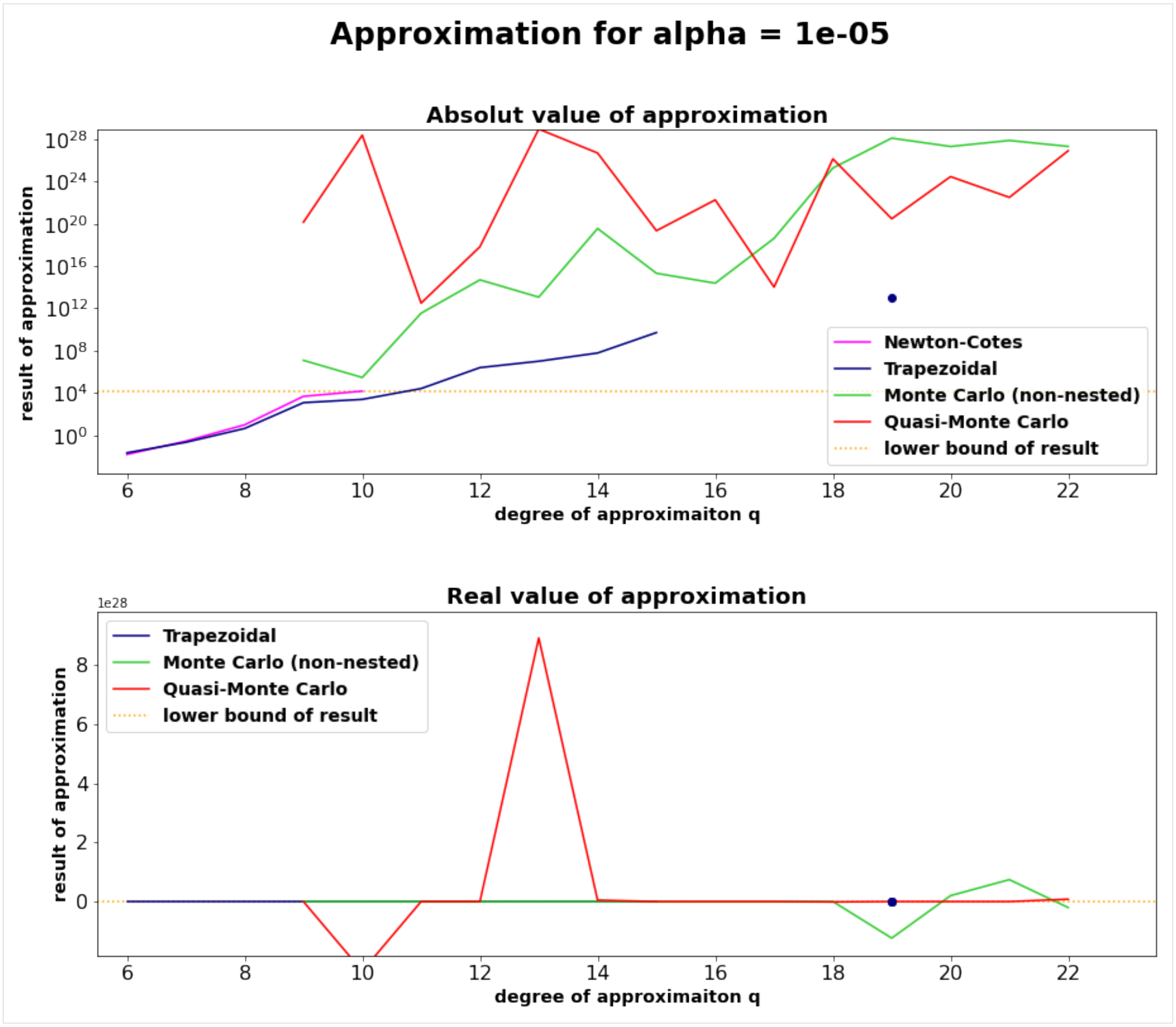

Approximation for alpha = 1
Absolut value of approximation
result of approximation
degree of approximaiton q
Newton-Cotes
Trapezoidal
Monte Carlo (non-nested)
Quasi-Monte Carlo
exact result
Real value of approximation
result of approximation
degree of approximaiton q
Trapezoidal
Monte Carlo (non-nested)
Quasi-Monte Carlo
exact result


Approximation for alpha = 0.1
Absolut value of approximation
result of approximation
degree of approximaiton q
Newton-Cotes
Trapezoidal
Monte Carlo (non-nested)
Quasi-Monte Carlo
exact result
lower bound of result
Real value of approximation
result of approximation
degree of approximaiton q
Trapezoidal
Monte Carlo (non-nested)
Quasi-Monte Carlo
exact result
lower bound of result


Approximation for alpha = 0.01
Absolut value of approximation
result of approximation
degree of approximaiton q
Newton-Cotes
Trapezoidal
Monte Carlo (non-nested)
Quasi-Monte Carlo
lower bound of result
Real value of approximation
1e8


Approximation for alpha = 0.001
Absolut value of approximation
result of approximation
degree of approximaiton q
Newton-Cotes
Trapezoidal
Monte Carlo (non-nested)
Quasi-Monte Carlo
lower bound of result
Real value of approximation
1e16


Approximation for alpha = 0.0001
Absolut value of approximation
result of approximation
degree of approximaiton q
Newton-Cotes
Trapezoidal
Monte Carlo (non-nested)
Quasi-Monte Carlo
lower bound of result
Real value of approximation
1e21
Trapezoidal
Monte Carlo (non-nested)
Quasi-Monte Carlo
lower bound of result
result of approximation
degree of approximaiton q



Often the deterministic quadratures the Smolyak algorithm seems approximate the integral for bigger α in a proper way. Never the less the error bounds are far away from being sufficiently small. In most cases these are bigger than the absolute value of the results. For alpha = 0.01 even the approximation using the trapezoidal algorithm for q = 19 is negative.

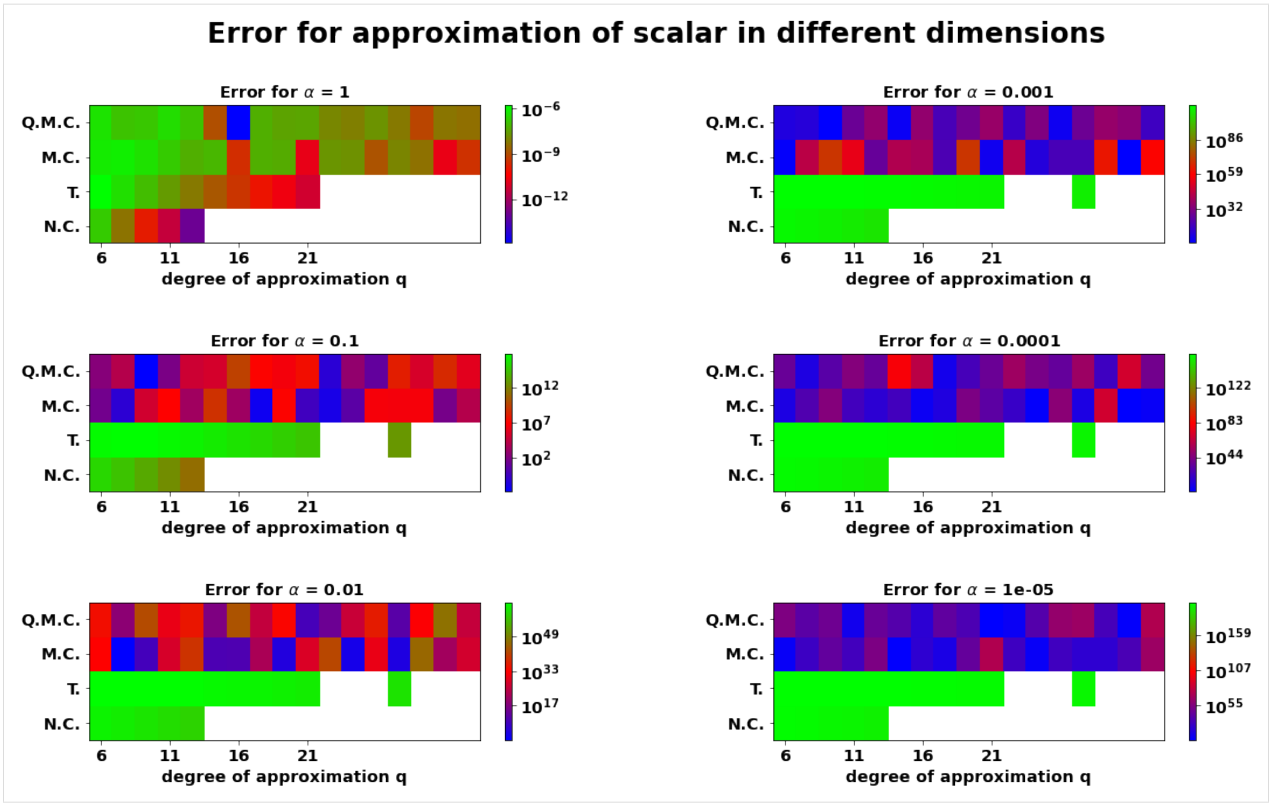
Especially the result for the non-deterministic quadratures are varying by several orders of magnitude. Aaprt from this the error estimation for the non-deterministic quadratures are also varying by several magnitudes depending on whether the estimation of the variance includes points near the maximum of the function. In this case it is not possible to say, applying these has advantages compared to especially the trapezoidal quadrature.

It should be noted, that reducing the number of points used by the one dimensional quadratures for q = 1 for this function implies that for q < 4 the result of the approximation is 0.

For the sake of completeness, we show the error estimation of the different algorithms below.

6.2 Error estimation

[27]:
```
plt.rc('font', size=14, weight="bold")
approx_simple_highdim_fct = bearcats.read_pickle("integral_over_one.pkl")

alpha = [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]
fig, ax = plt.subplots(3,2,figsize=(18,10))
plt.subplots_adjust(hspace=0.4, wspace=0.4)

for k_1 in range(len(alpha)):
    data = bearcats.read_pickle("test_higher_dim_approx_alpha_"+ str(alpha[k_1]) +".
↪pkl")
    if k_1 < 2:
        result_scipy_calc = bearcats.read_pickle("integral_Coulomb_alpha_
↪"+str(alpha[k_1])+".pkl")


    error = list()
    for k_2 in range(len(option_list)):
        error.append(data[option_list[k_2]][0][:17,1])
    error = np.vstack(error)

    if k_1  >0:
        single_approx_trap = bearcats.read_pickle("test_higher_dim_approx_one_alpha_"␣
↪+ str(alpha[k_1]) + "_only_trap.pkl").Trapezoidal[0]
        error[1,13]  =  single_approx_trap[1]

    plt.subplots_adjust(hspace=0.8, wspace=0.4)

    mat = ax[(k_1)%3, int((k_1)/3)].pcolor(error,norm=colors.LogNorm(vmin=error[error!
↪=0].min(), vmax=error.max()),
                      cmap='brg')
    fig.colorbar(mat, ax=ax[(k_1)%3, int((k_1)/3)])

    ax[(k_1)%3, int((k_1)/3)].set_yticks([0.5,1.5,2.5,3.5])
    ax[(k_1)%3, int((k_1)/3)].set_yticklabels(["N.C.","T.", "M.C.", "Q.M.C."],
↪fontsize=14, fontweight="bold")
    ax[(k_1)%3, int((k_1)/3)].set_xlabel("degree of approximation q",fontsize=14,␣
↪fontweight="bold")
    ax[(k_1)%3, int((k_1)/3)].set_xticks([0.5,3.5,6.5,9.5])
    ax[(k_1)%3, int((k_1)/3)].set_xticklabels([6,6+5,6+10,6+15],fontsize=14,␣
↪fontweight="bold")
    ax[(k_1)%3, int((k_1)/3)].set_title(r"Error for $\alpha$ = "+ str(alpha[k_1]),
↪fontsize=14, fontweight="bold")

fig.suptitle("Error for approximation of scalar in different dimensions\n",␣
↪fontsize=24,fontweight="bold")

plt.show()
```



6.3 Algorithm used for calculation of data

Below you see the algorithm, with which the integral was approximated.

```
[1]: '''
import Methodes_Studienproject.Studienprojekt_Smolyak_qmc_one_point as Studieproject_
↪one
print("remove quotes, if you want to generate data.")
alpha = [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]
dim = 6
option_list = ["Newton-Cotes",
               "Trapezoidal",
               "Monte Carlo (non-nested)",
                "Quasi-Monte Carlo"]

degree_of_approx = [5, 10, 17, 17, 17]
approx_result = np.zeros((max(degree_of_approx),len(option_list),2))

for k_3 in range( len(alpha)):
    print("")
    print("alpha = "+ str(alpha[k_3]))
    print("")
    prefactor = np.sum(np.array(list(range(1,6)))+0.5)
    function_string = str(prefactor) + " * ((z_1 - z_4) ** 4 * (z_2 - z_5) ** 4  * (z_
↪3 - z_6)**4)/(((z_1 - z_4) ** 2 + (z_2 - z_5) ** 2 + (z_3 - z_6) ** 2 + " +␣
↪str(alpha[k_3])+ ") ** (13))"
    variables_string = "(z_1 , z_2 , z_3, z_4 , z_5, z_6)"
```

(continues on next page)

(continued from previous page)

```
    for k_1  in range(len(option_list)):
        print(option_list[k_1])
        for k_2 in range(degree_of_approx[k_1]):
            approx_result[k_2,k_1,:] = Studieproject_one.controller_smolyak(function_
↪string, variables_string, option_list[k_1], k_2 + 6)[0:2]
            print(approx_result[k_2,k_1,:])
    data = {"Newton-Cotes": [approx_result[:,0,:]],"Trapezoidal": [approx_result[:,1,:
↪]], "Monte Carlo (non-nested)": [approx_result[:,2,:]],"Quasi-Monte Carlo": [approx_
↪result[:,3,:]]}
    data = bearcats.DataFrame(data=data)
    bearcats.to_pickle(data,"test_higher_dim_approx_alpha_"+ str(alpha[k_3]) +".pkl")
'''
'''
import scipy
import numpy as np
import pandas as bearcats
alpha = [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]
dim = 6

approx_result = np.zeros((max(degree_of_approx),len(option_list),2))

for k_3 in range(2,len(alpha)):

    prefactor = np.sum(np.array(list(range(1,6)))+0.5)
    function_string = str(prefactor) + " * ((z_1 - z_4) ** 4 * (z_2 - z_5) ** 4  * (z_
↪3 - z_6)**4)/(((z_1 - z_4) ** 2 + (z_2 - z_5) ** 2 + (z_3 - z_6) ** 2 + " +␣
↪str(alpha[k_3])+ ") ** (13))"
    variables_string = "(z_1 , z_2 , z_3, z_4 , z_5, z_6)"
    f = Studieproject_one.rewrite_function(function_string, variables_string)[0]
    borders = [[0,1],[0,1],[0,1],[0,1],[0,1],[0,1]]
    integral = scipy.integrate.nquad(f, borders, opts = {"epsrel": 1/alpha[k_3]**2,
↪"epsabs": 3e4/(alpha[k_3])**2, "limit": 1 })
    print(integral)
    bearcats.to_pickle(integral,"integral_Coulomb_alpha_"+str(alpha[k_3])+".pkl"
'''
print("Remove quotes to approximate function either using the Smolyak algorithm or␣
↪the scipy.integral package in python.")
```

```
Remove quotes to approximate function either using the Smolyak algorithm or the scipy.
↪integral package in python.
```

```
[ ]:
```

CHAPTER

SEVEN

VIEW

This notebook provides the possibility to use the methodes implemented by entering a function and a degree of approximation. Regarding the quadrature the option

- Newton-Cotes,
- Trapezoidal,
- Monte-Carlo (nested and not nested) and
- Quasi-Monte Carlo

exist. An other way is calling the View function.

[1]:
```
# The packages needed here are Tkinter, because the View is implemented using it
# and of course the packages of the projekt
import tkinter as tk

import os
os.chdir("..")
import Methodes_Studienproject.Studienprojekt_Smolyak_qmc_one_point as Studieproject_
↪one
```

[76]:
```
from tkinter import messagebox as mbox
# View/ Main

# In this script we make an interface with tkinter
#   The interface has: - An entry line for the function
#                      - An entry line for the variables
#                      - number of evaluations
#                      - Perhaps an entry line to verify dimension
#                      - button to start calculation
#                      - A separate popup for result showing of
#                      - Drop down menu for choosing of quadrature
#                      - Several popup for warnings
# import tkinter as tk

# own files imported
# import My_Controller as cont
# import controller_smolyak as cont

# create main viewer
root = tk.Tk()
root.title("Application for calculation of integral with Smolak algorithm")
root.geometry("800x300")
```

(continues on next page)

(continued from previous page)

```
# Label for description of functionality of viewer
description = tk.Label(root, text="To  use the application you need first need to␣
↪enter a function in the "
                                  "entry.\n"
                                  "After this the tuple of variables and in the end␣
↪the quadrature you want to be "
                                  "used.\n"
                                  "The calculation starts after clicking start.",␣
↪anchor="w", justify="left")
description.grid(row=0, column=0, sticky="w", columnspan=3)

#
# Column for entry of function
func_label = tk.Label(root, text="Please enter function: ", justify="center", anchor=
↪"w", pady=10, padx=10)
func_label.grid(row=1, column=0, sticky="w")

func_entry = tk.Entry(root, width=50, borderwidth=5)
func_entry.grid(row=1, column=1, sticky="e", padx=20, pady=20, columnspan=2)
func_entry.insert(0, "2 x y")

# Column for entry of variables
var_label = tk.Label(root, text="Please enter variables: ", justify="center", anchor=
↪"w", pady=10, padx=10)
var_label.grid(row=2, column=0, sticky="w")

var_entry = tk.Entry(root, width=50, borderwidth=5)
var_entry.grid(row=2, column=1, sticky="e", padx=20, pady=20, columnspan=2)
var_entry.insert(0, "(x , y)")

# Drop down menu for quadrature

# list of quadratures available
option_list = ["Newton-Cotes",
               "Trapezoidal",
               "Monte Carlo (nested)",
               "Monte Carlo (non-nested)",
               "Quasi-Monte Carlo"]
option = tk.StringVar()
option.set(option_list[0])
which_quad = tk.OptionMenu(root, option, *option_list)
which_quad.grid(row=3, column=2)


degree_label = tk.Label(root, text="Please choose degree of approximation: ", justify=
↪"center", anchor="w",
                        pady=10, padx=10)
degree_label.grid(row=3, column=0, sticky="w")

degree_of_approx = tk.Entry(root, width=10, borderwidth=5)
degree_of_approx.grid(row=3, column=1)
degree_of_approx.insert(0, 2)

# !! Here we also could  think about a button making it possible to define the␣
↪epsilon-error!!

```

(continues on next page)

(continued from previous page)

```
# Start button
def start():

    # read out entries
    function_string = func_entry.get()
    variables_string = var_entry.get()
    quadrature = option.get()
    degree = int(degree_of_approx.get())

    # We want to open an error box, if the degree of approximation is not big enough.␣
↪Hence, we need to
    # know the dimension of the function. Here we get this by counting the commas.
    if degree < (variables_string.count(",")+1):
        mbox.showerror("Input error", "The degree of the approximation needs to be at␣
↪least as high as the "
                                      "dimension of m the function. \n\nIn this case␣
↪this would be q = "
                                      + str(variables_string.count(",") + 1) + ".")


    else:

        # ask controller for benchmarks of approximation
        result_approx, error, cost = Studieproject.controller_smolyak(function_string,
                                                                      variables_
↪string, quadrature, degree)

        # Create new window where results are shown
        result_window = tk.Toplevel(root)
        result_window.title("Result")
        result_window.geometry("800x200")

        # Label for result of approx.
        result_label = tk.Label(result_window, text="The result of the approximation␣
↪is: " + str(result_approx),
                                justify="left", anchor="e", pady=10, padx=10)
        result_label.grid(row=0, column=0, columnspan=2)

        # Label for error
        error_label = tk.Label(result_window, text="Estimated error: " + str(error),␣
↪justify="center", anchor="w",
                               pady=10, padx=10)
        error_label.grid(row=1, column=0)

        # Label for cost estimation
        cost_label = tk.Label(result_window, text="Estimated number of evaluations: "␣
↪+ str(cost), justify="center",
                              anchor="w", pady=10, padx=10)
        cost_label.grid(row=1, column=2)

        # Quit button of result window
        quit_button_result = tk.Button(result_window, text="Quit", command=root.
↪destroy, width=10, borderwidth=5,
                                       padx=5, pady=5)
        quit_button_result.grid(row=2, column=2)
```

(continues on next page)

(continued from previous page)

```
#  Start button
start_button = tk.Button(root, text="Start", command=start, width=10, borderwidth=5,␣
↪padx=5, pady=5)

start_button.grid(row=4, column=0)

# Quit button
quit_button = tk.Button(root, text="Quit", command=root.destroy, width=10,␣
↪borderwidth=5, padx=5, pady=5)
quit_button.grid(row=4, column=2)

root.mainloop()
```

7.1 Overview over methods defined for project

Furthermore, an overview should be given over the methods defined in the project. These methods are written in alphabetic order.

[80]:
```
def write_functions():
    functions= [function for function in  dir(Studieproject_one) if
                ((not function.startswith('__') and function.find("_") > 0 and␣
↪function != "parse_expr") or function == 'smolyak')]
    for k_1 in range(len(functions)):
        print('\033[1m'+functions[k_1])
        print("")
    print('\033[0m')
write_functions()
```

```
calculate_stat_data
controller_smolyak
cost_smolyak
epsilon_cost
error_smolyak
find_all_sensible_combinations
find_next_tuples_index
find_right_position
implicit_multiplication
load_stat_data
modified_scatter_plot
monte_carlo_quad
newton_cotes
next_step
one_dim_newton_cotes
one_dim_trapezoidal
qmc_quad
rewrite_function
slice_find
smolyak
sort_tuples
standard_transformations
sum_tuple_shape
```

```
[ ]:
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search