補完と予測

数理科学研究会 BV19063　菅原 有一

令和 2 年 11 月 18 日

目次

1 研究背景 1

2 ラグランジュ補間 1

3 ラグランジュ補間の端点における補間精度を向上について 14

4 スプライン補間 47

5 XGBoost での補間 76

6 終わりに 89

1　研究背景

予言．これをうまく活用してしまえば人生イージーゲームとなり, この世の中を思うままにもできる可能性を秘めているが, 今のところ予言ができると証明された人はこの世の中にはいない．この予言というのは, 今知りえている情報から未知の, 未来の情報を手に入れるというようなことであるが, 現代社会では大量のデータを分析することによって近似的な予言を手に入れている．こういった予言のことを予測といい, 予測精度をより高くする方法について日々研究されている．ただ, 予測を凡人が理化するということは非常に複雑であり, 理解するために非常に時間と労力がかかる．なので本研究ではその基礎ともいえる 2 つの補完法, そしてデータ分析で用いられるライブラリを簡易的に用いてそれぞれの予測の特徴を考察していくことで, 予測について少しだけ勉強してみようと思う．

2　ラグランジュ補間

まずはラグランジュ補間という補完法を実装し, その特徴について考察していく．

2.1　ラグランジュ補間とは

ラグランジュ補間とは, $n+1$ 個の標本点 $(x_0, f_0), (x_1, f_1), \cdots, (x_n, f_n)$ が与えられた際に, それらすべての点を通る n 次関数の曲線を求めることで, その他の適当な点 $x(x_1 < x < x_n)$ に対応する関数値 $f(x)$ を推定することができる補間法の一つであり, 以下を満たす.

ラグランジュ補間 [1]

- 次の n 次多項式を考える．

$$\ell_j(x) := \frac{(x-x_0)(x-x_1)\cdots(x-x_{j-1})(x-x_{j+1})\cdots(x-x_n)}{(x_j-x_0)(x_j-x_1)\cdots(x_j-x_{j-1})(x_j-x_{j+1})\cdots(x_j-x_n)}$$

  - この多項式は変数 x に x_j を代入すると 1 になり, $x_k (j \neq k)$ を代入すると 0 になる
  - 黒ネッカーのデルタを用いて表すと

$$\ell_j(x_k) = \delta_{jk} \begin{cases} 1 & (j = k) \\ 0 & (j \neq k) \end{cases}$$

  - 補間条件を満たす n 次の多項式 (ラグランジュの n 次補間多項式) は次で与えられる

$$P_n(x) = f_0\ell_0(x) + f_1\ell_1(x) + \cdots + f_n\ell_n(x) = \sum_{j=0}^{n} f_j\ell_j(x)$$

2.2　補間を行うグラフについて

本研究では主に以下のグラフ $f(x) = \dfrac{1}{1+2x^2}$ について補間を行い, その特徴を考察する．グラフの概形は以下のようである (Desmos より).

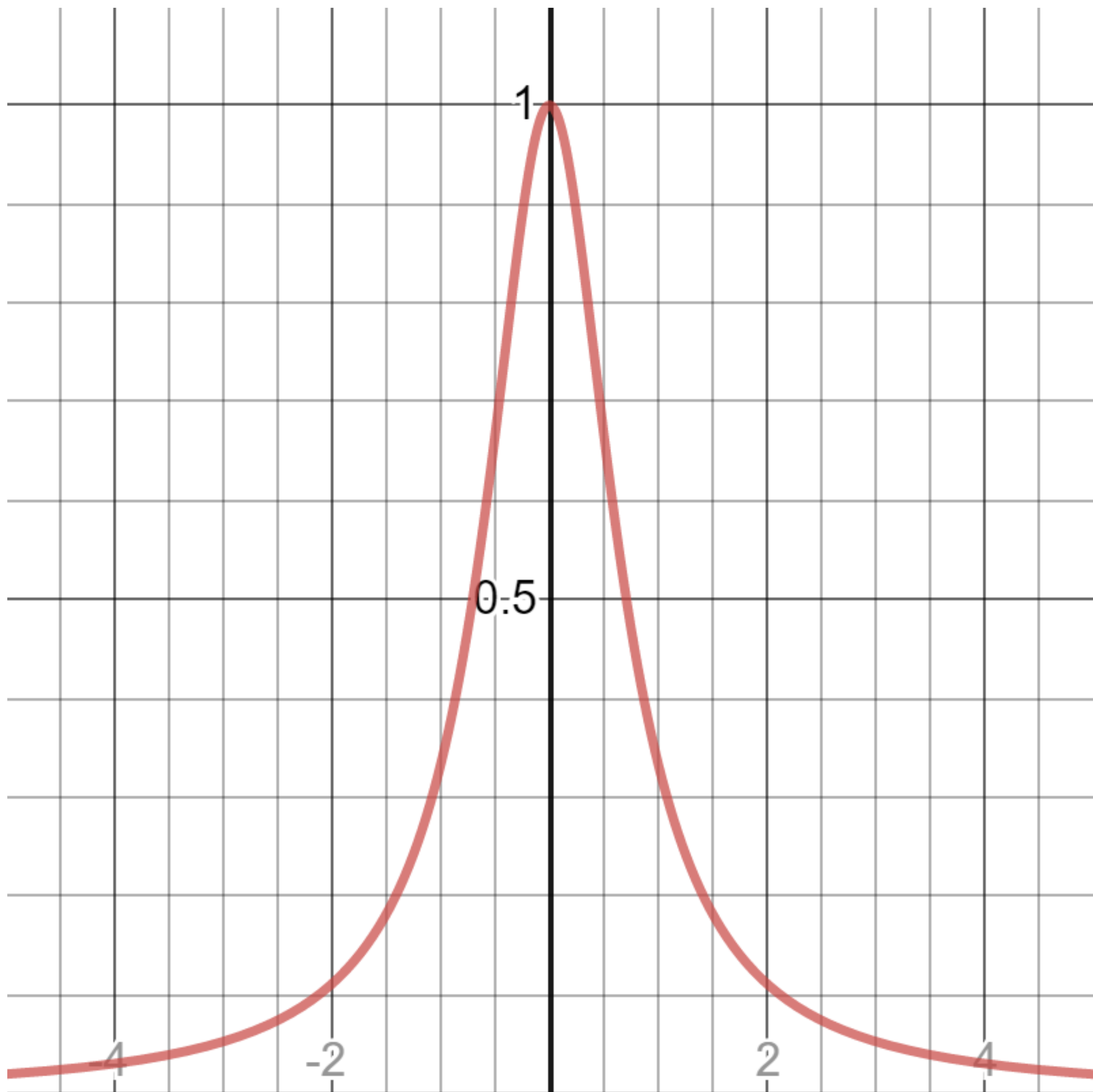

図 1　補間する関数のグラフ

2.3　ラグランジュ補間の実装

本研究では, Python にてラグランジュ補間を実装する.

2.3.1　Python の実行環境

本レポートを作成するにあたっての実行環境は以下の通りである.

- Windows 10 Home 64bit
- Python 3.5.6 :: Anaconda 4.2.0 (64-bit)
- matplotlib 1.5.3
- numpy 1.15.2
- scikit-learn 0.20.0

2.3.2　変数

プログラム作成にあたり上記のプログラムを参考に以下のようにインスタンスを作成する際の変数を作成した

- division：補間する点の分割点 (補間点 - 1 個)
- start：開始点の数
- end：終了点の数
- sample：標本点の数の値

2.3.3　プログラム

以上を用いてプログラムを次のように作成した.

ソースコード 1　Lagrange_1.py

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error

class Lagrange:
    def __init__(self, division, start, end, sample):
        self.division = division #補完する点の分割点（補完点 - 1 個）
        self.start = start #開始点
        self.end = end #終了点
        self.sample = sample #標本点

        self.length = end - start #範囲
        self.make_f_in()
        self.make_f_out()

    def make_f_in(self):
        self.x_in = []
        self.f_in = []
        for i in range(0, self.sample):
            self.x_in.append(self.start + i * self.length / (self.sample - 1))
            self.f_in.append(self.function(self.start + i * self.length / (self.
                sample - 1)))

    def make_f_out(self):
        self.x_out = []
        self.f_out = []
        self.true_f_out = []
        for i in range(0, self.division + 1):
            self.x_out.append(self.start + i * self.length / self.division)
            self.f_out.append(self.complement(self.x_out[i]))
```

```
31              self.true_f_out.append(self.function(self.x_out[i]))
32          self.split_f_out = list(np.array_split(self.f_out, 3))#3等分での精度確認
                用
33          self.split_true_f_out = list(np.array_split(self.true_f_out, 3))
34
35      def complement(self, xi):
36          Pn = 0
37          for j in range(0, self.sample):
38              L = 1
39              for i in range(0, self.sample):
40                  if i != j :
41                      L = L * (xi - self.x_in[i]) / (self.x_in[j] - self.x_in[i])
42              Pn = Pn + L * self.f_in[j]
43          return Pn
44
45      def plot(self):
46          plt.figure(figsize=(15, 10), dpi=50)
47
48          self.plot_range = np.arange(self.start, self.end, 0.1)
49          self.plot_function = self.function(self.plot_range)
50          plt.plot(self.plot_range, self.plot_function, label = u'True Function')
                # 真の関数
51
52          plt.scatter(self.x_in, self.f_in, label = u'Sample Point') # 標本点
53
54          plt.plot(self.x_out, self.f_out, marker="x", label = u'Lagrange
                Interpolation') # ラグランジュ補完
55
56          plt.legend()
57          name = str(self.sample) + "_sample.eps"
58          plt.savefig(name)
59          plt.show()
60
61
62      def value(self):
63          self.MAE_value = mean_absolute_error(self.true_f_out, self.f_out)
64          self.MSE_value = mean_squared_error(self.true_f_out, self.f_out)
65          print('{:.5e}'.format(self.MAE_value) + ', ' + '{:.5e}'.format(self.
                MSE_value))
66
67      def split_value(self, x):
68          self.MAE_split_value = mean_absolute_error(self.split_true_f_out[x], self
                .split_f_out[x])
69          self.MSE_split_value = mean_squared_error(self.split_true_f_out[x], self
                .split_f_out[x])
70          print('{:.5e}'.format(self.MAE_split_value) + ', ' + '{:.5e}'.format(
```

```
            self.MSE_split_value))
71
72      def function(self, x):
73          return 1/(1 + 2 * x * x)
74
75
76  def main():
77      M = 100 #グラフ作成用にM 分割して M+1点のラグランジュ補間を計算
78      start = -5 #範囲の開始位置
79      end = 5 #範囲の終了位置
80      sample = 8 #標本点の数 (n + 1)
81
82      L = Lagrange(M, start, end, sample)
83      L.plot()
84
85  if __name__ == '__main__':
86      main()
```

2.4　結果

プログラムを実行した結果以下のような結果が得られた.

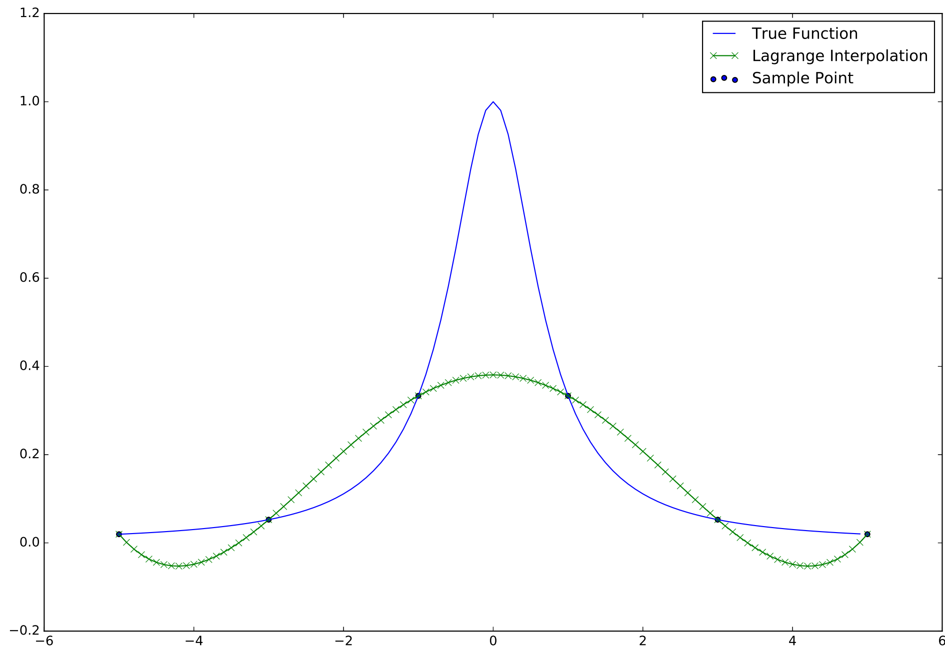

図 2　標本点の数が 6 の時のラグランジュ補間結果

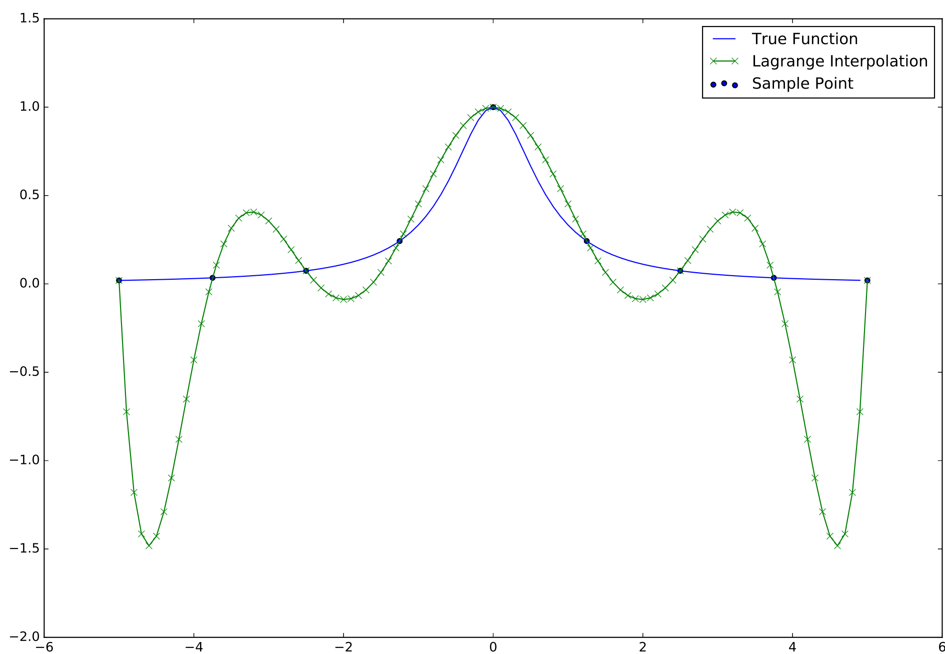

図 3　標本点の数が 9 の時のラグランジュ補間結果

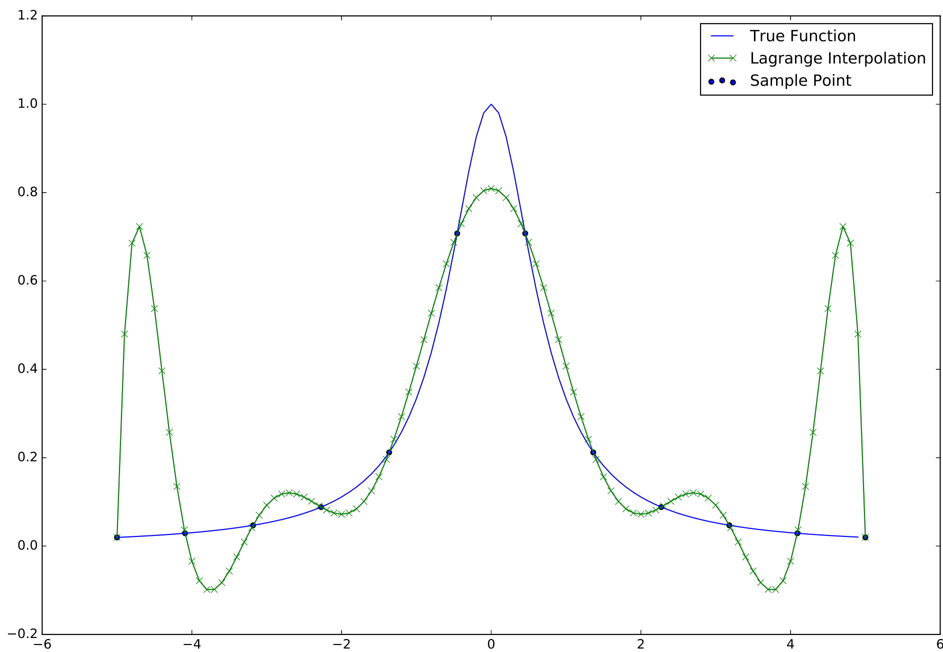

図 4　標本点の数が 12 の時のラグランジュ補間結果

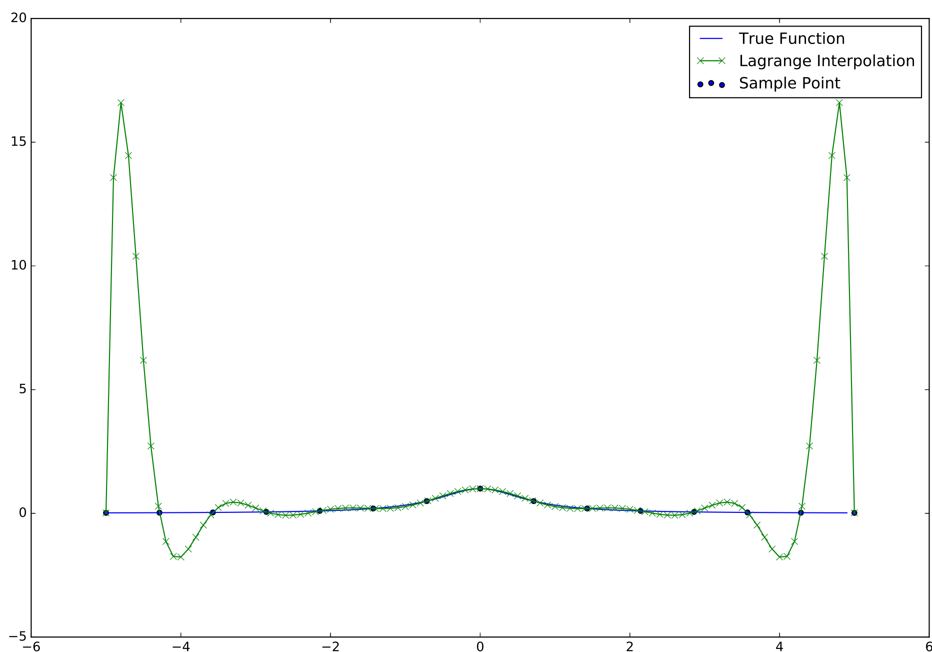

図 5　標本点の数が 15 の時のラグランジュ補間結果

結果を比較すると, 標本点の数が少ないときは中央の精度が悪く, 標本点の数が多いと端の精度が悪くなることがわかる.　また, 標本点の数が奇数であるとき, 中央に標本点が作成されるため, 近しい標本点の数の時よりは近似できているように思える.

2.5　標本点の数によるラグランジュ補間精度の評価

次に, 標本点の数によって全体的な近似精度はどのように変化しているかを定量的に考察する. 見た目の上では標本点の数が奇数であるときに近似できているように感じたが, 実際はどうなのかに関しても検証を行う. 検証を行う際には平均絶対誤差 (MAE : Mean Absolute Error) と平均二乗誤差 (MSE : Mean Squared Error）を用い, 評価を行う. また, 標本点の数はある程度の幅で計測を行うが, あまりに多すぎる標本点の数に関する検証はあまり意味を持たないのに加え, 検証に時間がかかってしまうので, 今回は標本点の数を 2 から 21 で変化させ考察を行う.

2.5.1　平均絶対誤差 (MAE : Mean Absolute Error）とは

平均絶対誤差 (MAE : Mean Absolute Error）とは実際の値と予測値の差の絶対値の平均を取ったもので, 補間を行った任意の点における実際の値との誤差が大きければ大きいほどより値が大きくなり, 近い値を取れていればいるほど値は小さくなる. 計算式は実際の値を y, 予測値 $\tilde{y}$ を, 予測数を n とすると以下のように示される.

$$MAE(y, \tilde{y}) = \frac{1}{n}\sum_{i=0}^{n-1}|y_i - \tilde{y}_i|$$

2.5.2　平均二乗誤差 (MSE : Mean Squared Error）とは

平均二乗誤差 (MSE : Mean Squared Error）とは実際の値と予測値の差の二乗の平均を取ったもので, 補間を行った任意の点における実際の値との誤差が大きければ大きいほどより値が大きくなり, 近い値を取れていればいるほど値は小さくなる. 平均絶対誤差と比較して, より大きい誤差があると値が大きくなりやすい. 計算式は実際の値を y, 予測値 $\tilde{y}$ を, 予測数を n とすると以下のように示される.

$$MSE(y, \tilde{y}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2$$

2.5.3　精度の評価結果

標本点の数を 2 から, 21 までの精度評価結果は以下のとおりである.

標本点の数	MAE	MSE	標本点の数	MAE	MSE
2	1.80858e-01	1.02370e-01	12	1.21078e-01	4.42075e-02
3	4.66201e-01	2.73577e-01	13	8.16983e-01	3.73493e+00
4	1.37384e-01	6.28741e-02	14	1.85770e-01	1.74273e-01
5	3.04600e-01	1.18302e-01	15	1.48940e+00	1.64673e+01
6	1.08593e-01	3.09867e-02	16	3.27467e-01	7.80830e-01
7	2.83654e-01	1.25943e-01	17	2.89630e+00	7.75699e+01
8	9.43682e-02	1.66282e-02	18	6.26380e-01	3.72521e+00
9	3.38612e-01	2.83058e-01	19	5.87152e+00	3.82788e+02
10	9.63167e-02	1.69963e-02	20	1.26821e+00	1.85456e+01
11	4.88646e-01	9.38274e-01	21	1.21032e+01	1.95456e+03

表 1　標本点の数による精度の評価結果

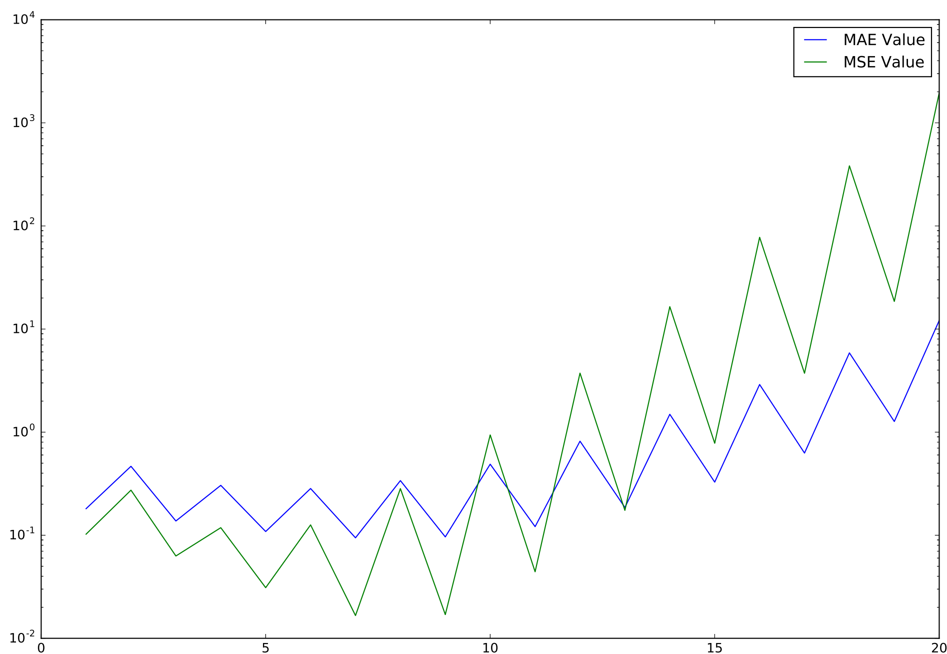

図 6　標本点の数による精度の変化

2.6　ラグランジュ補間の精度の評価結果に関する考察

以上の結果より, 標本数が偶数の方が精度がよいということがわかる. また, 標本点の数が 2 から 21 では標本点の数は 8 個の時に MAE, MSE 共に最も精度が高かった. これらの精度評価より, 全体的に外れた値を出力しないのは標本点の数が 8 の時であることがわかる.

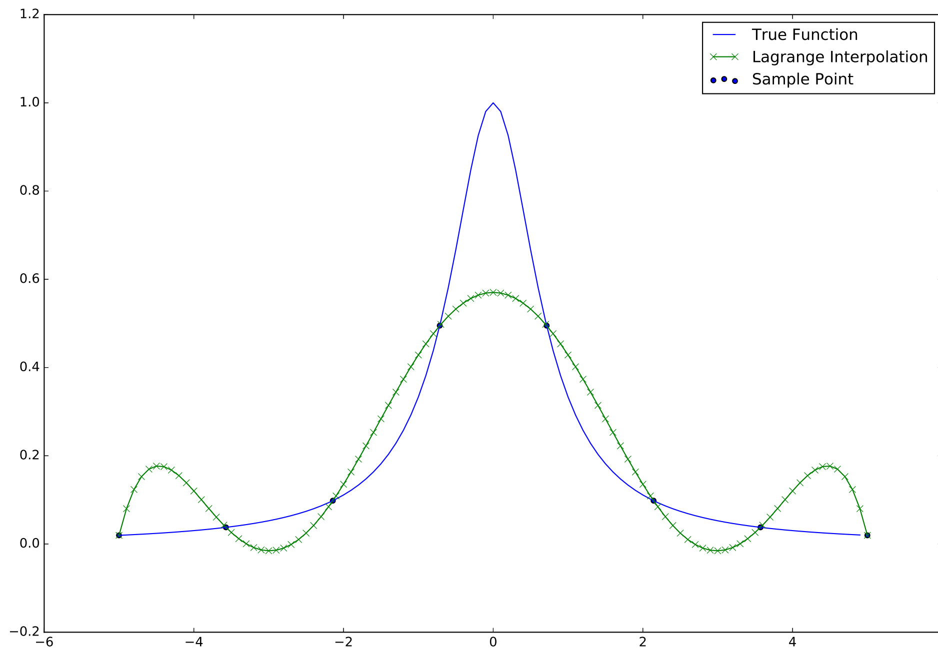

図 7　標本点の数が 8 の時のラグランジュ補間結果

2.7　ラグランジュ補間精度の区間による評価の変化

次に補間した値に関して, 区間を分割して再度同様の評価を行う. ラグランジュ補間結果の図 30 から図 33 より値が大きくなるほど中央の値の精度は上昇し, 端の精度が下がっているように見えるが, 具体的にはどの程度の差が生まれているのかについて考察を行う. 今回は区間を 3 等分し, それらの MAE, MSE の標本点の数による変化を観察する.

2.7.1　精度の評価結果

区間を x 座標の小さいほうから前, 中, 後の 3 つに分け, それぞれの精度の評価を行った結果は以下の通りとなった.

標本点の数	MAE	MSE	標本点の数	MAE	MSE
2	3.56437e-02	2.47821e-03	12	1.47437e-01	6.23313e-02
3	4.66905e-01	2.71037e-01	13	1.18828e+00	5.54577e+00
4	4.10163e-02	2.05949e-03	14	2.53347e-01	2.57351e-01
5	2.86621e-01	1.06057e-01	15	2.19630e+00	2.44580e+01
6	5.68962e-02	4.11183e-03	16	4.71441e-01	1.15911e+00
7	3.31184e-01	1.65035e-01	17	4.28984e+00	1.15214e+02
8	6.52944e-02	6.18526e-03	18	9.20656e-01	5.53275e+00
9	4.50457e-01	4.12816e-01	19	8.71300e+00	5.68552e+02
10	9.33466e-02	1.73805e-02	20	1.87683e+00	2.75455e+01
11	6.86790e-01	1.38949e+00	21	1.79719e+01	2.90310e+03

表 2　標本点の数に対しての精度の評価結果 (前)

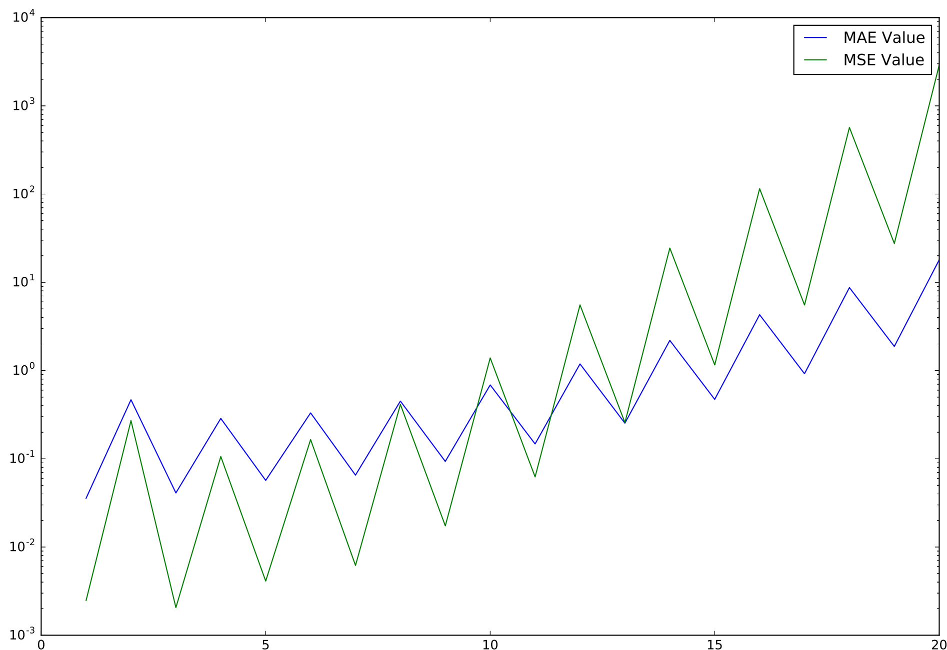

図 8　補間点の数による精度の変化 (前)

標本点の数	MAE	MSE	標本点の数	MAE	MSE
2	4.69729e-01	2.99624e-01	12	6.61681e-02	6.72342e-03
3	4.72820e-01	2.86679e-01	13	5.07445e-02	3.39175e-03
4	3.26208e-01	1.82655e-01	14	4.60249e-02	3.01675e-03
5	3.42421e-01	1.43295e-01	15	3.39195e-02	1.54335e-03
6	2.11842e-01	8.41409e-02	16	3.00140e-02	1.30198e-03
7	1.80780e-01	4.40647e-02	17	2.53272e-02	8.69569e-04
8	1.51944e-01	3.71901e-02	18	1.99684e-02	5.73062e-04
9	1.10307e-01	1.61877e-02	19	1.60879e-02	3.54900e-04
10	9.96132e-02	1.57293e-02	20	1.39853e-02	2.62292e-04
11	8.14375e-02	8.65946e-03	21	1.06621e-02	1.73231e-04

表 3　標本点の数に対しての精度の評価結果 (中)

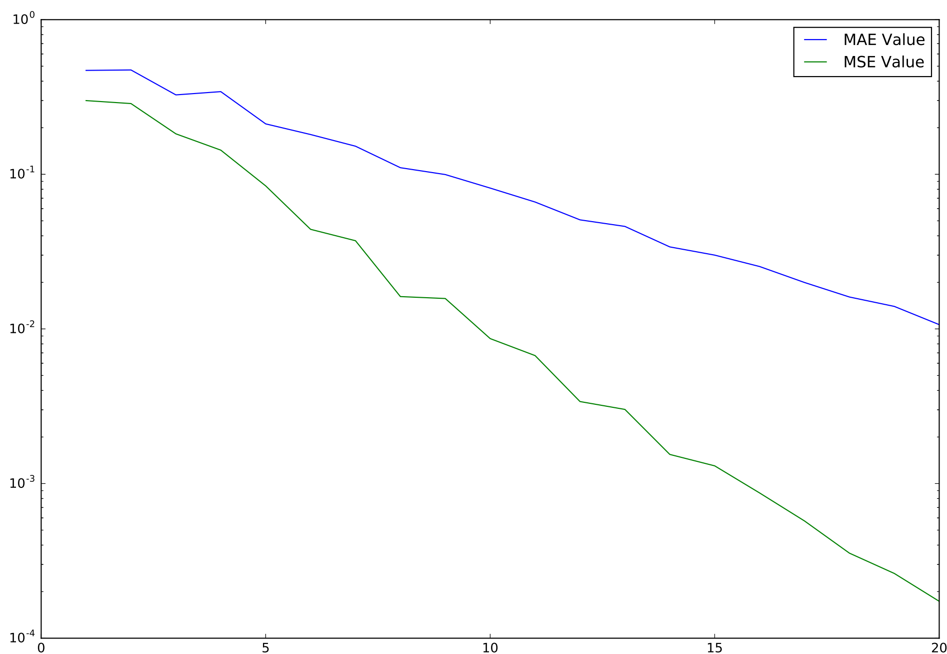

図 9　標本点の数による精度の変化 (中)

標本点の数	MAE	MSE	標本点の数	MAE	MSE
2	3.28485e-02	2.05771e-03	12	1.50493e-01	6.41543e-02
3	4.58655e-01	2.62693e-01	13	1.22389e+00	5.71382e+00
4	4.21266e-02	2.12132e-03	14	2.60126e-01	2.65123e-01
5	2.84156e-01	1.05168e-01	15	2.26067e+00	2.51990e+01
6	5.54799e-02	3.91097e-03	16	4.85596e-01	1.19424e+00
7	3.40676e-01	1.70026e-01	17	4.41850e+00	1.18705e+02
8	6.50021e-02	6.20251e-03	18	9.47975e-01	5.70040e+00
9	4.58601e-01	4.24325e-01	19	8.97679e+00	5.85781e+02
10	9.59805e-02	1.79059e-02	20	1.93337e+00	2.83802e+01
11	7.04044e-01	1.43117e+00	21	1.85156e+01	2.99107e+03

表 4　標本点の数に対しての評価結果 (後)

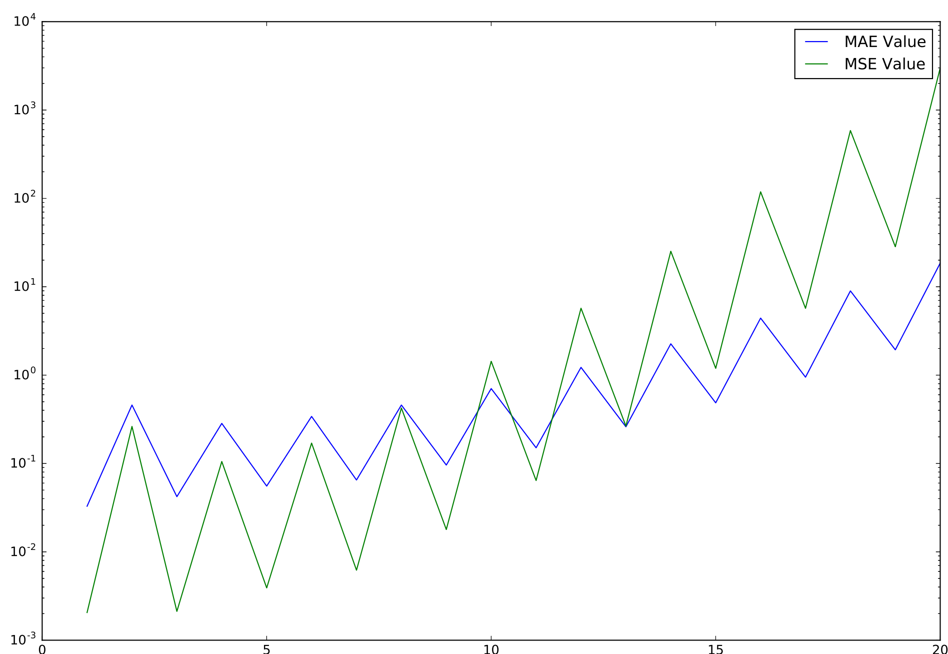

図 10　標本点の数による精度の変化 (後)

2.8　ラグランジュ補間精度の区間による評価の変化に関する考察

以上の結果より, ラグランジュ補間結果の図 30 から図 33 の見た目の通り, 端の精度は標本点の数が増えるにつれて精度が悪くなるのに対し, 中央の値に関しては標本点の数増やすと精度がよくなっていることがわかる.

2.9 ラグランジュ補間の補間精度に関する考察まとめ

全体の精度に対しての考察, 区間を分割しての考察より, 標本点が増加すればするほど中央の値は精度がよくなり, 端の精度は悪くなり, 全体としても精度が低下するということがわかった. また, それらのバランスが最も取れている点は標本点の数が 8 の時であった.

3 ラグランジュ補間の端点における補間精度を向上について

上記の考察より, 標本点の数が増えると端点の精度が下がり, 中央の精度が上がるという点から, これを解決するためにいくつかの方法を検証する. また, 標本点の数が増えると端点の精度が下がり, 中央の精度が上がるという点から全体での標本数を増やした際に精度が上昇した場合, 基本的には端の区間の精度が上昇しているといえる.

3.1 方針 1 : 標本点を区間ごとに等分割する

まずは, 標本点を複数の大きさの等しい区間に分割を行うことで精度の向上をすることができないかを検証していく.

3.1.1 変数

これを実現するためにプログラムに以下の変数を追加作成した

- division_sample : 標本点の区間の分割数

3.1.2 プログラム

以上を用いてプログラムを次のように変更した.

ソースコード 2 Lagrange_2.py

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  from sklearn.metrics import mean_absolute_error
4  from sklearn.metrics import mean_squared_error
5
6  class Lagrange:
7      def __init__(self, division, start, end, sample, division_sample):
8          self.division = division #補完する点の分割点（補完点 - 1 個）
9          self.start = start #開始点
10         self.end = end #終了点
11         self.sample = sample #標本点
12         self.division_sample = division_sample #標本点の区間の分割数
13
14         self.length = end - start #範囲
15         self.make_f_in()
16         self.make_f_out()
```

```
17
18      def make_f_in(self):
19          self.x_in = []
20          self.f_in = []
21          for i in range(0, self.sample):
22              self.x_in.append(self.start + i * self.length / (self.sample - 1))
23              self.f_in.append(self.function(self.start + i * self.length / (self.
                    sample - 1)))
24          self.split_x_in = list(np.array_split(self.x_in, self.division_sample))
25          self.split_f_in = list(np.array_split(self.f_in, self.division_sample))
26
27      def make_f_out(self):
28          self.x_out = []
29          self.f_out = []
30          self.true_f_out = []
31          for i in range(0, self.division + 1):
32              self.x_out.append(self.start + i * self.length / self.division)
33              self.true_f_out.append(self.function(self.x_out[i]))
34
35          self.split_x_out = list(np.array_split(self.x_out, self.division_sample
                ))
36
37          for i in range(0, len(self.split_x_out)):
38              for j in self.split_x_out[i]:
39                  self.f_out.append(self.complement(j, self.split_x_in[i], self.
                        split_f_in[i]))
40
41      def complement(self, xi, x, f):
42          Pn = 0
43          for j in range(0, len(x)):
44              L = 1
45              for i in range(0, len(x)):
46                  if i != j :
47                      L = L * (xi - x[i]) / (x[j] - x[i])
48              Pn = Pn + L * f[j]
49          return Pn
50
51      def plot(self):
52          plt.figure(figsize=(15, 10), dpi=50)
53
54          self.plot_range = np.arange(self.start, self.end, 0.1)
55          self.plot_function = self.function(self.plot_range)
56          plt.plot(self.plot_range, self.plot_function, label = u'True Function')
                # 真の関数
57
58          plt.scatter(self.x_in, self.f_in, label = u'Sample Point') # 標本点
```

```
59
60        plt.plot(self.x_out, self.f_out, marker="x", label = u'Lagrange
              Interpolation') # ラグランジュ補完
61
62        plt.legend()
63        plt.savefig("Split_Lagrange_20_2.eps")
64        plt.show()
65
66    def value(self):
67        self.MAE_value = mean_absolute_error(self.true_f_out, self.f_out)
68        self.MSE_value = mean_squared_error(self.true_f_out, self.f_out)
69        print('{:.5e}'.format(self.MAE_value) + ', ' + '{:.5e}'.format(self.
              MSE_value))
70
71    def function(self, x):
72        return 1/(1 + 2 * x * x)
73
74
75 def main():
76    M = 100 #グラフ作成用にM分割してM+1点のラグランジュ補間を計算
77    start = -5 #範囲の開始位置
78    end = 5 #範囲の終了位置
79    sample = 20 #標本点の数
80    division_sample = 2 #区間の分割数
81
82    L = Lagrange(M, start, end, sample, division_sample)
83    L.plot()
84    L.value()
85
86 if __name__ == '__main__':
87    main()
```

3.1.3 結果

標本点の数は 8, 分割数は 2 から 8 にし, プログラムを実行した結果以下のような結果が得られた.

分割数	MAE	MSE
2	9.43682e-02	1.66282e-02
3	2.64064e-02	1.89476e-03
4	6.47467e-02	1.54762e-02
5	5.06820e-02	8.32233e-03
6	7.69664e-02	1.69908e-02
7	1.30041e-01	5.24360e-02
8	9.79918e-02	2.84968e-02
9	7.40898e-02	2.01191e-02

表 5　標本点の数が 8 の時の分割数に対しての精度の評価結果

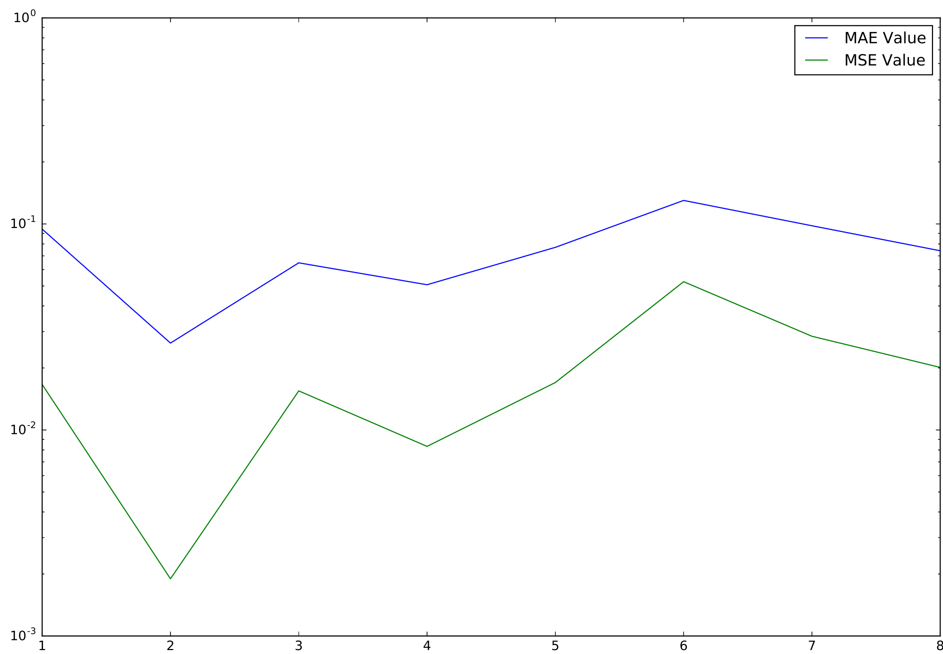

図 11　標本点の数が 8 の時の分割数による精度の評価結果

3.1.4　考察

結果より, 分割数が 2 の際に最も精度がよくなった. またその時のグラフは以下のとおりである.

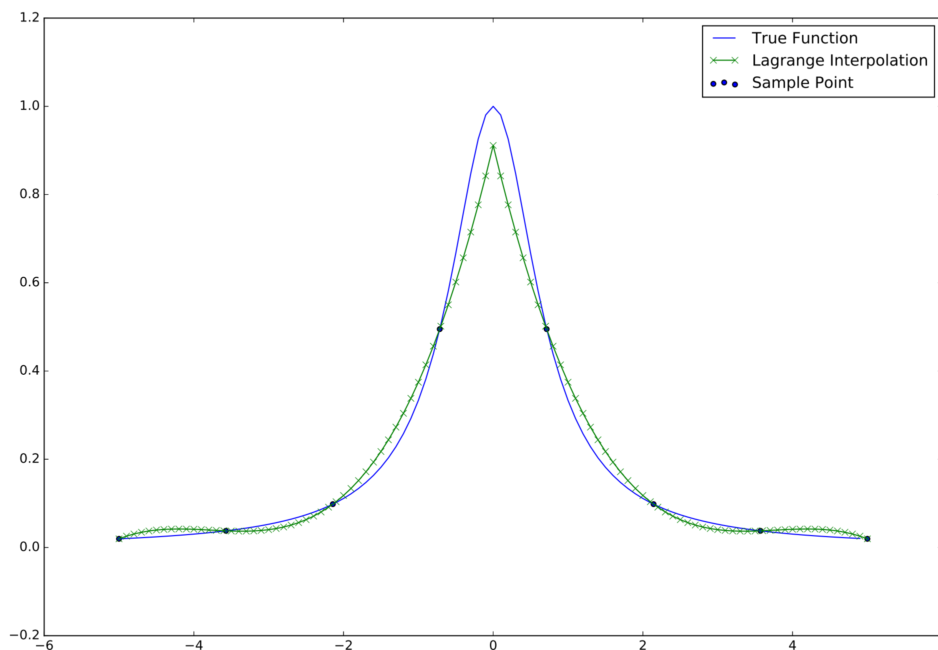

図 12　標本点の数が 8, 分割数 2 の時のラグランジュ補間結果

この図より, 中央が尖っているという問題はあるがその他は非常によく近似できていることがわかる. なので, 分割数を 2 の状態で n の値を変化させて検証を行う.

標本点の数	MAE	MSE	標本点の数	MAE	MSE
2	1.80858e-01	1.02370e-01	12	9.41072e-03	1.44968e-03
3	2.37818e-01	1.07557e-01	13	7.83809e-03	4.12784e-04
4	1.22754e-01	5.50089e-02	14	9.56228e-03	1.76386e-03
5	1.04942e-01	3.78131e-02	15	7.45373e-03	7.09506e-04
6	6.66361e-02	1.59004e-02	16	8.22949e-03	1.29642e-03
7	5.30444e-02	1.72095e-02	17	6.74163e-03	9.11080e-04
8	2.64064e-02	1.89476e-03	18	5.79036e-03	6.54835e-04
9	2.98629e-02	6.18656e-03	19	5.16176e-03	7.47093e-04
10	9.51565e-03	4.12414e-04	20	3.38061e-03	2.12646e-04
11	1.53645e-02	1.39990e-03	21	3.53864e-03	4.28648e-04

表 6　分割数 2 の時の標本点の数に対しての精度評価

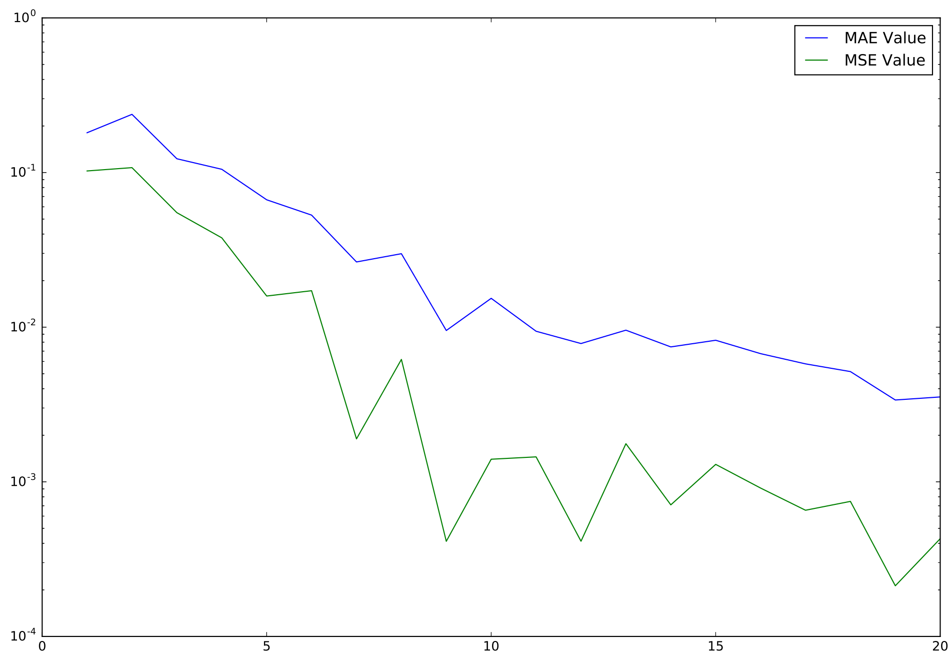

図 13　分割数 2 の時の標本点の数の変化による精度評価の変化

グラフよりおおよそ標本点の数が多くなると精度がよくなることがわかった．最も精度の良かった標本点の数が 20 の時のグラフは以下である．

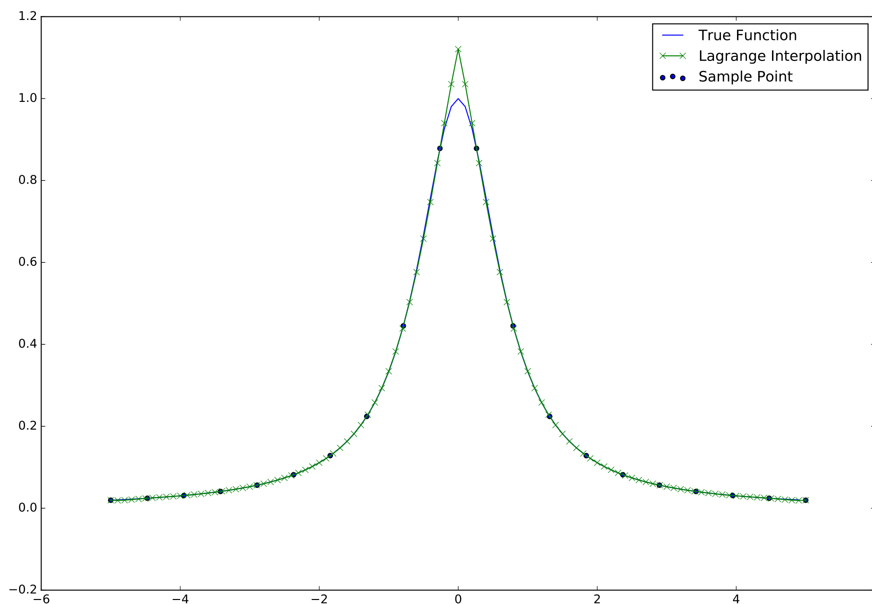

図 14　標本点の数が 20, 分割数 2 の時のラグランジュ補間結果

この図は端点の精度は非常に良くなったと思われる．

3.2　方針 2：標本点をコピーする

精度が上がるかは全く不明だが, 全部の点をそのままコピーしてみる.

3.2.1　変数

これを実現するためにプログラムに以下の変数を追加作成した

- copy：同じ点の標本点の数

3.2.2　プログラム

以上を用いてプログラムを次のように変更した.

ソースコード 3　Lagrange_3.py

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 from sklearn.metrics import mean_absolute_error
4 from sklearn.metrics import mean_squared_error
5
6 class Lagrange:
7     def __init__(self, division, start, end, sample, copy):
8         self.division = division #補完する点の分割点（補完点 - 1 個）
9         self.start = start #開始点
10        self.end = end #終了点
11        self.sample = sample #標本点
12        self.copy = copy
13
14        self.length = end - start #範囲
15        self.make_f_in()
16        self.make_f_out()
17
18    def make_f_in(self):
19        self.x_in = []
20        self.f_in = []
21        for i in range(0, self.sample):
22            for _ in range(0, self.copy):
23                self.x_in.append(self.start + i * self.length / (self.sample -
                       1))
24                self.f_in.append(self.function(self.start + i * self.length / (
                       self.sample - 1)))
25
26    def make_f_out(self):
27        self.x_out = []
28        self.f_out = []
29        self.true_f_out = []
```

```
30          for i in range(0, self.division + 1):
31              self.x_out.append(self.start + i * self.length / self.division)
32              self.f_out.append(self.complement(self.x_out[i]))
33              self.true_f_out.append(self.function(self.x_out[i]))
34          self.split_f_out = list(np.array_split(self.f_out, 3))#3等分での精度確認
                用
35          self.split_true_f_out = list(np.array_split(self.true_f_out, 3))
36
37      def complement(self, xi):
38          Pn = 0
39          for j in range(0, self.sample * self.copy):
40              L = 1
41              for i in range(0, self.sample * self.copy):
42                  if self.x_in[i] != self.x_in[j] :
43                      L = L * (xi - self.x_in[i]) / (self.x_in[j] - self.x_in[i])
44              Pn = Pn + L * self.f_in[j]
45          return Pn
46
47      def plot(self):
48          plt.figure(figsize=(15, 10), dpi=50)
49
50          self.plot_range = np.arange(self.start, self.end, 0.1)
51          self.plot_function = self.function(self.plot_range)
52          plt.plot(self.plot_range, self.plot_function, label = u'True Function')
                # 真の関数
53
54          plt.scatter(self.x_in, self.f_in, label = u'Sample Point') # 標本点
55
56          plt.plot(self.x_out, self.f_out, marker="x", label = u'Lagrange
                Interpolation') # ラグランジュ補完
57
58          plt.legend()
59          name = str(self.sample) + "_sample_3.eps"
60          plt.savefig(name)
61          plt.show()
62
63
64      def value(self):
65          self.MAE_value = mean_absolute_error(self.true_f_out, self.f_out)
66          self.MSE_value = mean_squared_error(self.true_f_out, self.f_out)
67          print('{:.5e}'.format(self.MAE_value) + ', ' + '{:.5e}'.format(self.
                MSE_value))
68
69      def split_value(self, x):
70          self.MAE_split_value = mean_absolute_error(self.split_true_f_out[x], self
                .split_f_out[x])
```

```
71          self.MSE_split_value = mean_squared_error(self.split_true_f_out[x], self
                .split_f_out[x])
72          print('{:.5e}'.format(self.MAE_split_value) + ', ' + '{:.5e}'.format(
                self.MSE_split_value))
73
74      def function(self, x):
75          return 1/(1 + 2 * x * x)
76
77
78  def main():
79      M = 100 #グラフ作成用にM分割してM+1点のラグランジュ補間を計算
80      start = -5 #範囲の開始位置
81      end = 5 #範囲の終了位置
82      sample = 8 #標本点の数 (n + 1)
83      copy = 2 #同じ点の標本点の数
84
85      L = Lagrange(M, start, end, sample, copy)
86      L.plot()
87
88  if __name__ == '__main__':
89      main()
```

3.2.3　結果

試しに標本点の数 8, 同じ点の標本点を 2 個にしてみた．結果は以下のようなグラフになった．

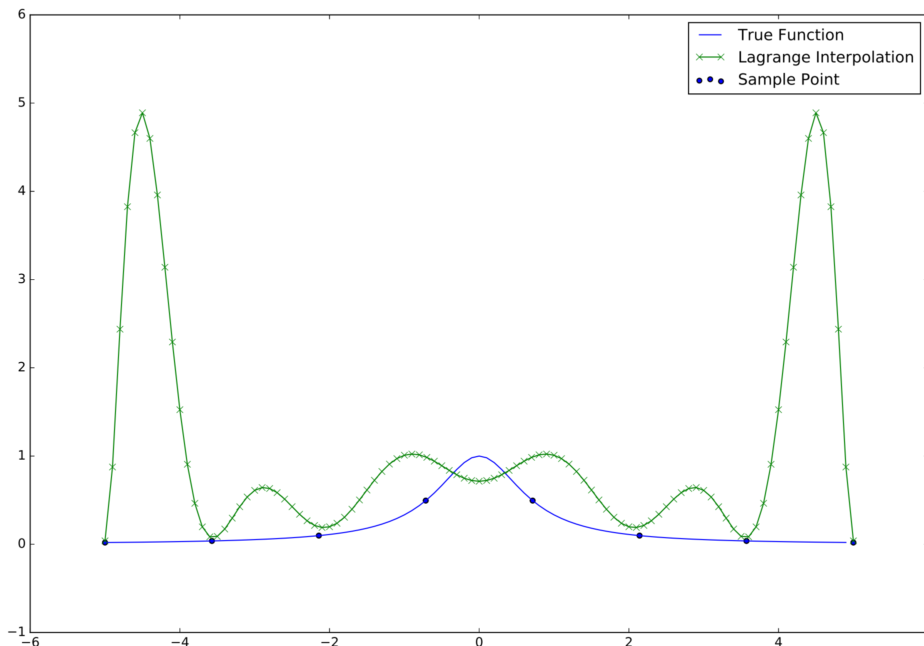

図 15　標本点の数が 8, 標本点 2 倍の時のラグランジュ補間結果

明らかに変である．標本点の上を通っていない．これはラグランジュ補間の式を考えることで明らかであるが，ラグランジュ補間の式は上述したように以下の式で表される．

$$\ell_j(x) := \frac{(x-x_0)(x-x_1)\cdots(x-x_{j-1})(x-x_{j+1})\cdots(x-x_n)}{(x_j-x_0)(x_j-x_1)\cdots(x_j-x_{j-1})(x_j-x_{j+1})\cdots(x_j-x_n)}$$

$$P_n(x) = f_0\ell_0(x) + f_1\ell_1(x) + \cdots + f_n\ell_n(x) = \sum_{j=0}^{n} f_j\ell_j(x)$$

単純に点を 2 倍にした場合，

$$\ell_j(x) := \frac{(x-x_0)^2(x-x_1)^2\cdots(x-x_{j-1})^2(x-x_{j+1})^2\cdots(x-x_n)^2}{(x_j-x_0)^2(x_j-x_1)^2\cdots(x_j-x_{j-1})^2(x_j-x_{j+1})^2\cdots(x_j-x_n)^2}$$

$$P_n(x) = 2f_0\ell_0(x) + 2f_1\ell_1(x) + \cdots + 2f_n\ell_n(x) = \sum_{j=0}^{n} 2f_j\ell_j(x)$$

となり，点が 2 倍の時に $P_n(x)$ は 2 倍，同様に n 倍になると $P_n(x)$ は n 倍になってしまう．従って標本点を複製する場合は $P_n(x)$ を複製した数で割らなくてはならない．これらを修正したものが以下のプログラムとなる．

ソースコード 4 Lagrange_4.py

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error

class Lagrange:
    def __init__(self, division, start, end, sample, copy):
        self.division = division #補完する点の分割点（補完点 - 1 個）
        self.start = start #開始点
        self.end = end #終了点
        self.sample = sample #標本点
        self.copy = copy

        self.length = end - start #範囲
        self.make_f_in()
        self.make_f_out()

    def make_f_in(self):
        self.x_in = []
        self.f_in = []
        for i in range(0, self.sample):
            for _ in range(0, self.copy):
                self.x_in.append(self.start + i * self.length / (self.sample -
                    1))
                self.f_in.append(self.function(self.start + i * self.length / (
                    self.sample - 1)))

    def make_f_out(self):
        self.x_out = []
        self.f_out = []
        self.true_f_out = []
        for i in range(0, self.division + 1):
            self.x_out.append(self.start + i * self.length / self.division)
            self.f_out.append(self.complement(self.x_out[i]))
            self.true_f_out.append(self.function(self.x_out[i]))
        self.split_f_out = list(np.array_split(self.f_out, 3))#3等分での精度確認
            用
        self.split_true_f_out = list(np.array_split(self.true_f_out, 3))

    def complement(self, xi):
        Pn = 0
        for j in range(0, self.sample * self.copy):
            L = 1
            for i in range(0, self.sample * self.copy):
                if self.x_in[i] != self.x_in[j] :
```

```
43                 L = L * (xi - self.x_in[i]) / (self.x_in[j] - self.x_in[i])
44             Pn = Pn + L * self.f_in[j]
45         return Pn/self.copy
46
47     def plot(self):
48         plt.figure(figsize=(15, 10), dpi=50)
49
50         self.plot_range = np.arange(self.start, self.end, 0.1)
51         self.plot_function = self.function(self.plot_range)
52         plt.plot(self.plot_range, self.plot_function, label = u'True Function')
               # 真の関数
53
54         plt.scatter(self.x_in, self.f_in, label = u'Sample Point') # 標本点
55
56         plt.plot(self.x_out, self.f_out, marker="x", label = u'Lagrange
               Interpolation') # ラグランジュ補完
57
58         plt.legend()
59         name = str(self.sample) + "_sample_4.eps"
60         plt.savefig(name)
61         plt.show()
62
63
64     def value(self):
65         self.MAE_value = mean_absolute_error(self.true_f_out, self.f_out)
66         self.MSE_value = mean_squared_error(self.true_f_out, self.f_out)
67         print('{:.5e}'.format(self.MAE_value) + ', ' + '{:.5e}'.format(self.
               MSE_value))
68
69     def split_value(self, x):
70         self.MAE_split_value = mean_absolute_error(self.split_true_f_out[x], self
               .split_f_out[x])
71         self.MSE_split_value = mean_squared_error(self.split_true_f_out[x], self
               .split_f_out[x])
72         print('{:.5e}'.format(self.MAE_split_value) + ', ' + '{:.5e}'.format(
               self.MSE_split_value))
73
74     def function(self, x):
75         return 1/(1 + 2 * x * x)
76
77
78 def main():
79     M = 100 #グラフ作成用にM分割してM+1点のラグランジュ補間を計算
80     start = -5 #範囲の開始位置
81     end = 5 #範囲の終了位置
82     sample = 8 #標本点の数 (n + 1)
```

```
83     copy = 2 #同じ点の標本点の数
84
85     L = Lagrange(M, start, end, sample, copy)
86     L.plot()
87
88 if __name__ == '__main__':
89     main()
```

3.2.4 修正後の結果

修正したプログラムを実行した結果以下のようになった.

標本点の数	MAE	MSE
2	9.43682e-02	1.66282e-02
3	4.47523e-01	6.26662e-01
4	2.19219e-01	1.40834e-01
5	6.76413e-01	2.34660e+00
6	4.96413e-01	1.19039e+00
7	1.29848e+00	1.17087e+01
8	1.17087e+00	9.79614e+00
9	2.79549e+00	6.76800e+01
10	2.87890e+00	7.57551e+01
11	6.48184e+00	4.25261e+02

表 7 標本点の数が 8 の時の同じ点の個数に対しての精度評価

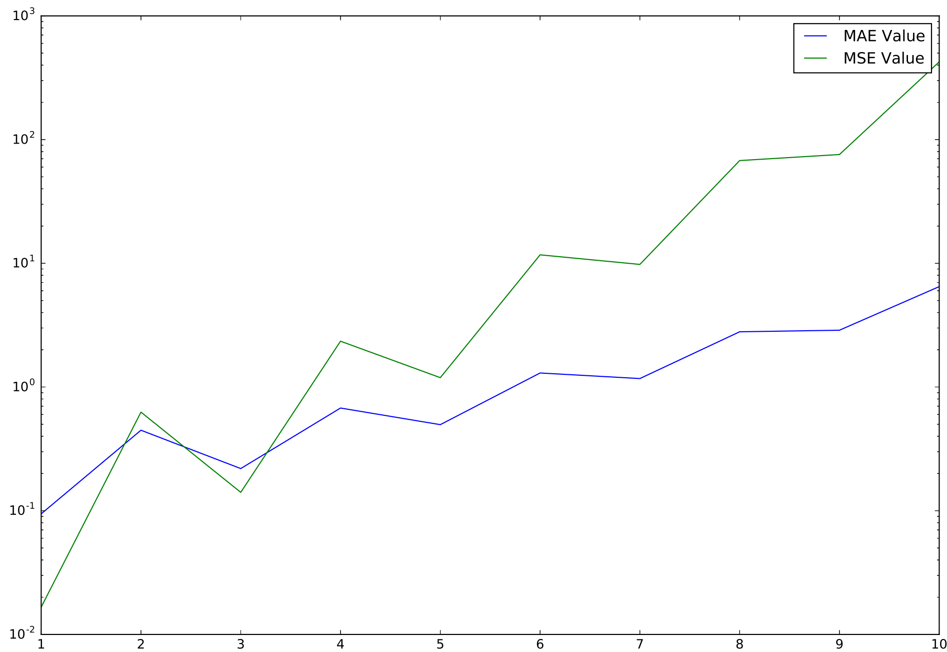

図 16　標本点の数が 8, 同じ点の個数の変化と精度評価

3.2.5　考察

結果より，コピーをすることで精度が悪化することがわかった．標本点を簡易的に増やすことに意味がないということがわかった．

3.3　方針 3：端の区間にある標本点の数を増やす

理想としては中央に行くにつれて点の感覚を大きくすることであるが，実装が大変になってしまうので，今回は区間を 3 つに分け，端の区間を中の区間の定数倍の標本数にすることで精度向上できないか検証する．

3.3.1　区間の決定

3 つに分ける範囲に関しては山のようになっている部分とそうでない部分に分ける為に，まずは変曲点を境に行う．

$$f(x) = \frac{1}{1+2x^2}$$

$$f'(x) = -\frac{4x}{(2x^2+1)^2}$$

$$f''(x) = \frac{4\left(6x^2-1\right)}{(2x^2+1)^3}$$

であるから, $f''(x) = 0$ の時, $x = \pm\frac{1}{\sqrt{6}}$ となる. 即ち変曲点は $x = \pm\frac{1}{\sqrt{6}}$ なので, それを境に 3 区間に分ける.

3.3.2 プログラム

これらを反映したプログラムは以下のようになる. このプログラムはこの関数以外での利用を想定していないので, 変数の変更などについては省略する.

ソースコード 5 Lagrange_5.py

```
1  import math
2  import matplotlib.pyplot as plt
3  import numpy as np
4  from sklearn.metrics import mean_absolute_error
5  from sklearn.metrics import mean_squared_error
6
7  class Lagrange:
8      def __init__(self, division, sample_1, sample_2, sample_3):
9          self.division = division #補完する点の分割点(補完点 - 1 個)
10         self.start = -5 #開始点
11         self.end = 5 #終了点
12         self.length = 10
13
14         self.sample_1 = sample_1
15         self.sample_2 = sample_2
16         self.sample_3 = sample_3 #標本点
17         self.sample = sample_1 + sample_2 + sample_3
18
19         self.length_1 = -1 / math.sqrt(6) + 5
20         self.length_2 = 2 / math.sqrt(6)
21         self.length_3 = -1 / math.sqrt(6) + 5 #範囲
22
23         self.make_f_in()
24         self.make_f_out()
25
26     def make_f_in(self):
27         self.x_in = []
28         self.f_in = []
29         for i in range(0, self.sample_1):
30             self.x_in.append(self.start + i * self.length_1 / (self.sample_1 -
                   1))
31             self.f_in.append(self.function(self.start + i * self.length_1 / (
                   self.sample_1 - 1)))
32
33         for i in range(0, self.sample_2):
34             self.x_in.append(self.start + self.length_1 + (i + 1) * self.
                   length_2 / (self.sample_2 + 1))
35             self.f_in.append(self.function(self.start + self.length_1 + (i + 1)
```

```
                * self.length_2 / (self.sample_2 + 1)))
36
37          for i in range(0, self.sample_3):
38              self.x_in.append(self.end - self.length_3 + i * self.length_3 / (
                    self.sample_3 - 1))
39              self.f_in.append(self.function(self.end - self.length_3 + i * self.
                    length_3 / (self.sample_3 - 1)))
40
41      def make_f_out(self):
42          self.x_out = []
43          self.f_out = []
44          self.true_f_out = []
45          for i in range(0, self.division + 1):
46              self.x_out.append(self.start + i * self.length / self.division)
47              self.f_out.append(self.complement(self.x_out[i]))
48              self.true_f_out.append(self.function(self.x_out[i]))
49          self.split_f_out = list(np.array_split(self.f_out, 3))#3等分での精度確認
              用
50          self.split_true_f_out = list(np.array_split(self.true_f_out, 3))
51
52      def complement(self, xi):
53          Pn = 0
54          for j in range(0, self.sample):
55              L = 1
56              for i in range(0, self.sample):
57                  if i != j :
58                      L = L * (xi - self.x_in[i]) / (self.x_in[j] - self.x_in[i])
59              Pn = Pn + L * self.f_in[j]
60          return Pn
61
62      def plot(self):
63          plt.figure(figsize=(15, 10), dpi=50)
64
65          self.plot_range = np.arange(self.start, self.end, 0.1)
66          self.plot_function = self.function(self.plot_range)
67          plt.plot(self.plot_range, self.plot_function, label = u'True Function')
              # 真の関数
68
69          plt.scatter(self.x_in, self.f_in, label = u'Sample Point') # 標本点
70
71          plt.plot(self.x_out, self.f_out, marker="x", label = u'Lagrange
              Interpolation') # ラグランジュ補完
72
73          plt.legend()
74          name = str(self.sample) + "_sample_5.eps"
75          plt.savefig(name)
```

```
76          plt.show()
77
78
79      def value(self):
80          self.MAE_value = mean_absolute_error(self.true_f_out, self.f_out)
81          self.MSE_value = mean_squared_error(self.true_f_out, self.f_out)
82          print('{:.5e}'.format(self.MAE_value) + ', ' + '{:.5e}'.format(self.
                MSE_value))
83
84      def split_value(self, x):
85          self.MAE_split_value = mean_absolute_error(self.split_true_f_out[x], self
                .split_f_out[x])
86          self.MSE_split_value = mean_squared_error(self.split_true_f_out[x], self
                .split_f_out[x])
87          print('{:.5e}'.format(self.MAE_split_value) + ', ' + '{:.5e}'.format(
                self.MSE_split_value))
88
89      def function(self, x):
90          return 1/(1 + 2 * x * x)
91
92
93  def main():
94      M = 100 #グラフ作成用にM分割してM+1点のラグランジュ補間を計算
95      sample_1 = 30 #区間1の標本点の数
96      sample_2 = 1 #区間2の標本点の数
97      sample_3 = 30 #区間3の標本点の数
98
99      L = Lagrange(M, sample_1, sample_2, sample_3)
100     L.plot()
101
102 if __name__ == '__main__':
103     main()
```

3.3.3　結果

比較的極端な数字で実行した結果は以下のとおりである.

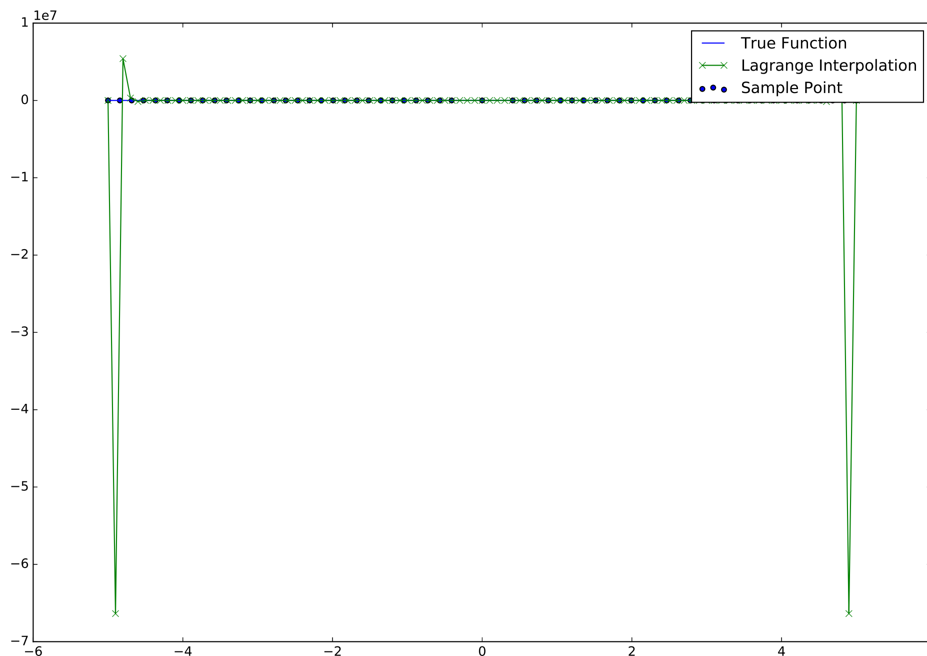

図 17　左の区間から順に 30, 1, 30 の標本数にした際のラグランジュ補完結果

3.3.4　考察

中央の区間が端の区間よりも小さく, 標本点がほとんど均等に配置されてしまい, 非常に端点の誤差が大きくなってしまった. なので区切りを行う点を変更する必要があることがわかる.

3.3.5　区間の変更

次は区間を 3 等分することとする. プログラムは以下のようになる.

ソースコード 6　Lagrange_6.py

```
1  import math
2  import matplotlib.pyplot as plt
3  import numpy as np
4  from sklearn.metrics import mean_absolute_error
5  from sklearn.metrics import mean_squared_error
6
7  class Lagrange:
8      def __init__(self, division, sample_1, sample_2, sample_3):
9          self.division = division #補完する点の分割点 (補完点 - 1 個)
10         self.start = -5 #開始点
```

```
11         self.end = 5 #終了点
12         self.length = 10
13
14         self.sample_1 = sample_1
15         self.sample_2 = sample_2
16         self.sample_3 = sample_3 #標本点
17         self.sample = sample_1 + sample_2 + sample_3
18
19         self.length_1 = 10/3
20         self.length_2 = 10/3
21         self.length_3 = 10/3 #範囲
22
23         self.make_f_in()
24         self.make_f_out()
25
26     def make_f_in(self):
27         self.x_in = []
28         self.f_in = []
29         for i in range(0, self.sample_1):
30             self.x_in.append(self.start + i * self.length_1 / (self.sample_1 -
                   1))
31             self.f_in.append(self.function(self.start + i * self.length_1 / (
                   self.sample_1 - 1)))
32
33         for i in range(0, self.sample_2):
34             self.x_in.append(self.start + self.length_1 + (i + 1) * self.
                   length_2 / (self.sample_2 + 1))
35             self.f_in.append(self.function(self.start + self.length_1 + (i + 1)
                    * self.length_2 / (self.sample_2 + 1)))
36
37         for i in range(0, self.sample_3):
38             self.x_in.append(self.end - self.length_3 + i * self.length_3 / (
                   self.sample_3 - 1))
39             self.f_in.append(self.function(self.end - self.length_3 + i * self.
                   length_3 / (self.sample_3 - 1)))
40
41     def make_f_out(self):
42         self.x_out = []
43         self.f_out = []
44         self.true_f_out = []
45         for i in range(0, self.division + 1):
46             self.x_out.append(self.start + i * self.length / self.division)
47             self.f_out.append(self.complement(self.x_out[i]))
48             self.true_f_out.append(self.function(self.x_out[i]))
49         #self.split_f_out = list(np.array_split(self.f_out, 3))#3等分での精度確
               認用
```

```
50          #self.split_true_f_out = list(np.array_split(self.true_f_out, 3))
51
52      def complement(self, xi):
53          Pn = 0
54          for j in range(0, self.sample):
55              L = 1
56              for i in range(0, self.sample):
57                  if i != j :
58                      L = L * (xi - self.x_in[i]) / (self.x_in[j] - self.x_in[i])
59              Pn = Pn + L * self.f_in[j]
60          return Pn
61
62      def plot(self):
63          plt.figure(figsize=(15, 10), dpi=50)
64
65          self.plot_range = np.arange(self.start, self.end, 0.1)
66          self.plot_function = self.function(self.plot_range)
67          plt.plot(self.plot_range, self.plot_function, label = u'True Function')
                # 真の関数
68
69          plt.scatter(self.x_in, self.f_in, label = u'Sample Point') # 標本点
70
71          plt.plot(self.x_out, self.f_out, marker="x", label = u'Lagrange
                Interpolation') # ラグランジュ補完
72
73          plt.legend()
74          #name = str(self.sample) + "_sample_6.eps"
75          #plt.savefig(name)
76
77          name = "Best_Lagrange_3.eps"
78          plt.savefig(name)
79          plt.show()
80
81
82      def value(self):
83          self.MAE_value = mean_absolute_error(self.true_f_out, self.f_out)
84          self.MSE_value = mean_squared_error(self.true_f_out, self.f_out)
85          #print('{:.5e}'.format(self.MAE_value) + ', ' + '{:.5e}'.format(self.
                MSE_value))
86
87      #def split_value(self, x):
88      # self.MAE_split_value = mean_absolute_error(self.split_true_f_out[x], self.
            split_f_out[x])
89      # self.MSE_split_value = mean_squared_error(self.split_true_f_out[x], self.
            split_f_out[x])
90      # print('{:.5e}'.format(self.MAE_split_value) + ', ' + '{:.5e}'.format(
```

```
            self.MSE_split_value))
91
92      def function(self, x):
93          return 1/(1 + 2 * x * x)
94
95
96  def main():
97      M = 100 #グラフ作成用にM分割してM+1点のラグランジュ補間を計算
98      sample_1 = 42 #区間1の標本点の数
99      sample_2 = 1 #区間2の標本点の数
100     sample_3 = 42 #区間3の標本点の数
101
102     L = Lagrange(M, sample_1, sample_2, sample_3)
103     L.plot()
104
105 if __name__ == '__main__':
106     main()
```

3.3.6 結果

中央の区間の標本点を 1 とし, 端の区間は左右同じ数の標本点の数 (2 から 30) にし, 実行した場合の結果は以下のようになる.

端の区間の標本数	MAE	MSE	端の区間の標本数	MAE	MSE
2	6.58484e-01	6.77056e-01	17	1.02216e-02	5.12289e-04
3	2.83654e-01	1.25943e-01	18	9.03194e-03	4.11318e-04
4	1.56520e-01	4.29054e-02	19	8.03507e-03	3.32077e-04
5	9.97232e-02	2.06821e-02	20	7.15597e-03	2.69493e-04
6	7.01780e-02	1.21302e-02	21	6.38271e-03	2.19761e-04
7	5.29105e-02	7.91942e-03	22	5.70331e-03	1.80009e-04
8	4.17605e-02	5.49945e-03	23	5.10553e-03	1.48055e-04
9	3.39648e-02	3.96919e-03	24	4.58075e-03	1.22228e-04
10	2.83098e-02	2.94031e-03	25	4.11736e-03	1.01247e-04
11	2.38976e-02	2.21926e-03	26	3.70730e-03	8.41225e-05
12	2.03193e-02	1.69889e-03	27	3.34406e-03	7.00836e-05
13	1.75834e-02	1.31518e-03	28	3.02159e-03	5.85293e-05
14	1.52640e-02	1.02759e-03	29	2.73478e-03	4.89862e-05
15	1.33056e-02	8.09273e-04	30	2.47918e-03	4.10795e-05
16	1.16414e-02	6.41828e-04	31	2.25091e-03	3.45103e-05

表 8　端の区間の標本数と精度評価の変化

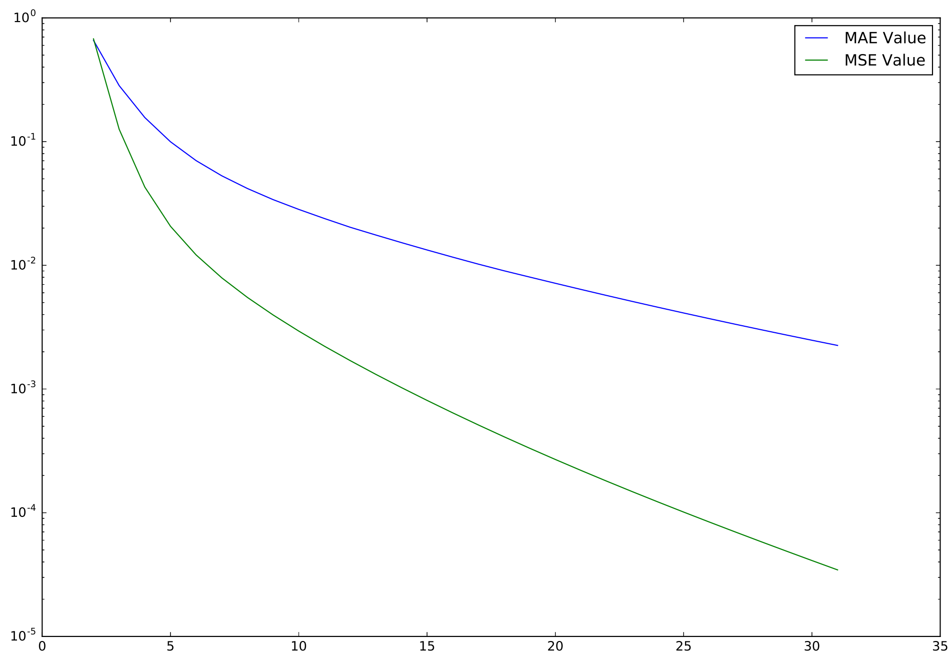

図 18　標本数の変化と精度評価の変化

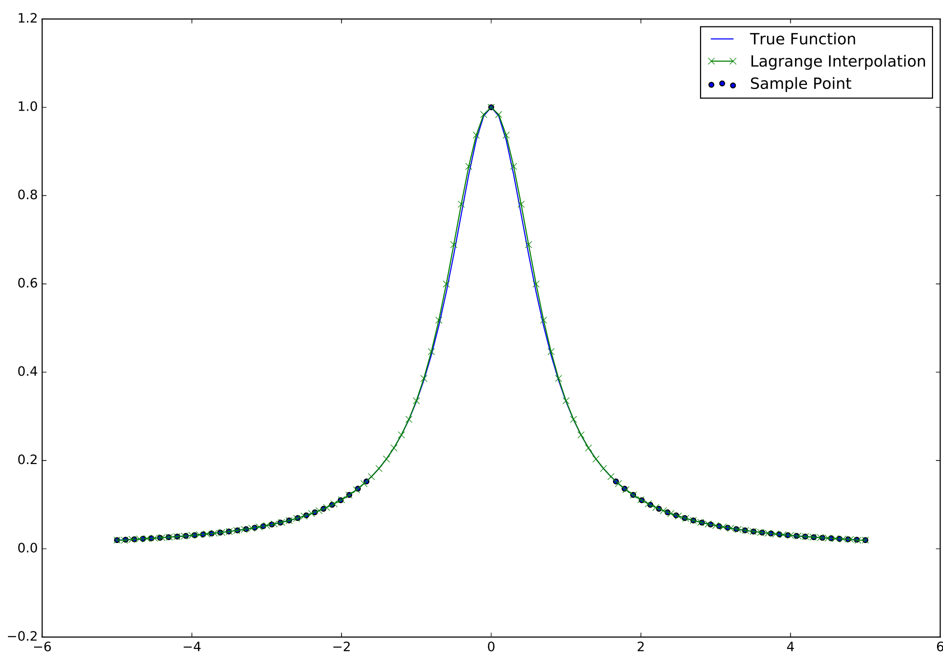

図 19　左の区間から順に 30, 1, 30 の標本数にした際のラグランジュ補完結果

3.3.7　考察

結果より, 等分にした場合は端の区間の標本数を増加させると精度が上昇することがわかる. また, 図 19 より, 端の区間の精度も非常によくなっていることがわかる.

3.3.8 おまけ

精度の変わり方についてどれだけ大きくすると精度が悪化するのかについて検証した結果が以下のとおりである.

端の区間の標本数	MAE	MSE	端の区間の標本数	MAE	MSE
2	6.58484e-01	6.77056e-01	36	1.42155e-03	1.47526e-05
3	2.83654e-01	1.25943e-01	37	1.30253e-03	1.25030e-05
4	1.56520e-01	4.29054e-02	38	1.19551e-03	1.06445e-05
5	9.97232e-02	2.06821e-02	39	1.09582e-03	9.05558e-06
6	7.01780e-02	1.21302e-02	40	1.00604e-03	7.65067e-06
7	5.29105e-02	7.91942e-03	41	9.15170e-04	6.45423e-06
8	4.17605e-02	5.49945e-03	42	7.35073e-04	4.63023e-06
9	3.39648e-02	3.96919e-03	43	1.18424e-03	1.64050e-05
10	2.83098e-02	2.94031e-03	44	2.04365e-03	3.80531e-05
11	2.38976e-02	2.21926e-03	45	3.54356e-03	1.21659e-04
12	2.03193e-02	1.69889e-03	46	1.49804e-02	2.67875e-03
13	1.75834e-02	1.31518e-03	47	1.87450e-02	3.68014e-03
14	1.52640e-02	1.02759e-03	48	1.26213e-01	2.23262e-01
15	1.33056e-02	8.09273e-04	49	2.59498e-01	6.69353e-01
16	1.16414e-02	6.41828e-04	50	4.81724e-01	2.01719e+00
17	1.02216e-02	5.12289e-04	51	1.03754e+00	1.37135e+01
18	9.03194e-03	4.11318e-04	52	3.01635e+00	8.86865e+01
19	8.03507e-03	3.32077e-04	53	6.78660e+00	4.81849e+02
20	7.15597e-03	2.69493e-04	54	2.56902e+01	7.18790e+03
21	6.38271e-03	2.19761e-04	55	5.74863e+01	4.66181e+04
22	5.70331e-03	1.80009e-04	56	1.05586e+02	1.75622e+05
23	5.10553e-03	1.48055e-04	57	3.25045e+02	1.28700e+06
24	4.58075e-03	1.22228e-04	58	2.56870e+03	1.07184e+08
25	4.11736e-03	1.01247e-04	59	4.33173e+03	2.04844e+08
26	3.70730e-03	8.41225e-05	60	1.01748e+04	1.52990e+09
27	3.34406e-03	7.00836e-05	61	3.78168e+04	2.13814e+10
28	3.02159e-03	5.85293e-05	62	1.62902e+05	3.76264e+11
29	2.73478e-03	4.89862e-05	63	2.37522e+05	5.74521e+11
30	2.47918e-03	4.10795e-05	64	6.53877e+05	6.07992e+12
31	2.25091e-03	3.45103e-05	65	1.03844e+06	1.26734e+13
32	2.04661e-03	2.90386e-05	66	3.27540e+06	1.35559e+14
33	1.86333e-03	2.44699e-05	67	7.01313e+06	5.76506e+14
34	1.69866e-03	2.06512e-05	68	2.02740e+07	1.41224e+16
35	1.55300e-03	1.74482e-05	69	6.94055e+07	6.62297e+16

表 9　端の区間の標本数と精度評価

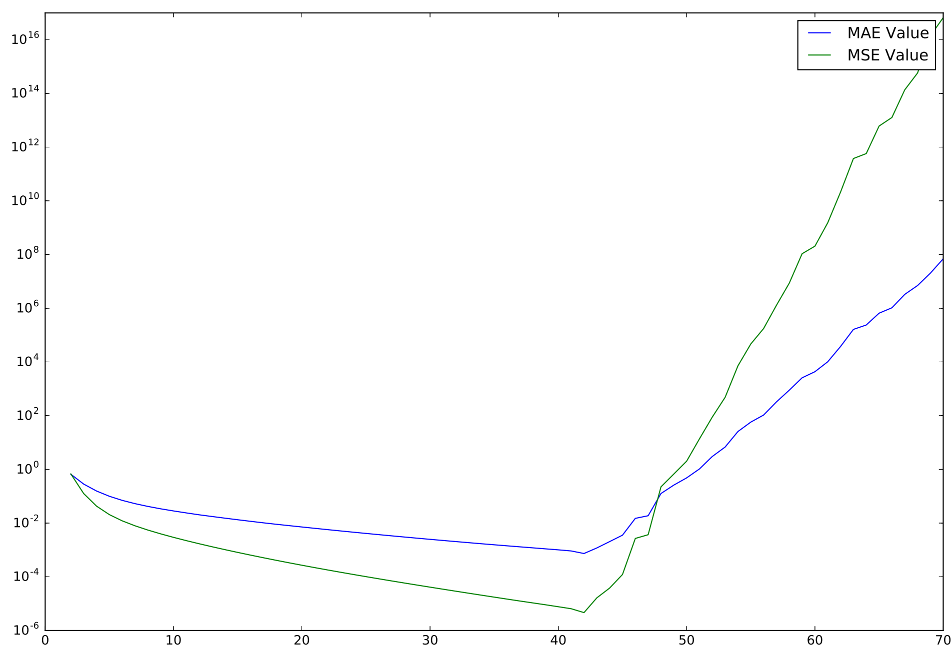

図 20　もう少し範囲を大きくした結果 (おまけ)

これより, 中央の区間の標本点の数が 1 のとき, 最も精度がよくなる端の区間のひょうほんてんの数は 42 であることがわかる.

3.3.9　標本点の理想の数についての検証

区間を 3 等分すると精度が標本点の数が上昇するのに合わせて良くなることがわかった. 次に, 点の最適な数について検証していく. これには新しくプログラムを書かず, ソースコード 5 を用いて検証を行う.

3.3.10 結果

標本点の合計数を 5 から 61 まで 2 づつ増やし, 端の区間の標本数は左右で同数, 真ん中の区間の標本数は奇数という条件で検証を行った. 以下はそれぞれの標本点の合計数毎に最も精度の良かったもののの組み合わせの一覧表とその時の精度を比較したグラフ, 区間ごとの標本点の数である.

標本数	端標本数	中央標本数	MAE	端標本数	中央標本数	MSE
5	1	3	6.58484e-01	1	3	6.77056e-01
7	2	3	2.83654e-01	2	3	1.25943e-01
9	3	3	1.56520e-01	3	3	4.29054e-02
11	4	3	9.97232e-02	4	3	2.06821e-02
13	5	3	7.01780e-02	5	3	1.21302e-02
15	6	3	5.29105e-02	6	3	7.91942e-03
17	7	3	4.17605e-02	7	3	5.49945e-03
19	8	3	3.39648e-02	8	3	3.96919e-03
21	9	3	2.83098e-02	9	3	2.94031e-03
23	10	3	2.38976e-02	10	3	2.21926e-03
25	11	3	2.03193e-02	11	3	1.69889e-03
27	12	3	1.75834e-02	12	3	1.31518e-03
29	13	3	1.52640e-02	13	3	1.02759e-03
31	14	3	1.33056e-02	14	3	8.09273e-04
33	15	3	1.16414e-02	15	3	6.41828e-04
35	16	3	1.02216e-02	16	3	5.12289e-04
37	17	3	9.03194e-03	17	3	4.11318e-04
39	18	3	8.03507e-03	18	3	3.32077e-04
41	19	3	7.15597e-03	19	3	2.69493e-04
43	20	3	6.38271e-03	20	3	2.19761e-04
45	21	3	5.70331e-03	21	3	1.80009e-04
47	22	3	5.10553e-03	22	3	1.48055e-04
49	23	3	4.58075e-03	23	3	1.22228e-04
51	24	3	4.11736e-03	24	3	1.01247e-04
53	25	3	3.70730e-03	25	3	8.41225e-05
55	25	5	3.32172e-03	26	3	7.00836e-05
57	26	5	2.66167e-03	27	3	5.85293e-05
59	27	5	2.13393e-03	28	3	4.89862e-05
61	28	5	1.71586e-03	28	5	3.00900e-05

表 10　最も精度の良かった標本点の組み合わせとその時の精度評価

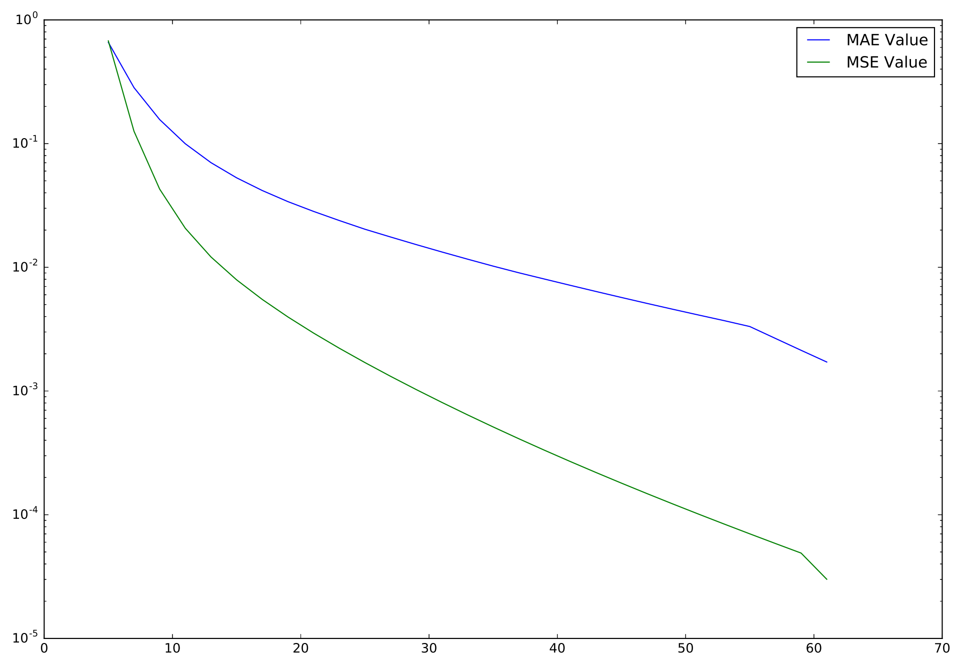

図 21　標本点の合計数と精度の最も良かった時の精度の変化

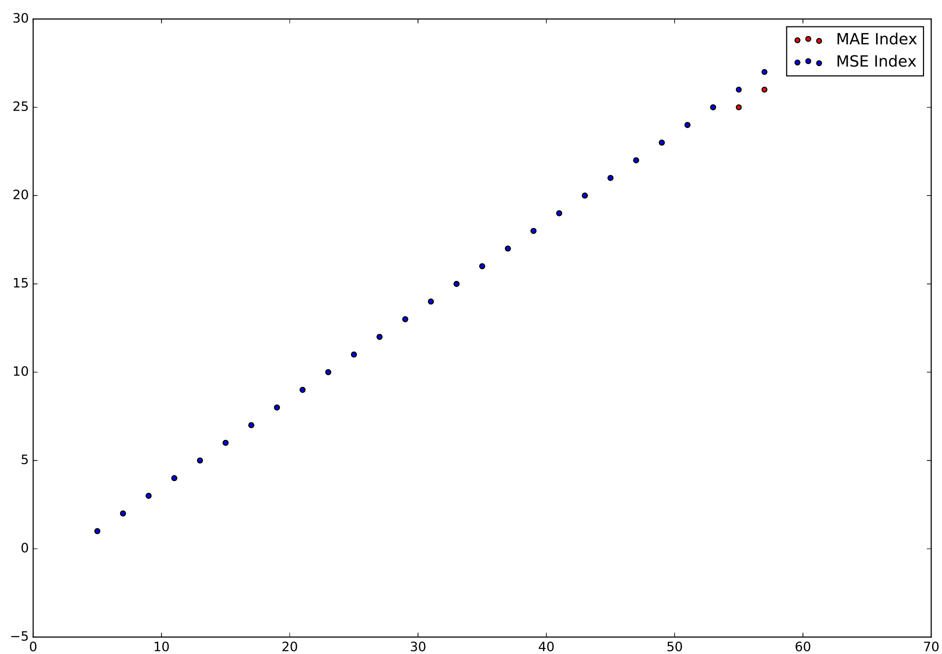

図 22　標本点の合計数と精度の最も良かった時の端の区間の標本点の数

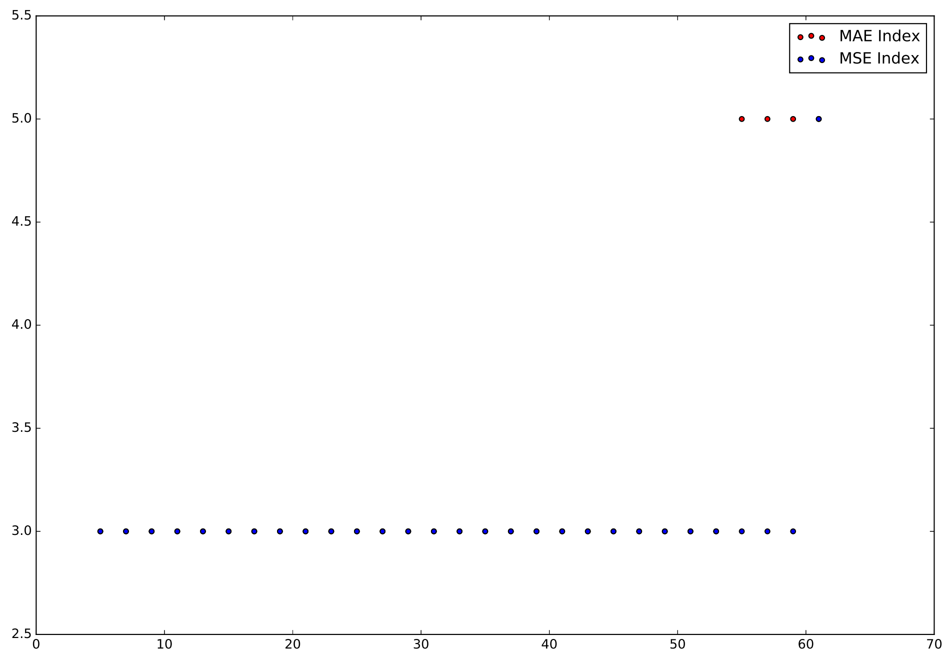

図 23　標本点の合計数と精度の最も良かった時の中央の区間の標本点の数

3.3.11　標本点の理想の数についての考察

以上の結果より, 標本点の数が増えると精度が上昇することがわかった. また, 中央の区間の標本点の数は, 標本点の合計数が 50 以下であれば 3 でよく, それ以上の場合はもう少し増えることもあるということがわかった. しかし, 今回のこの結果での最も精度のよかったラグランジュ補間の結果は以下の通りとなった.

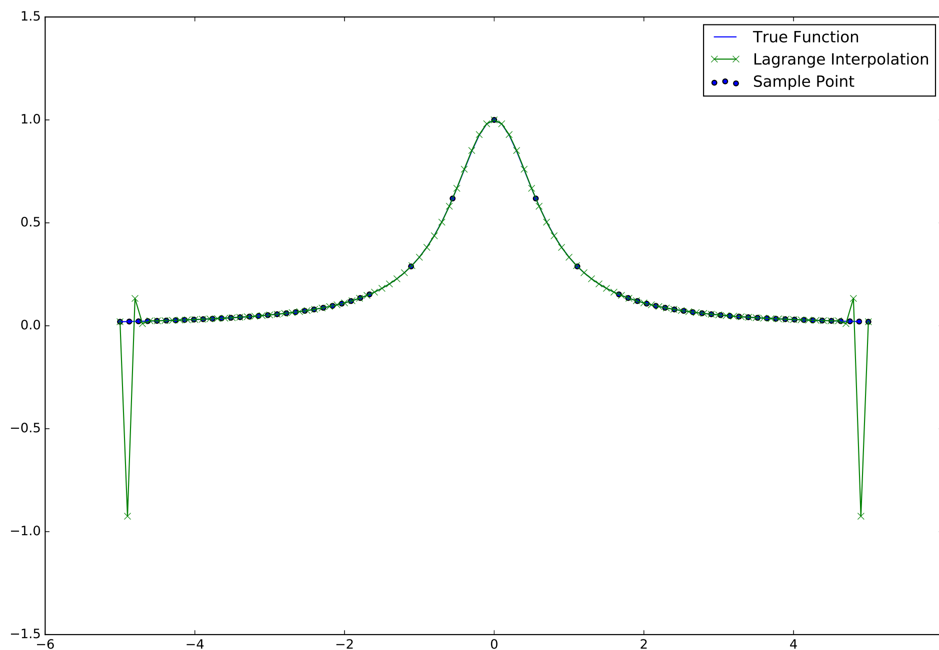

図 24　結果のもっともよかったラグランジュ補間の結果 (1)

精度の評価方法として, 1 点のみ外れ値があり, 見た目の上では図 24 より図 19 の方がきれいに補間できている. これは標本点の数とラグランジュ補間の補間を行う点の数が近すぎるために起こっているものと考えることができる. このようなことから, 補間を行う分割点を 1000 に増やし, 再度検証を行う.

3.3.12 結果

以上をもとに先ほどと補間する点の数以外同様の条件で検証を行った結果は以下のようになった.

標本数	端標本数	中央標本数	MAE	端標本数	中央標本数	MSE
5	1	3	6.64752e-01	1	3	6.83145e-01
7	2	3	2.86573e-01	2	3	1.27080e-01
9	3	3	1.58543e-01	3	3	4.32997e-02
11	4	3	1.01171e-01	4	3	2.08820e-02
13	5	3	7.13705e-02	5	3	1.22592e-02
15	6	3	5.39640e-02	6	3	8.01641e-03
17	7	3	4.27486e-02	7	3	5.58010e-03
19	8	3	3.49363e-02	8	3	4.04066e-03
21	9	3	2.91656e-02	9	3	3.00600e-03
23	10	3	2.47209e-02	10	3	2.28067e-03
25	11	3	2.11886e-02	11	3	1.75655e-03
27	12	3	1.83185e-02	12	3	1.36911e-03
29	13	3	1.59471e-02	13	3	1.07757e-03
31	14	3	1.39616e-02	14	3	8.55056e-04
33	15	3	1.22822e-02	15	3	6.83220e-04
35	16	3	1.08493e-02	16	3	5.49210e-04
37	17	3	9.61833e-03	17	3	4.43820e-04
39	18	3	8.55384e-03	18	3	3.60334e-04
41	19	3	7.62863e-03	19	3	2.93778e-04
43	20	3	6.82070e-03	20	3	2.40418e-04
45	21	3	6.11213e-03	21	3	1.97424e-04
47	22	3	5.48877e-03	22	3	1.62625e-04
49	23	3	4.93856e-03	23	3	1.34343e-04
51	24	3	4.45092e-03	24	3	1.11272e-04
53	25	3	4.01824e-03	25	3	9.23882e-05
55	26	3	3.63283e-03	26	3	7.68827e-05
57	27	3	3.28910e-03	27	3	6.41145e-05
59	28	3	2.98154e-03	28	3	5.35721e-05
61	29	3	2.70629e-03	29	3	4.48459e-05

表 11　最も精度の良かった標本点の組み合わせとその時の精度評価

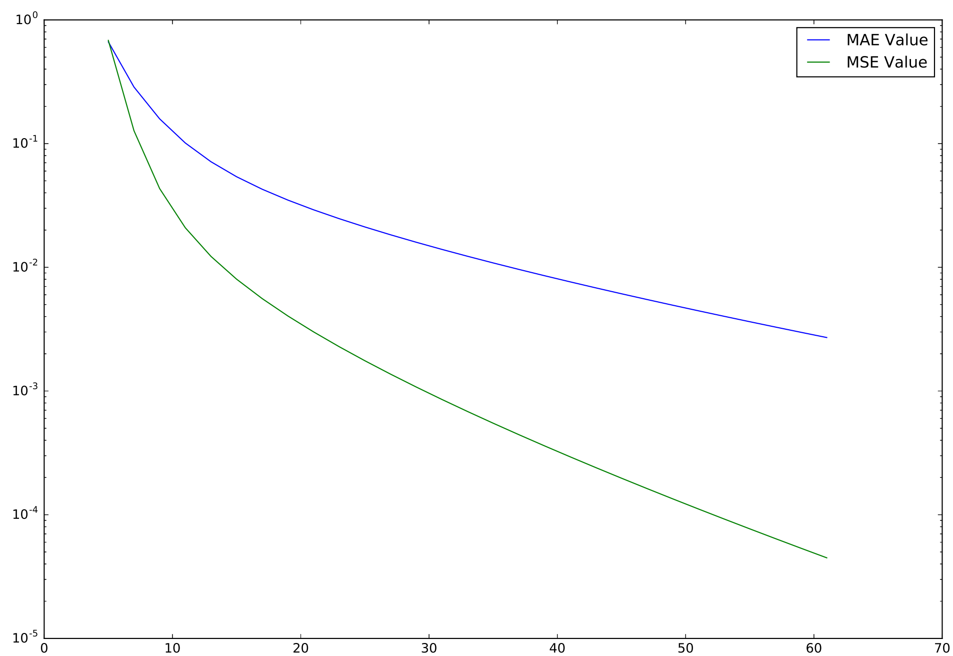

図 25　標本点の合計数と精度の最も良かった時の精度の変化

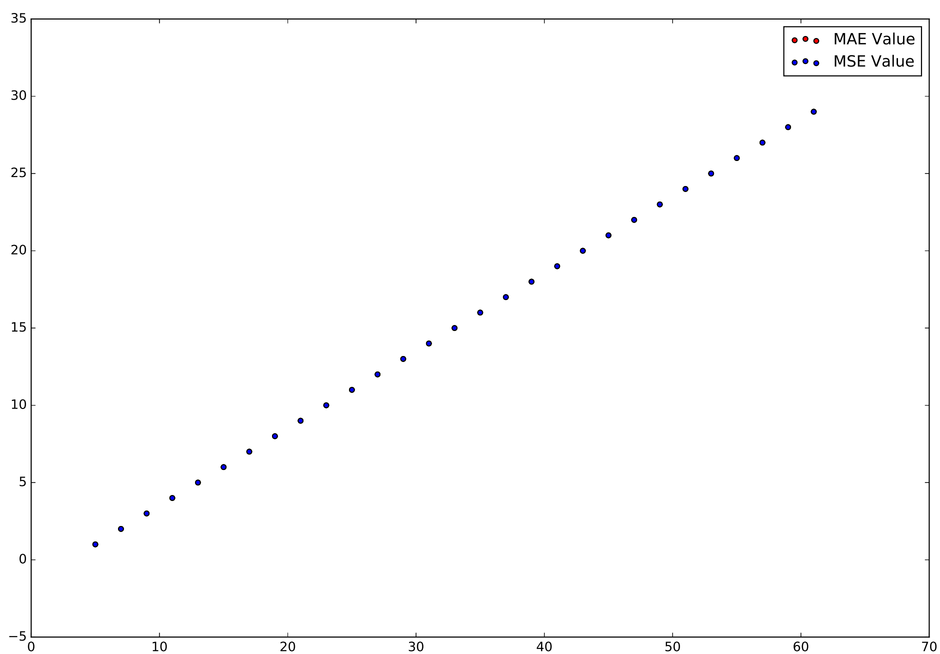

図 26　標本点の合計数と精度の最も良かった時の端の区間の標本点の数

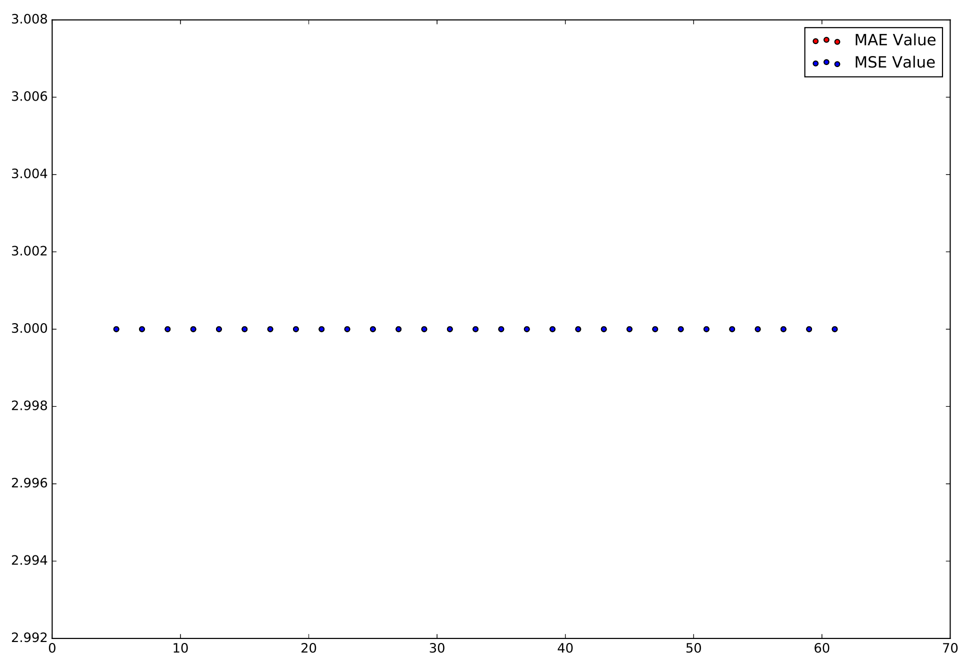

図 27　標本点の合計数と精度の最も良かった時の中央の区間の標本点の数

3.3.13　考察

補間点を増やした結果, すべての標本点の数の合計に対してそれぞれ最も精度の良かった際の中央標本点の数が 3 で一定となった. しかし, またこの時の補間したグラフは以下のようになった.

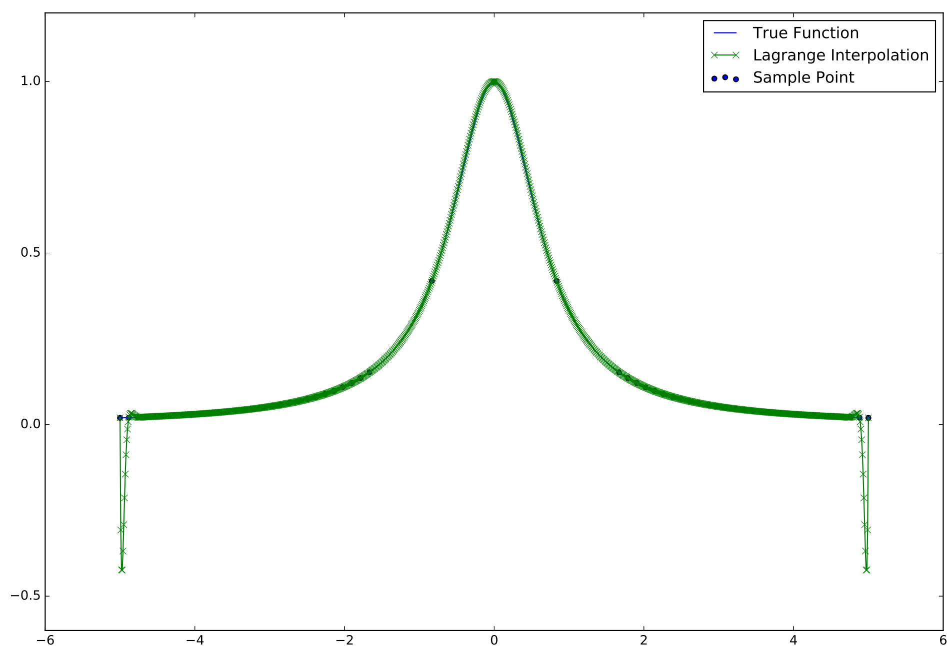

図 28　結果のもっともよかったラグランジュ補間の結果 (2)

先ほど同様にいくつかの点に外れ値ができてしまっている．これでは端の区間の精度を上げるという点に関して条件を満たしているとは言えないが, 図 19 は外れ値が存在しないことから, 中央の区間の標本点の数を 1 にすることで端の区間の精度がよい結果が得られることがわかる．以上より中央の区間は 1 とすることで外れ値のない綺麗な補間ができることがわかった．また, 中央の区間の標本点の数が 1 であるとき 3.3.8 より, 端の区間の標本点の数は 42 が最も良いという結果であった．その時の補間結果は以下のとおりである．ただし, 補完する区間の分割点は 100 にしたものである．

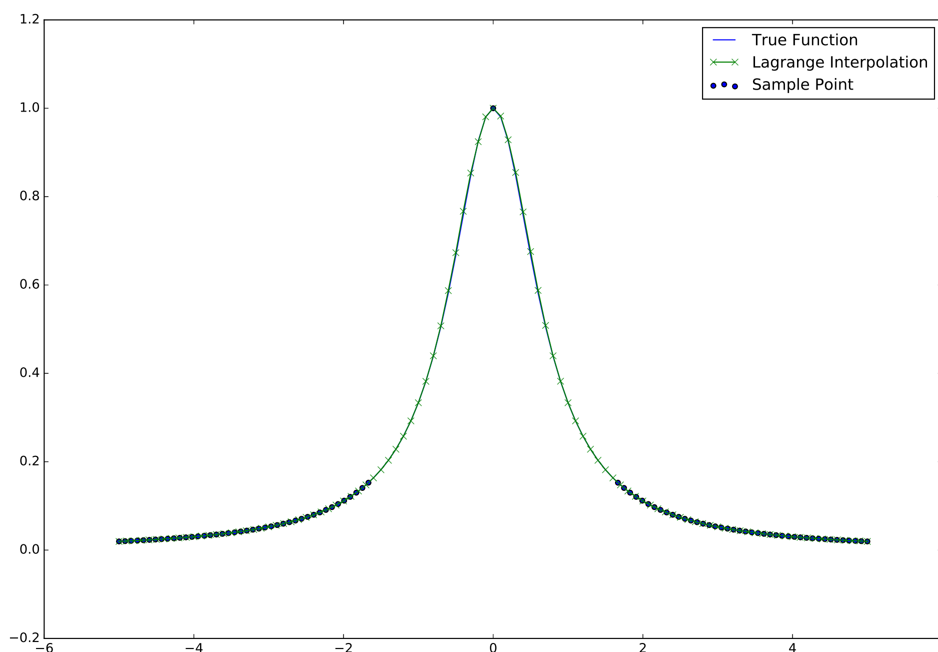

図 29　結果のもっともよかったラグランジュ補間の結果 (3)

4　スプライン補間

次にスプライン補間という補完法を実装し, その特徴について考察していく.

4.1　スプライン補間とは

スプライン補完とは, ラグランジュ補間と同様の補間法の一つであり, 区間ごとに異なる 3 次スプライン関数を用いて補間を行う方法であり, 3 次スプライン関数は以下を満たす.

スプライン補間 [3]

- スプライン関数

$$S(x) = S_k(x) = a_k(x - x_k)^3 + b_k(x - x_k)^2 + c_k(x - x_k) + d_k,$$

$$x_k \leq x \leq x_{k+1}, \;\; k = 0, 1, \cdots, n - 1$$

$$S''_k(x_k) = u_k, \;\; k = 0, 1, \cdots, n$$

- 係数

$$a_k = \frac{u_{k+1} - u_k}{6(x_{k+1} - x_k)}, \;\; b_k = \frac{u_k}{2}, \;\; d_k = f_k$$

$$c_k = \frac{f_{k+1} - f_k}{x_{k+1} - x_k} - \frac{1}{6}(u_{k+1} + 2u_k)(x_{k+1} - x_k)$$

4.2　補間する関数

スプライン補間はその特徴から，比較的複雑な関数でも補間ができそうであるため，以下の 3 つについて補間を行ってみる．

4.2.1　ラグランジュ補間でも補完した関数

まずは，ラグランジュ補間でも補間を行った関数である $f(x) = \dfrac{1}{1+2x^2}$ である．補間を行う区間は $[-5, 5]$ であり，概形は以下ようになる．

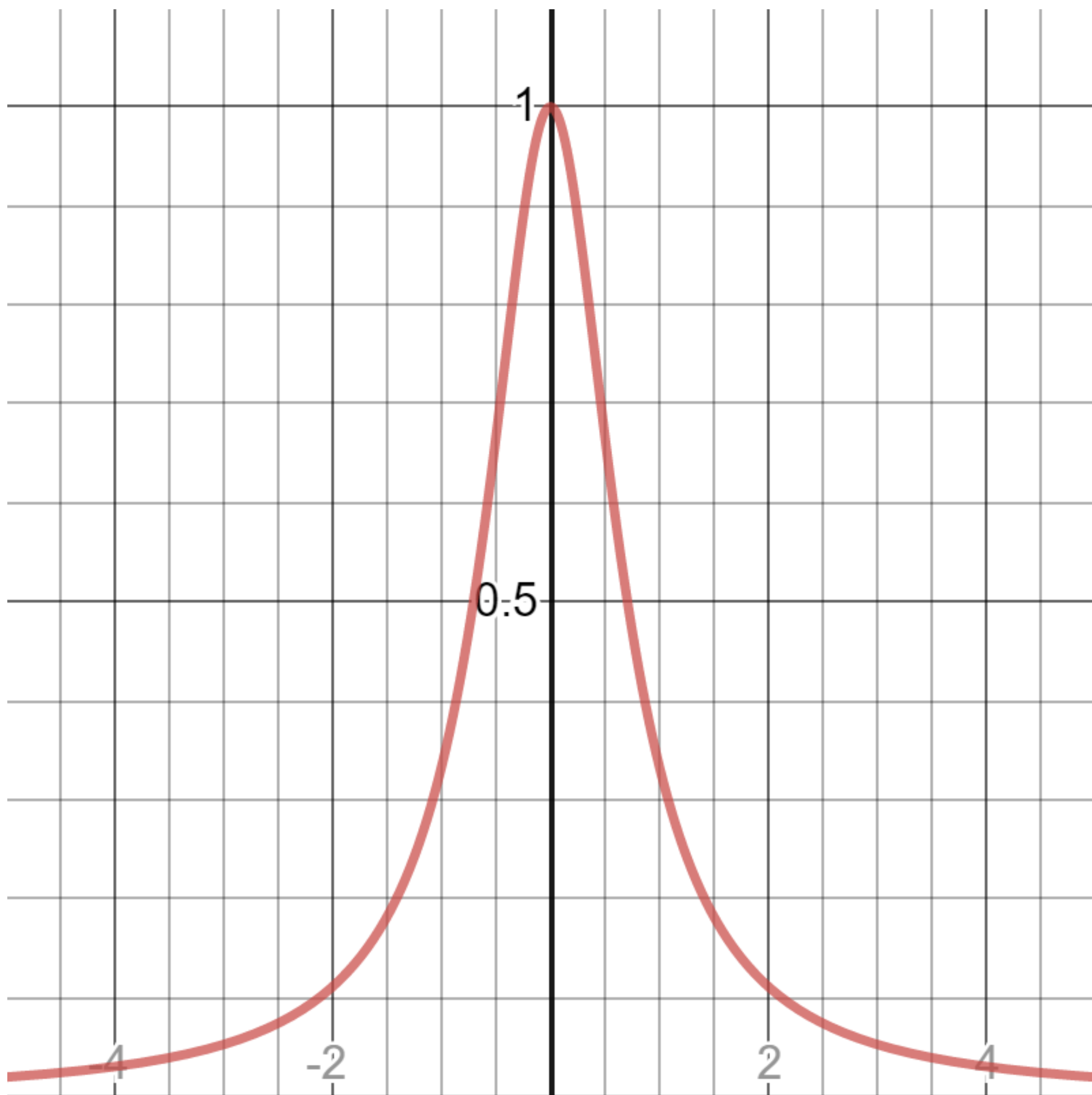

図 30　補間を行う関数のグラフ (1)

4.2.2　三角関数の組み合わせ

2 つ目は三角関数を組み合わせた式を補間する．式は $f(x) = \sin(\tan x)$．この関数の補間を行う区間は $[-4, 4]$ とし，概形は以下のようになる．

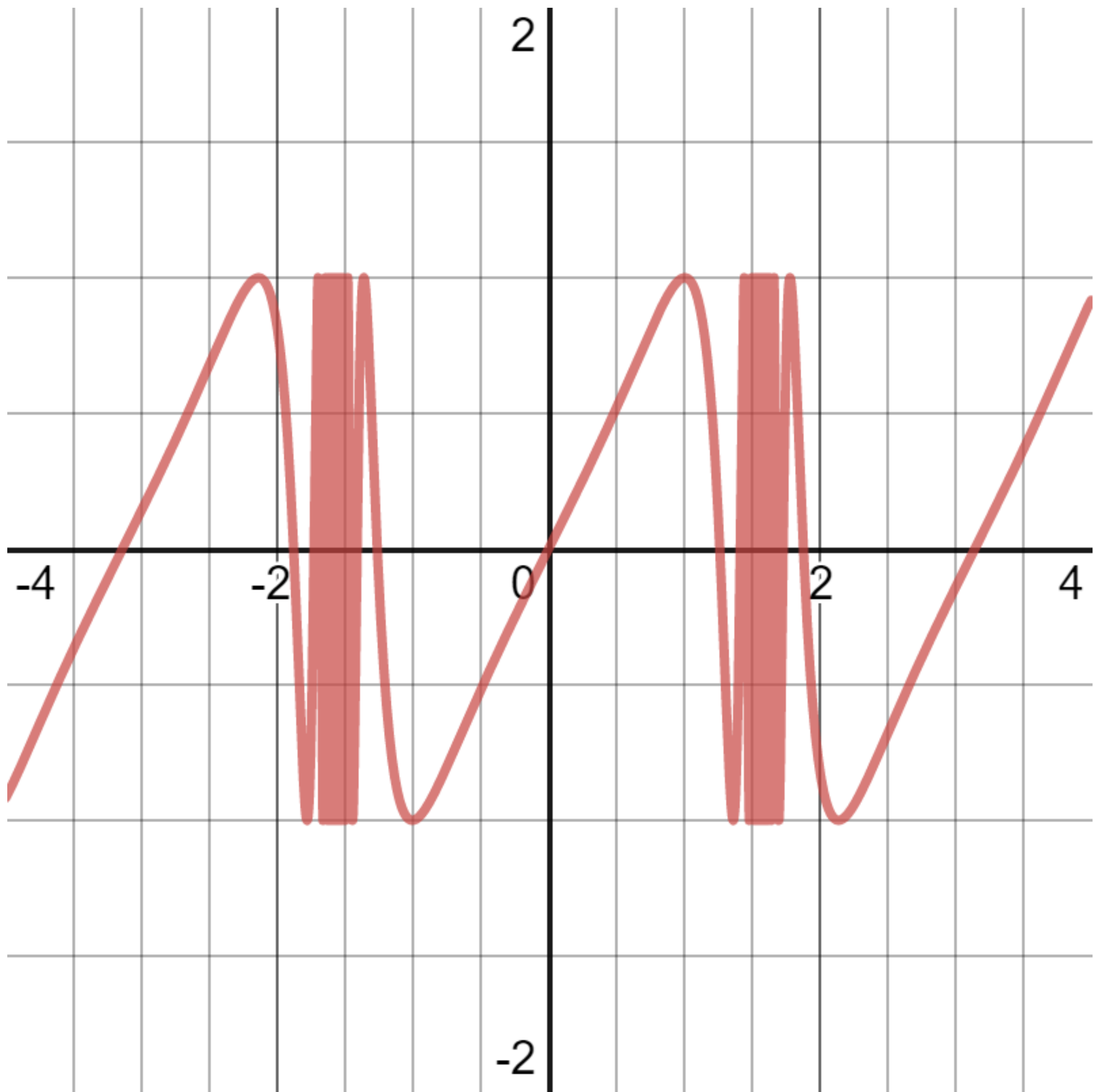

図 31　補間を行う関数のグラフ (2)

4.2.3　ハートの形の関数

2 つ目はハートの形になる関数である．式は $f(x) = \frac{\sqrt{1-(|x|-1)^2} + \arcsin(|x|-1) - \frac{\pi}{2}}{2} + \sin 100x \frac{\sqrt{1-(|x|-1)^2} - \arcsin(|x|-1) - \frac{\pi}{2}}{2}$ である．複雑な形であることから補間が難しいのではないかということで検証を行う．定義域より，この関数の補間を行う区間のみ他と異なり，$[-2,2]$ とする．概形は以下のとおりである．

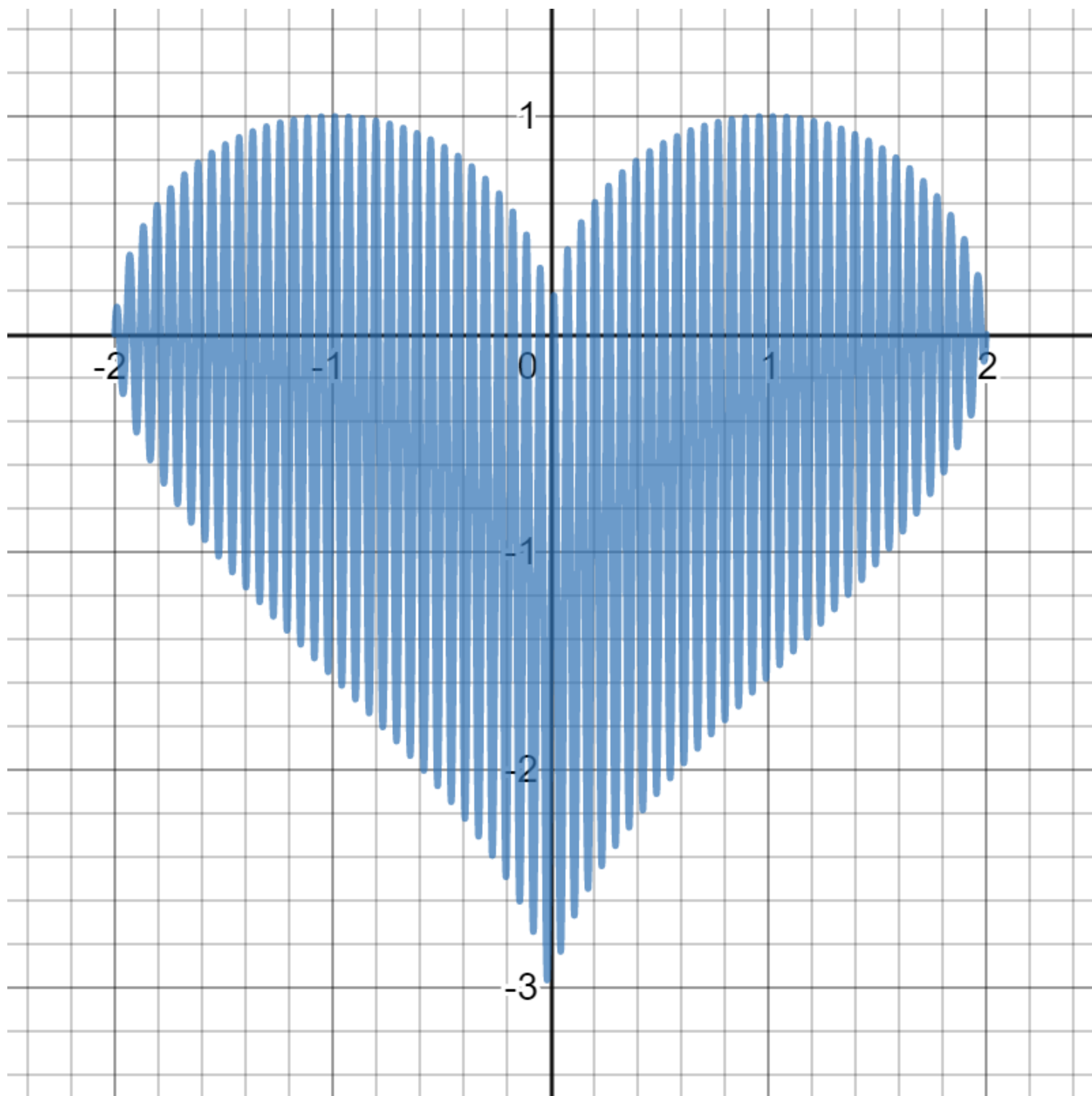

図 32　補間を行う関数のグラフ (3)

4.3　スプライン補間の実装

ラグランジュ補間と同様に Python にてスプライン補間を実装する．

4.3.1　変数

プログラム作成にあたり上記のプログラムを参考に以下のようにインスタンスを作成する際の変数を作成した

- division：補間する点の分割点 (補間点 - 1 個)
- start：開始点の数
- end：終了点の数
- sample：標本点の数 $(n+1)$ の値

4.3.2　プログラム

以上を用いてプログラムを次のように作成した.

ソースコード 7　spline.py

```
1  import math
2  import matplotlib.pyplot as plt
3  import numpy as np
4  from sklearn.metrics import mean_absolute_error
5  from sklearn.metrics import mean_squared_error
6
7  class Spline:
8      def __init__(self, division, start, end, sample):
9          self.division = division #補完する点の分割点（補完点 - 1 個）
10         self.start = start #開始点
11         self.end = end #終了点
12         self.sample = sample #標本点の数
13         self.n = sample - 1#標本点の数 - 1
14         self.length = end - start #範囲
15
16         self.make_f_in()
17         self.h = self.x_in[2] - self.x_in[1] #標本点の間隔（等間隔前提）
18
19         self.preparation()
20         self.make_f_out()
21
22     def make_f_in(self):
23         self.x_in = []
24         self.f_in = []
25         for i in range(0, self.sample):
26             self.x_in.append(self.start + i * self.length / (self.sample - 1))
27             self.f_in.append(self.function(self.start + i * self.length / (self.
                   sample - 1)))
28
29     def make_f_out(self):
30         self.x_out = []
```

```
31          self.f_out = []
32          self.true_f_out = []
33          self.error = []
34          self.k = 1 #区間のカウント
35          for i in range(0, self.division + 1):
36              self.x_out.append(self.start + i * self.length / self.division)
37              if self.x_out[i] > self.x_in[self.k + 1]:
38                  self.k += 1
39              self.f_out.append(self.complement(i, self.k))
40              self.true_f_out.append(self.function(self.x_out[i]))
41              self.error.append(np.abs(self.true_f_out[i] - self.f_out[i]))
42
43      def preparation(self):
44          self.A = np.diag(4*np.ones(self.n - 1)) + np.diag(np.ones(self.n
                -2),-1) + np.diag(np.ones(self.n-2),1)
45          self.g = []
46          for k in range(0, self.n - 1):
47              self.g.append(6 * (self.f_in[k] - 2 * self.f_in[k+1] + self.f_in[k
                    +2]) / (self.h * self.h))
48          self.u_temp = np.linalg.inv(self.A) @ self.g
49          self.u = np.append(0, self.u_temp)
50          self.u = np.append(self.u, 0)
51
52          self.a = []
53          self.b = []
54          self.c = []
55          self.d = []
56          for k in range(0, self.n):
57              self.a.append((self.u[k+1] - self.u[k]) / 6 * (self.x_in[k+1] -
                    self.x_in[k]))
58              self.b.append(self.u[k] / 2)
59              self.c.append((self.f_in[k+1] - self.f_in[k]) / (self.x_in[k+1] -
                    self.x_in[k]) - 1 / 6 * (self.u[k+1] + 2 * self.u[k]) * (self
                    .x_in[k+1] - self.x_in[k]))
60              self.d.append(self.f_in[k])
61
62      def complement(self, i, k):
63          return self.a[k] * (self.x_out[i] - self.x_in[k]) ** 3 + self.b[k] * (
                self.x_out[i] - self.x_in[k]) ** 2 + self.c[k] * (self.x_out[i] -
                self.x_in[k]) + self.d[k]
64
65      def plot(self):
66          plt.figure(figsize=(15, 10), dpi=50)
67
68          self.plot_range = np.arange(self.start, self.end, 0.001)
69          self.plot_function = self.function(self.plot_range)
```

```
70          plt.plot(self.plot_range, self.plot_function, label = u'True Function')
                # 真の関数
71
72          plt.scatter(self.x_in, self.f_in, label = u'Sample Point') # 標本点
73
74          plt.plot(self.x_out, self.f_out, marker="x", label = u'Spline
                Interpolation') # ラグランジュ補完
75
76          plt.legend()
77          name = "function" + str(self.sample) + "_complement.eps"
78          plt.savefig(name)
79          plt.show()
80
81
82      def value(self):
83          self.MAE_value = mean_absolute_error(self.true_f_out, self.f_out)
84          self.MSE_value = mean_squared_error(self.true_f_out, self.f_out)
85          print('{:.5e}'.format(self.MAE_value) + ', ' + '{:.5e}'.format(self.
                MSE_value))
86
87      def function(self, x):
88          return 1/(1 + 2 * x * x)
89          #return np.sin(np.tan(x))
90          #return (self.function_a(x) + self.function_b(x)) / 2 + np.sin(100 * x
                ) * (self.function_a(x) - self.function_b(x)) / 2
91
92      def function_a(self, x):
93          return np.sqrt(1 - ( np.abs(x) - 1)**2)
94
95      def function_b(self, x):
96          return (np.arcsin(np.abs(x)-1) - math.pi / 2)
97
98  def main():
99      M = 100 #グラフ作成用にM分割してM+1点のラグランジュ補間を計算
100     start = -5 #範囲の開始位置
101     end = 5 #範囲の終了位置
102     sample = 101 #標本点の数
103
104     S = Spline(M, start, end, sample)
105     S.plot()
106
107 if __name__ == '__main__':
108     main()
```

4.4 結果

スプライン補間を行う区間の分割点を 100 にし, 標本点の数を変えプログラムを実行した結果以下のような結果が得られた. また, 補間点と真の関数との誤差のグラフも同様に以下のようになった.

4.4.1 ラグランジュ補間でも補完した関数の補間結果

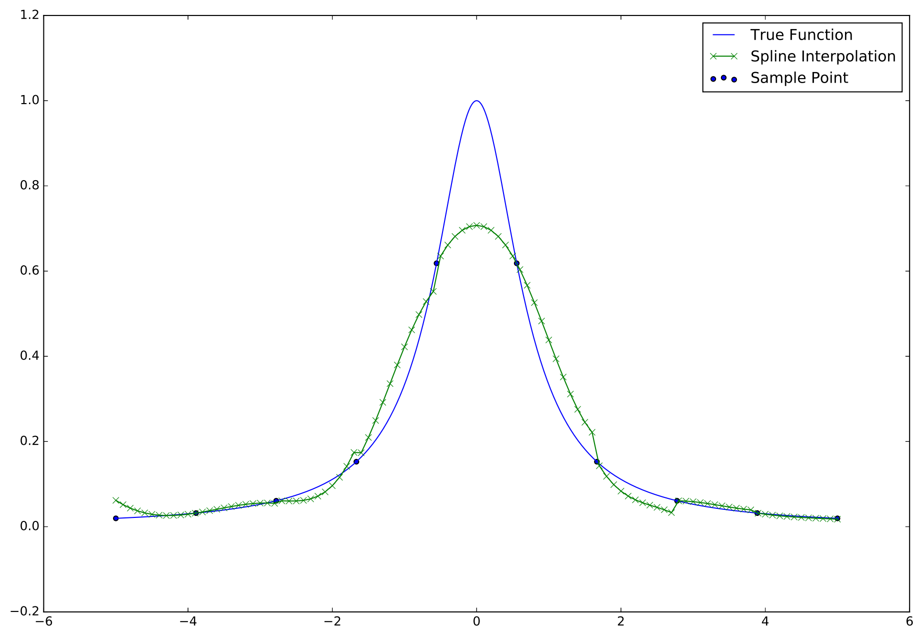

図 33　標本点の数が 10 の時の補間結果

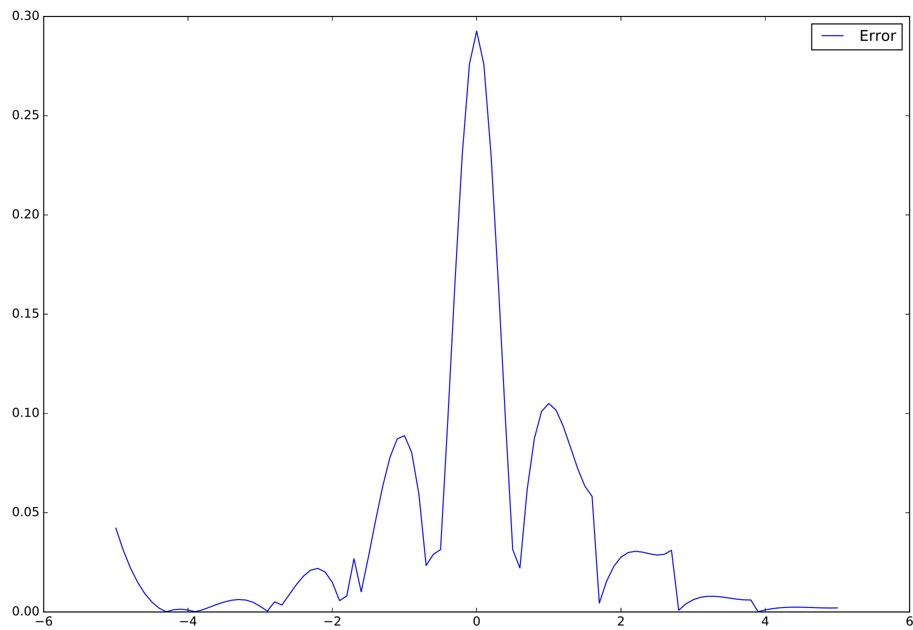

図 34　標本点の数が 10 の時の補間誤差

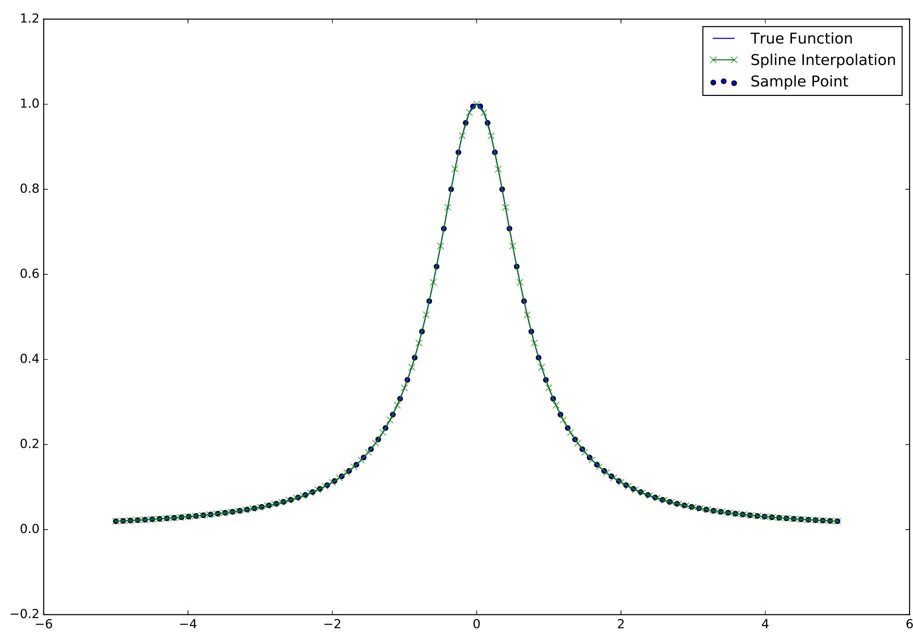

図 35　標本点の数が 100 の時の補間結果

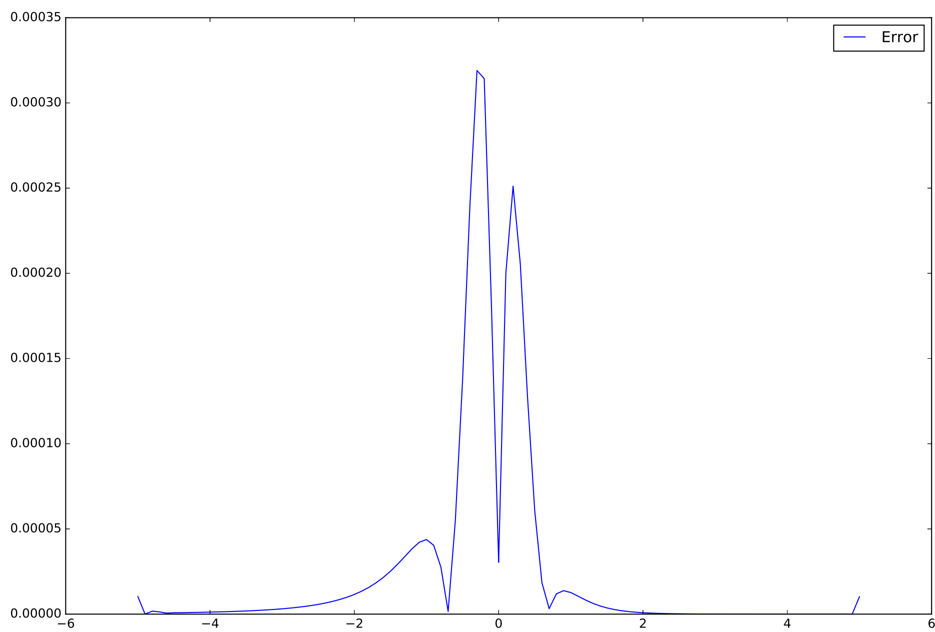

図 36　標本点の数が 100 の時の補間誤差

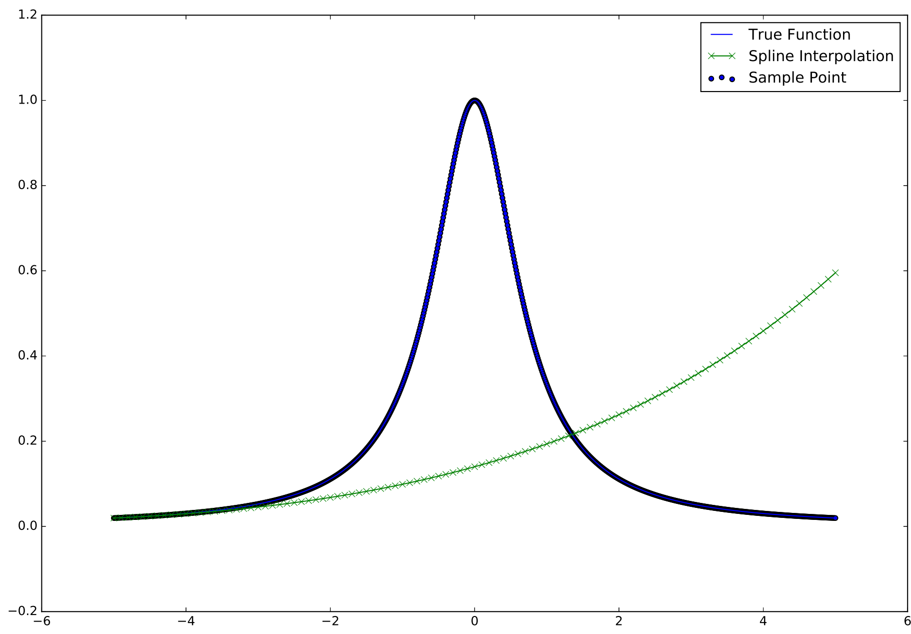

図 37　標本点の数が 1000 の時の補間結果

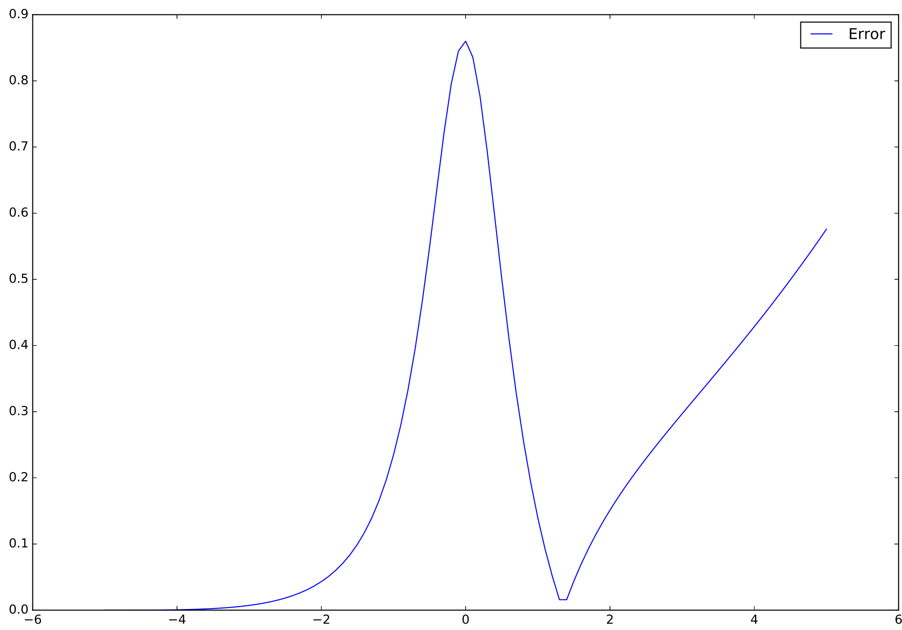

図 38　標本点の数が 1000 の時の補間誤差

4.4.2　三角関数の組み合わせの関数の補間結果

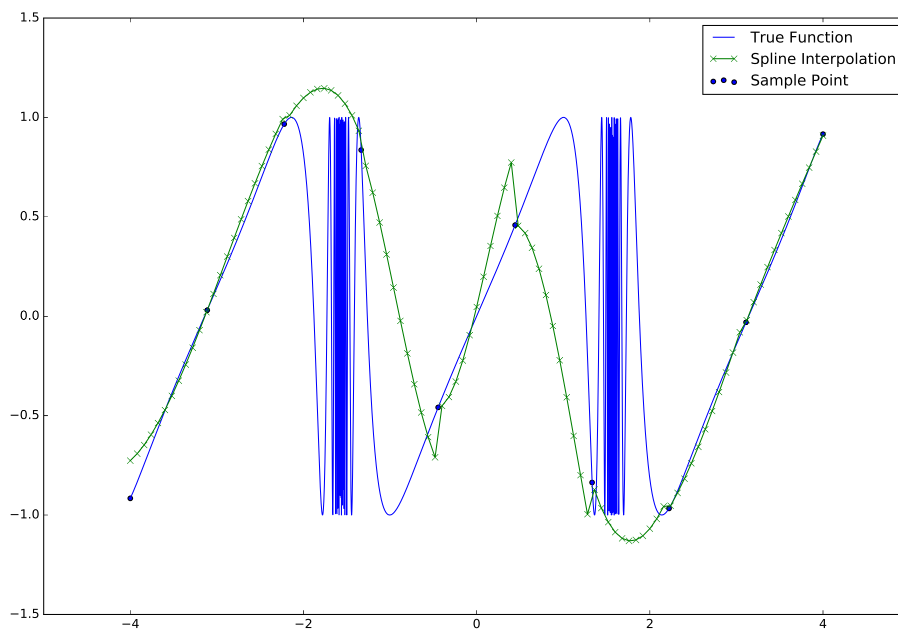

図 39　標本点の数が 10 の時の補間結果

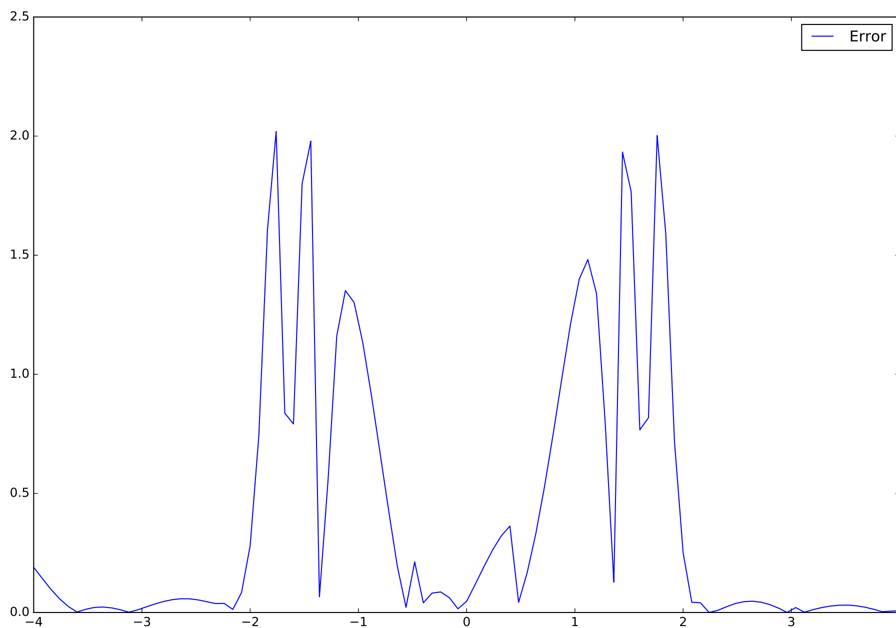

図 40　標本点の数が 10 の時の補間誤差

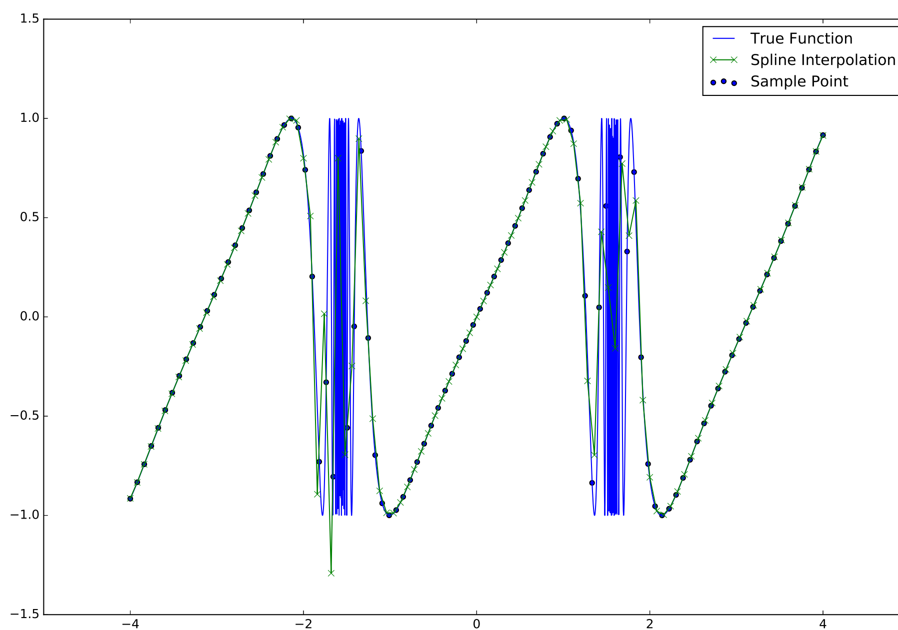

図 41　標本点の数が 100 の時の補間結果

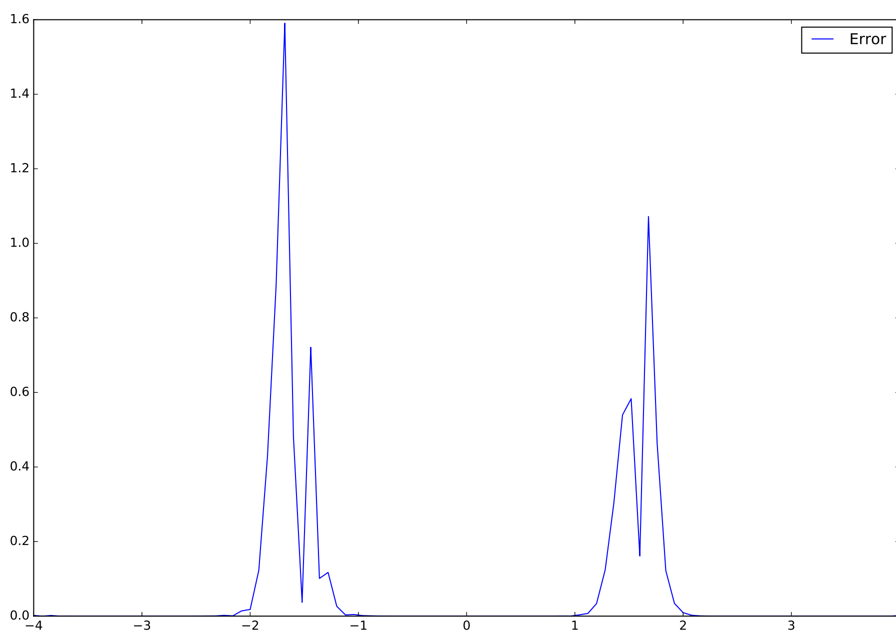

図 42　標本点の数が 100 の時の補間誤差

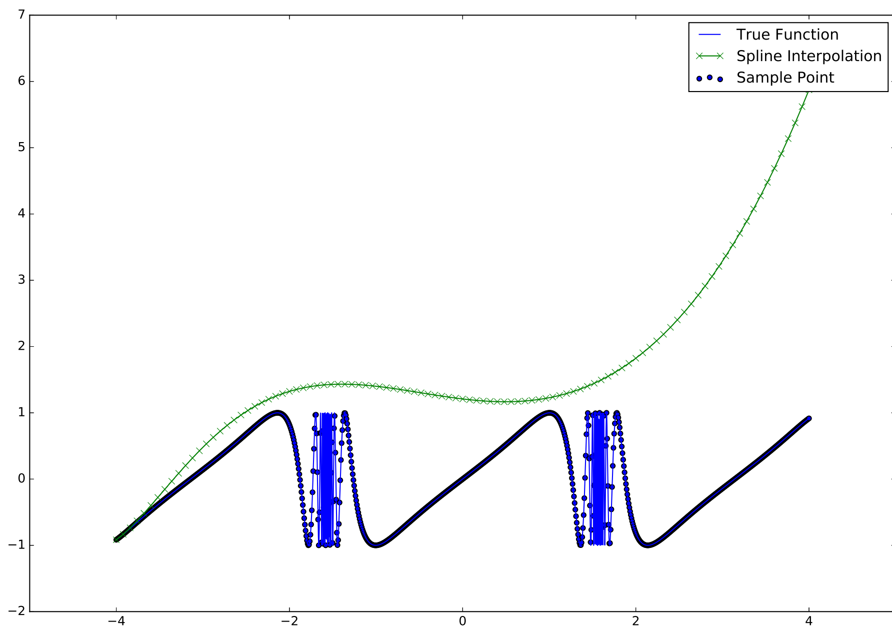

図 43　標本点の数が 1000 の時の補間結果

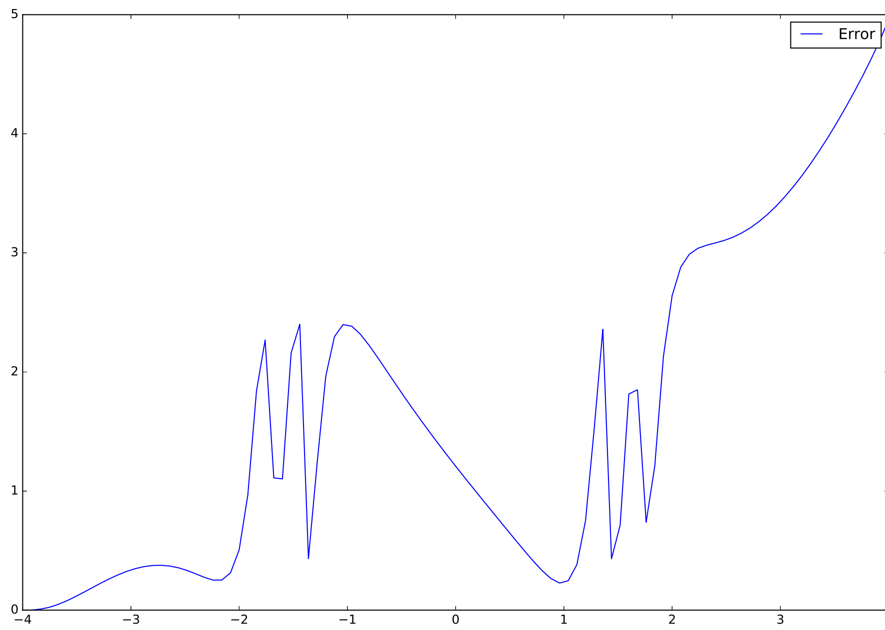

図 44　標本点の数が 1000 の時の補間誤差

4.4.3 ハートの形の関数の補間結果

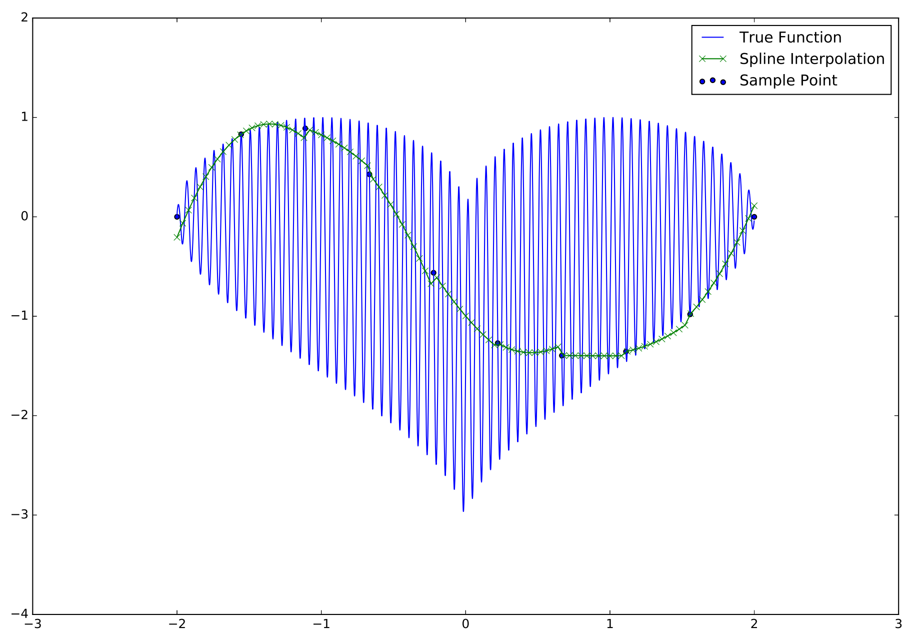

図 45　標本点の数が 10 の時の補間結果

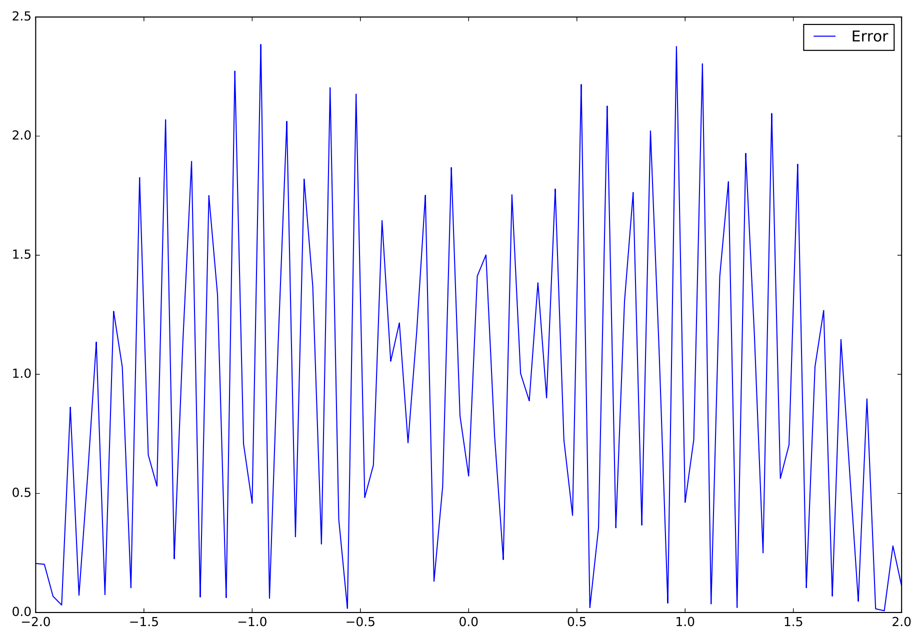

図 46　標本点の数が 10 の時の補間誤差

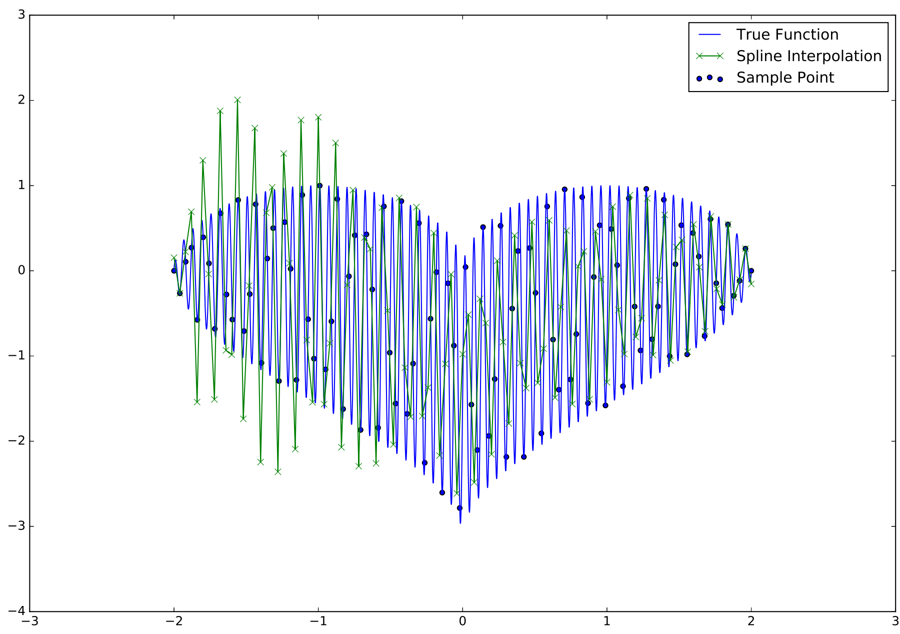

図 47　標本点の数が 100 の時の補間結果

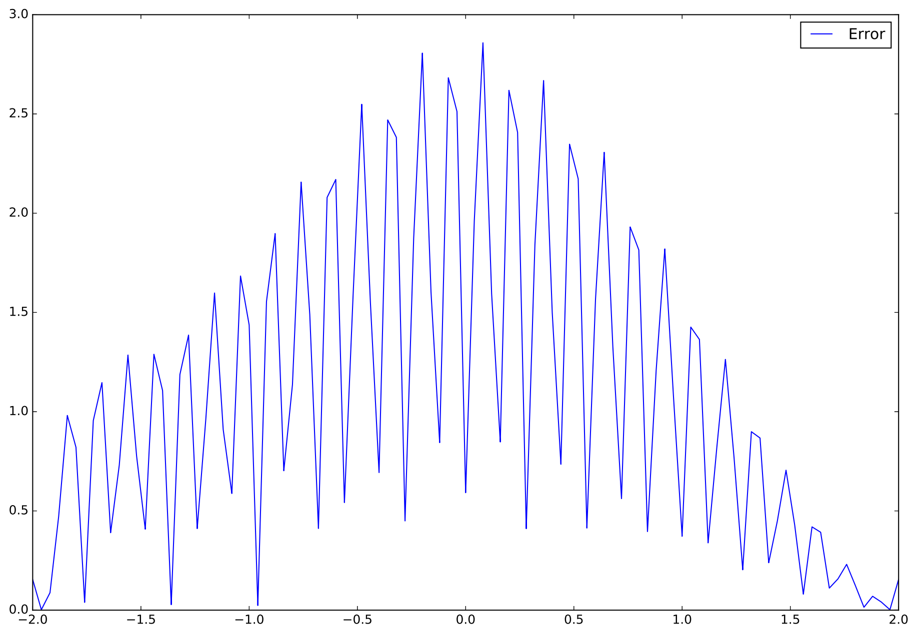

図 48　標本点の数が 100 の時の補間誤差

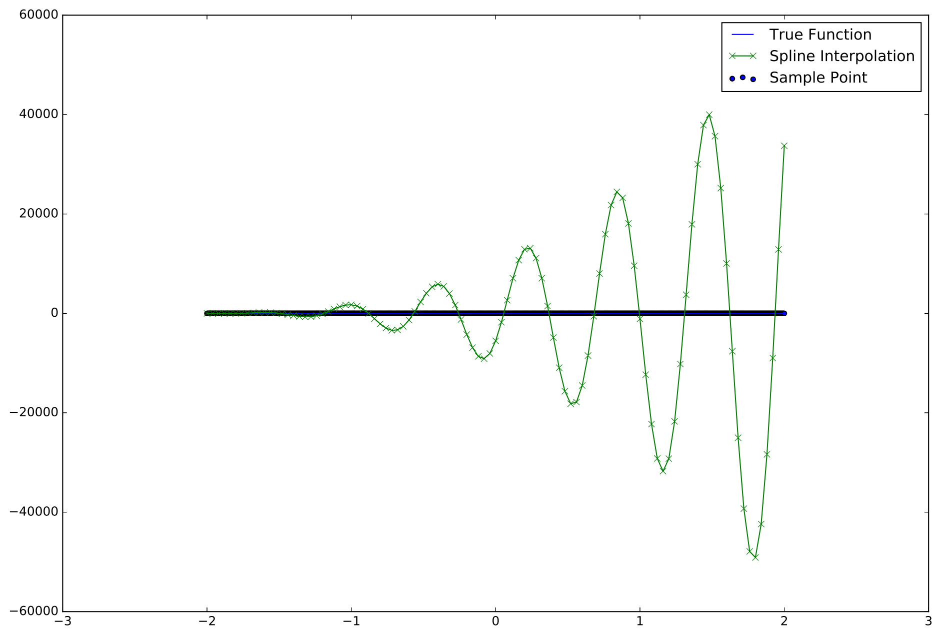

図 49　標本点の数が 1000 の時の補間結果

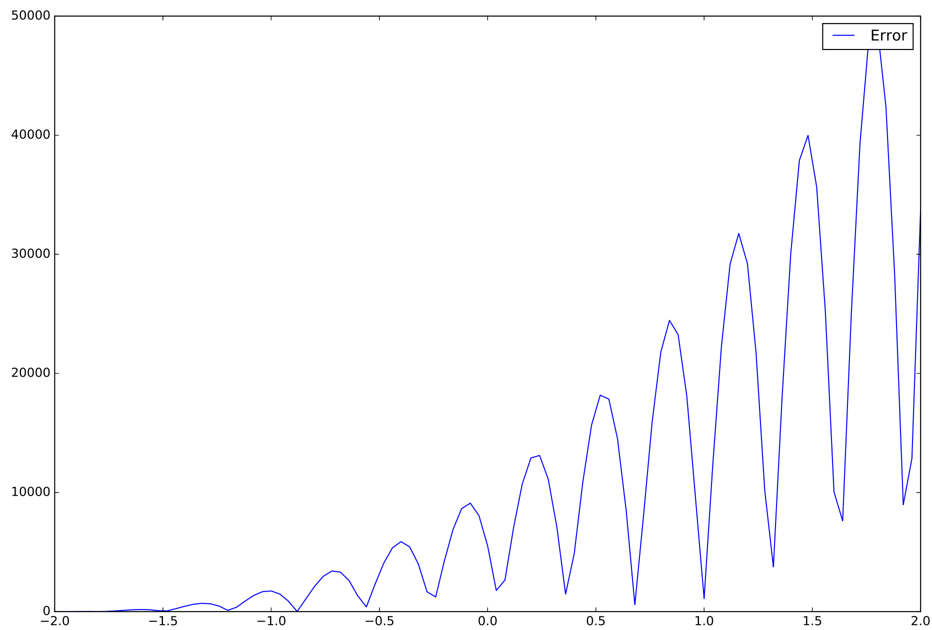

図 50　標本点の数が 1000 の時の補間誤差

結果より, $f(x) = \dfrac{1}{1+2x^2}$, $f(x) = \sin(\tan x)$ の 2 つの関数に関しては, 標本点の数が 10 より 100 の方が精度がよくなっているように思える. 誤差の出方に関しては, さまざまで, ラグランジュ補間と

比較をすると, ラグランジュ補間は $x = 0$ を境に左右が対称になるのに対して, スプライン補間は左右対称でないことがわかる.

4.5 標本点の数と補間を行う点の数による差に関する検証

標本点の数と補間を行う点の数によって全体的な近似精度はどのように変化しているかを定量的に考察する. 検証を行う際にはラグランジュ補間の時と同様に平均絶対誤差 (MAE : Mean Absolute Error) と平均二乗誤差 (MSE : Mean Squared Error）を用い, 評価を行う.

4.6　精度の評価結果

標本点の数を 3 から, 1000 までの精度評価結果は以下のとおりである.

4.6.1　ラグランジュ補間でも補完した関数

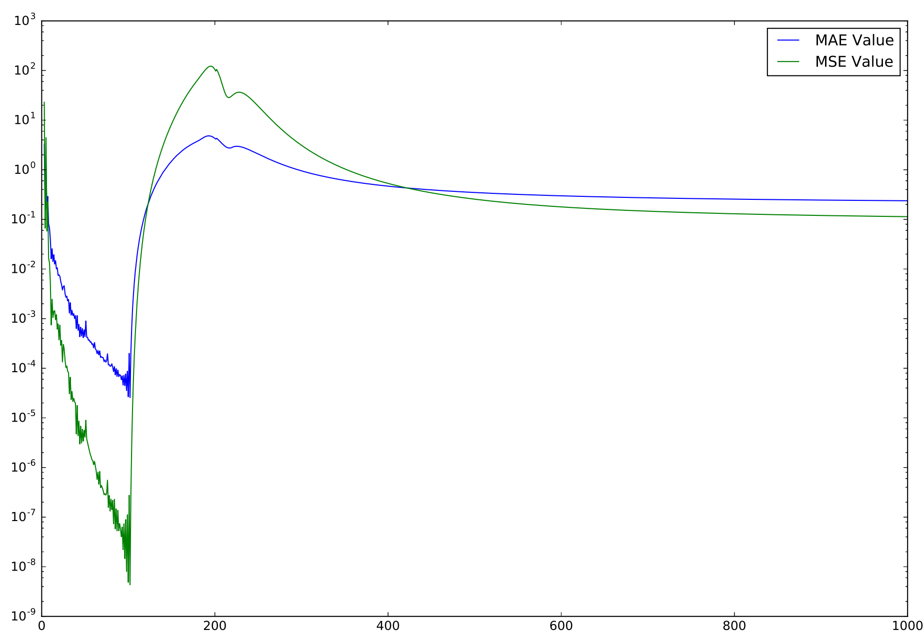

図 51　補間の分割が 100 の時の標本点の数による精度の変化

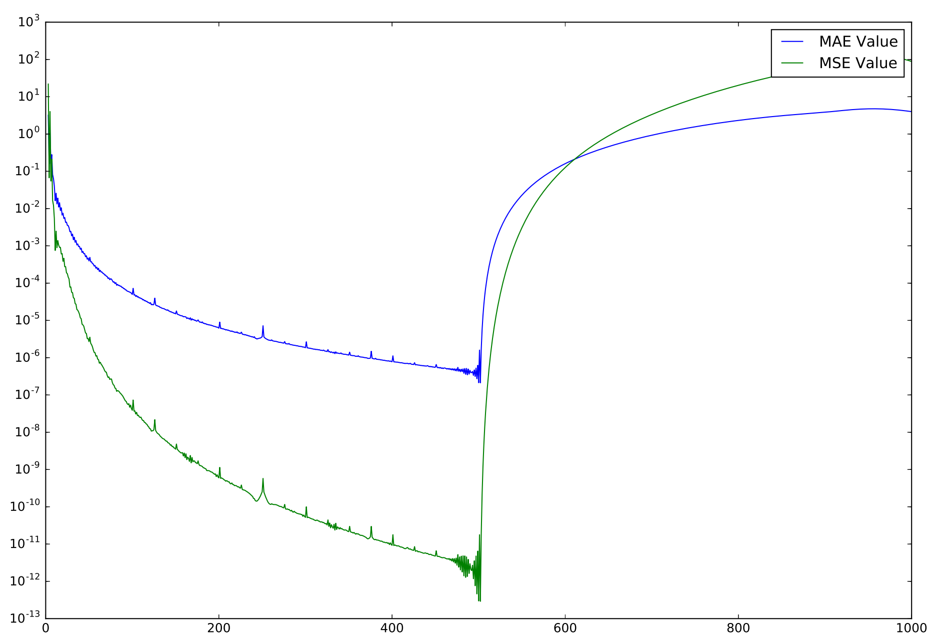

図 52　補間の分割が 500 の時の標本点の数による精度の変化

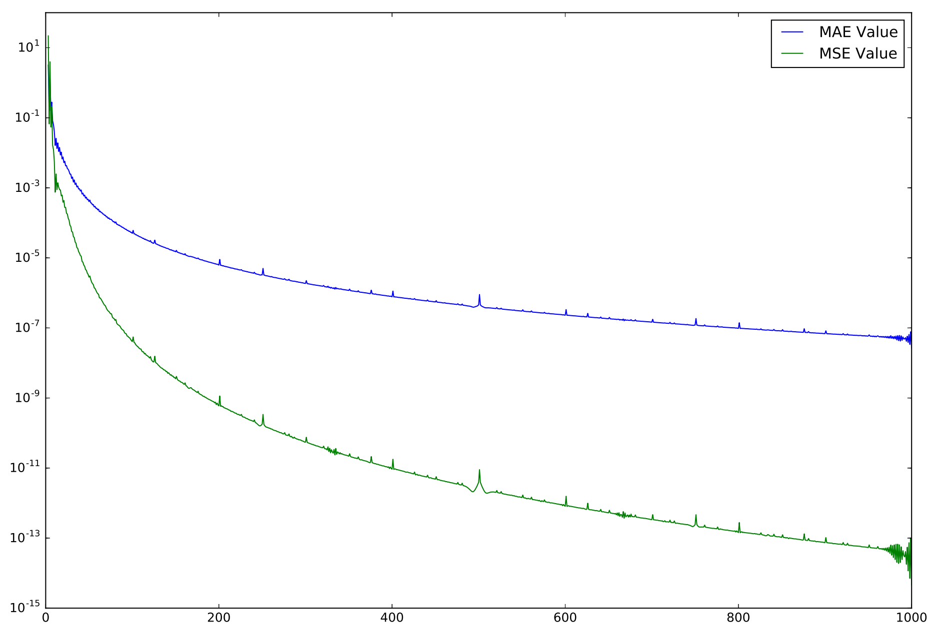

図 53　補間の分割が 1000 の時の標本点の数による精度の変化

4.6.2　三角関数の組み合わせ

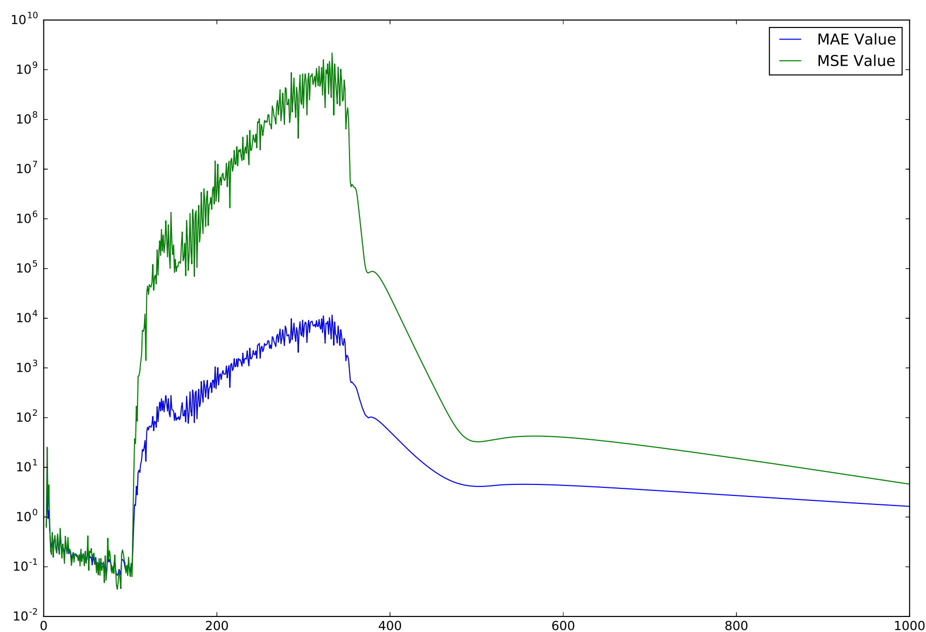

図 54　補間の分割が 100 の時の標本点の数による精度の変化

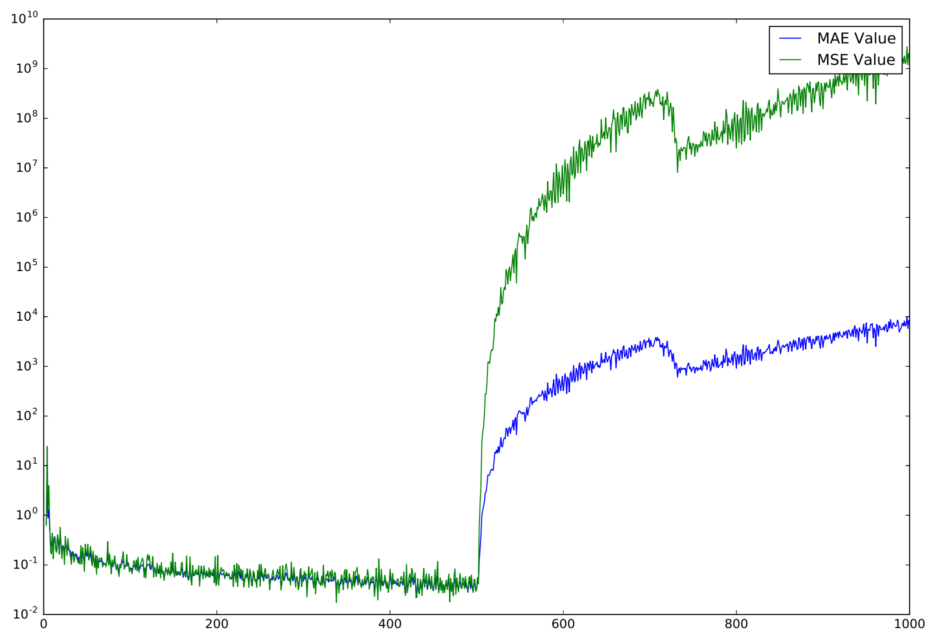

図 55　補間の分割が 500 の時の標本点の数による精度の変化

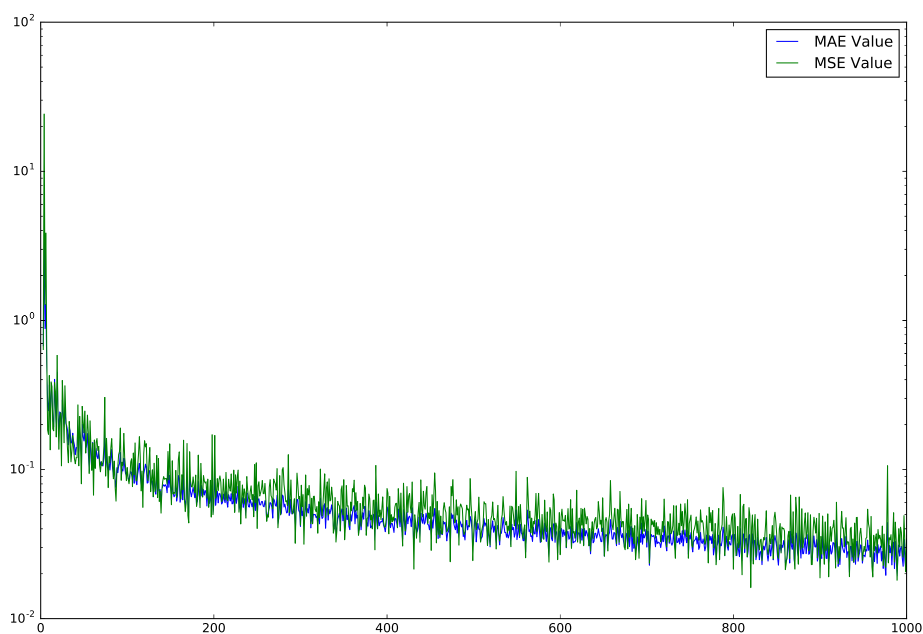

図 56　補間の分割が 1000 の時の標本点の数による精度の変化

4.6.3　ハートの形の関数

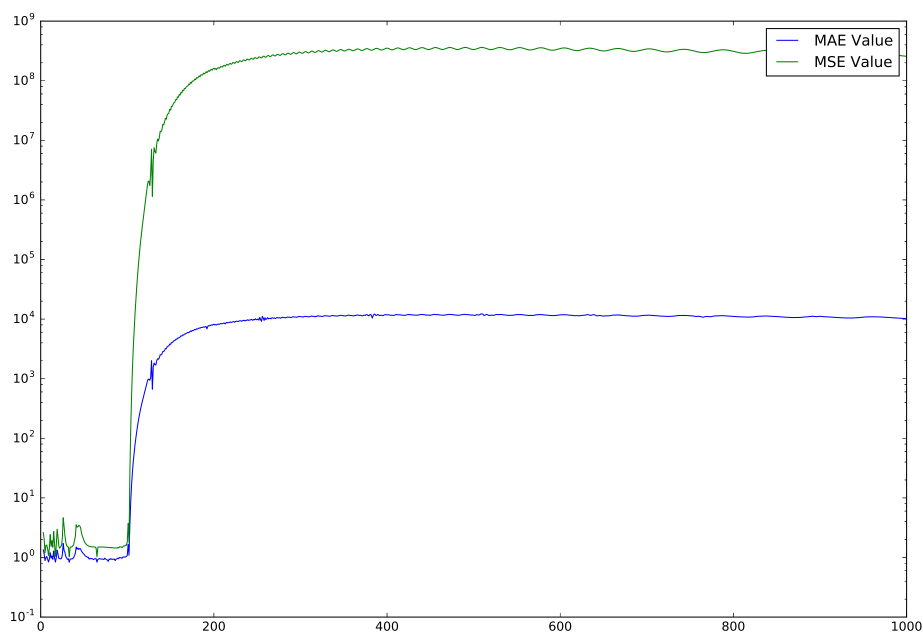

図 57　補間の分割が 100 の時の標本点の数による精度の変化

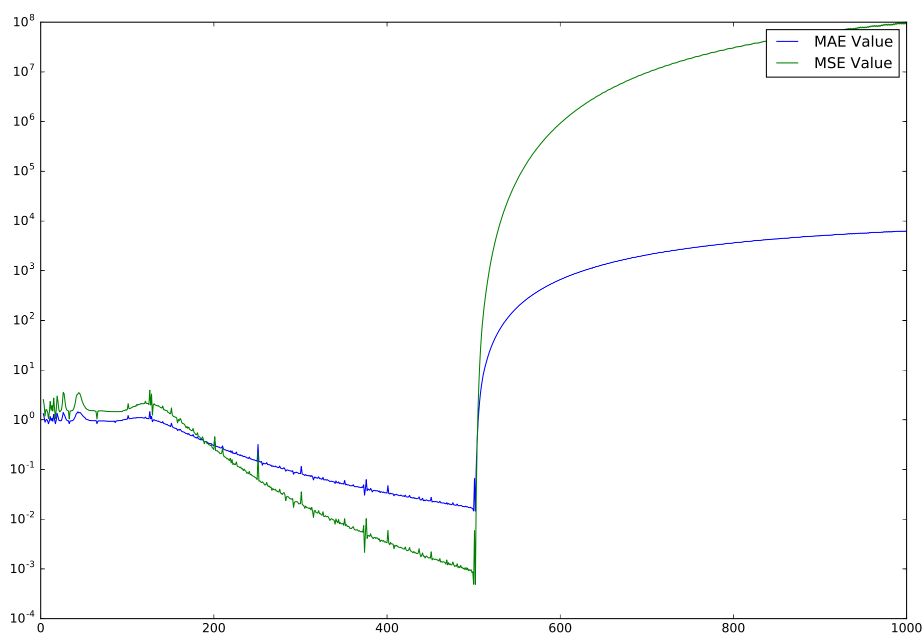

図 58　補間の分割が 500 の時の標本点の数による精度の変化

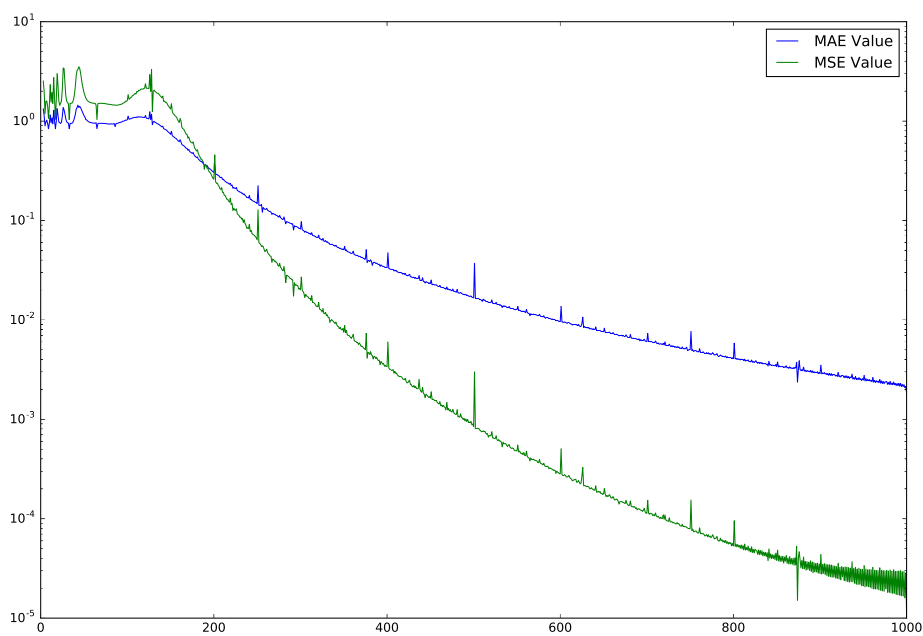

図 59　補間の分割が 1000 の時の標本点の数による精度の変化

4.7 考察

結果より, 3 つの関数はすべて標本点の数と補間を行う点の数が近づくにつれおおよそ精度がよくなり標本点の数が補間を行う点の数を超えると急激に精度が悪くなることがわかる. また, 補間を行う点の数が増えると, 最も精度のよいときの精度がよくなっている. さらに, ラグランジュ補間と比較を行うと, ラグランジュ補間では補間点は標本点上を通ることから標本点の数と補間点の数を一致させると補間誤差はなくなるが, 結果より, スプライン補間では標本点の数と補間点の数が一致しても誤差が 0 になることはないことがわかる. このことより, スプライン補間は標本点上を通るわけではないことがわかった.

4.8 「スプライン補間」と「ラグランジュ補間を改良した方法」の比較

次にスプライン補間とラグランジュ補間を改良した方法を比較し, 標本点の数による補間法の精度の差を比較する. また, ラグランジュ補間を改良した方法は $f(x) = \dfrac{1}{1+2x^2}$ の精度をよくするために改良を行い, 補間点の工夫などは汎用的ではない為

4.8.1 使用するプログラム

スプライン補間はソースコード 7 を使用し, ラグランジュ補間を改良した方法はソースコード 6 を使用する.

4.8.2 プログラム実行結果

以上を利用し, 補間を行う分割点の数を 100 にし, 標本点の数を 5 から 101 に 2 ずつ変化させ, MAE の精度比較, MSE の精度比較, MAE と MSE の積の比較をそれぞれ行う. 結果は以下のようになった.

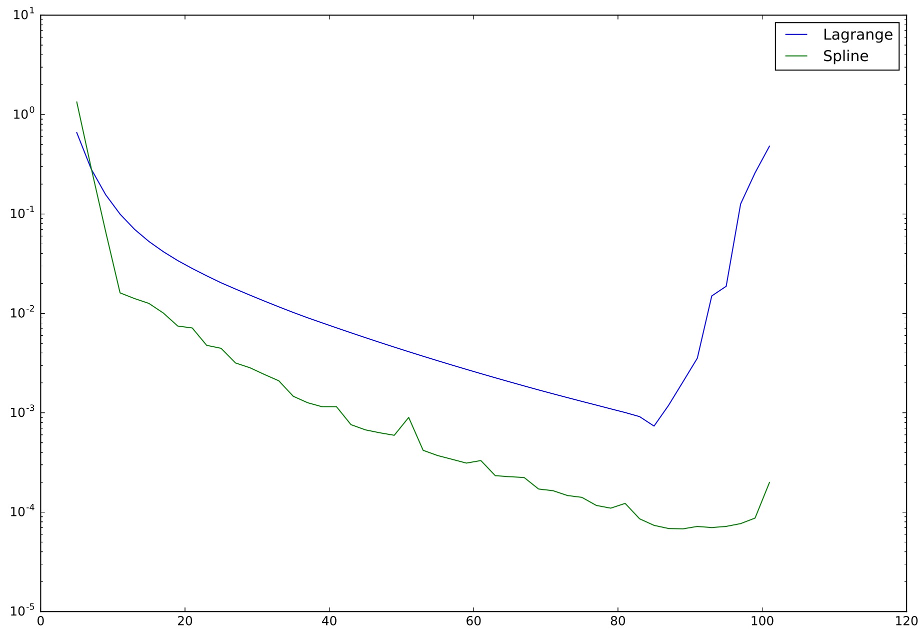

図 60　標本点の数による MAE の変化

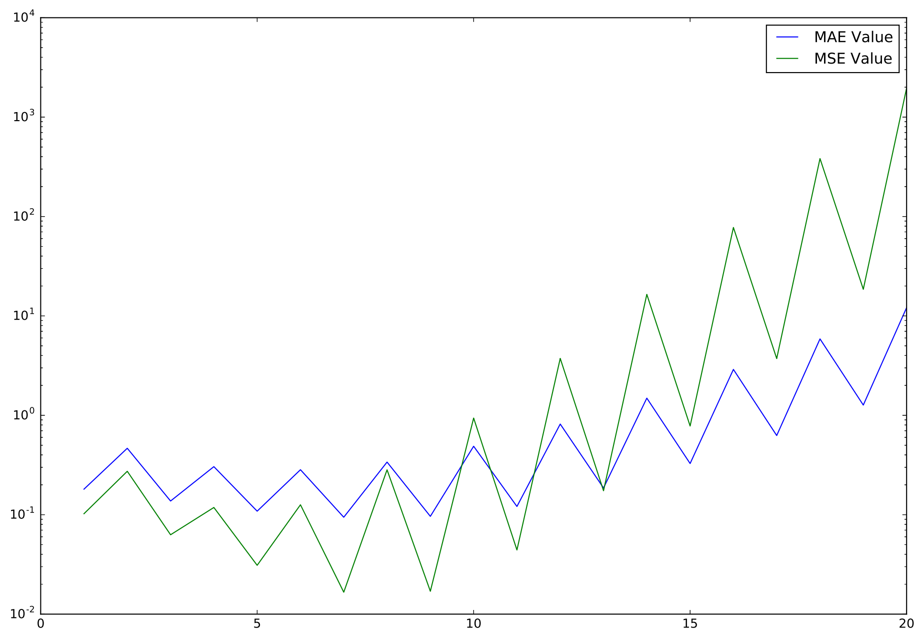

図 61　標本点の数による MSE の変化

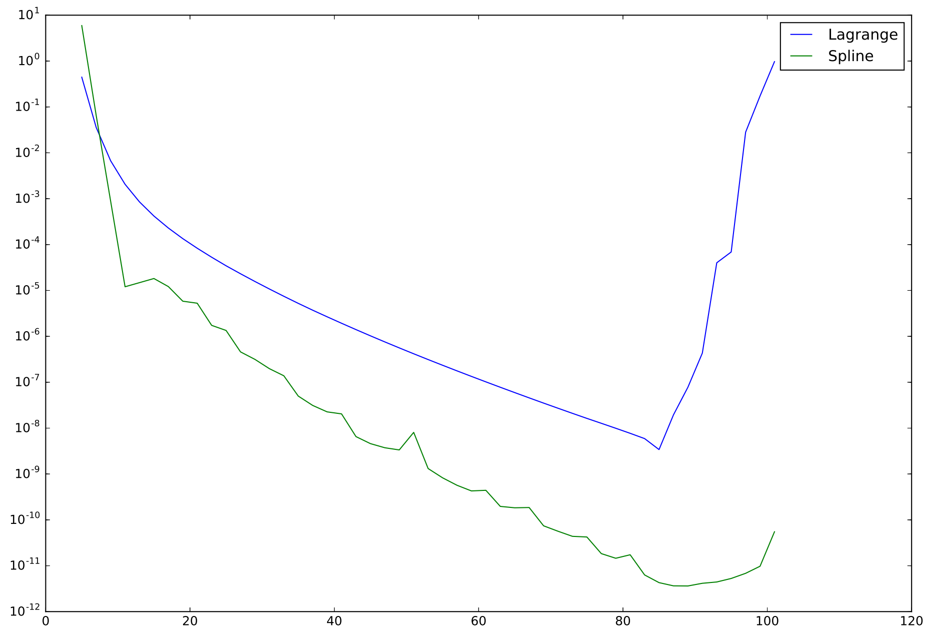

図 62　標本点の数による MAE と MSE の積の変化

同様に, 今度は補間を行う分割点の数を 1000 にし, 標本点の数を 5 から 201 に 2 ずつ変化させた時の精度の変化は以下のようになった.

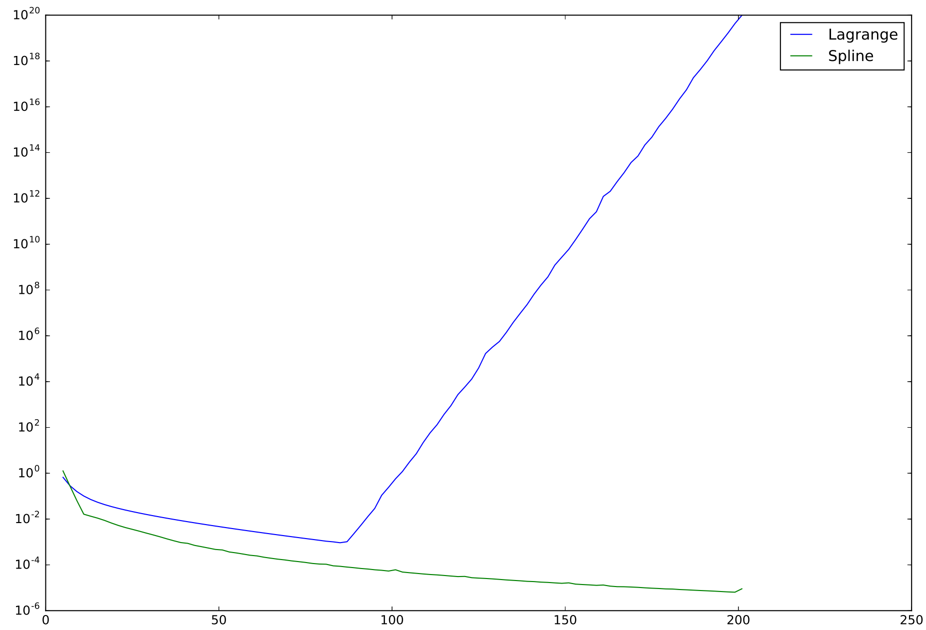

図 63　標本点の数による MAE の変化

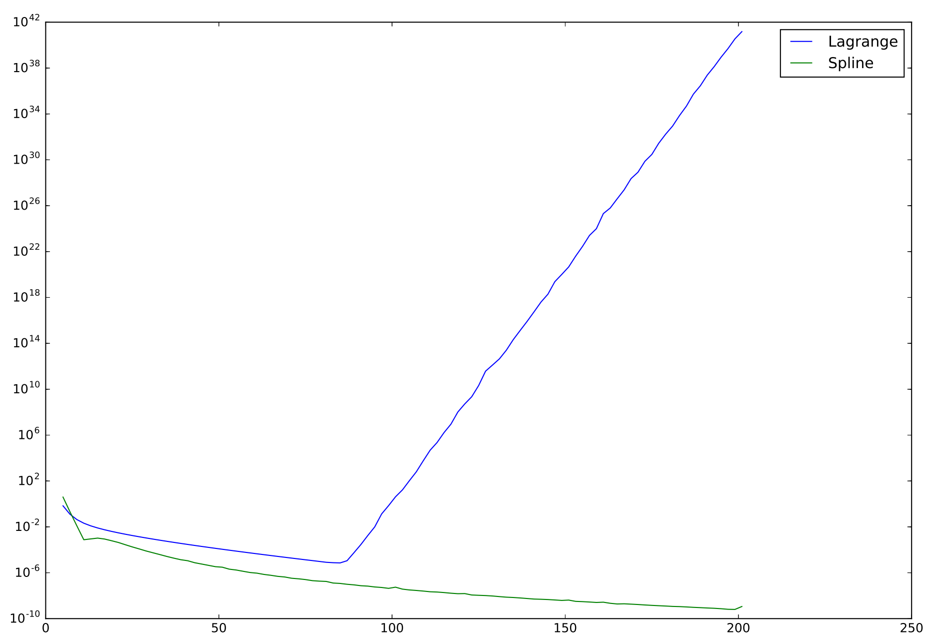

図 64　標本点の数による MSE の変化

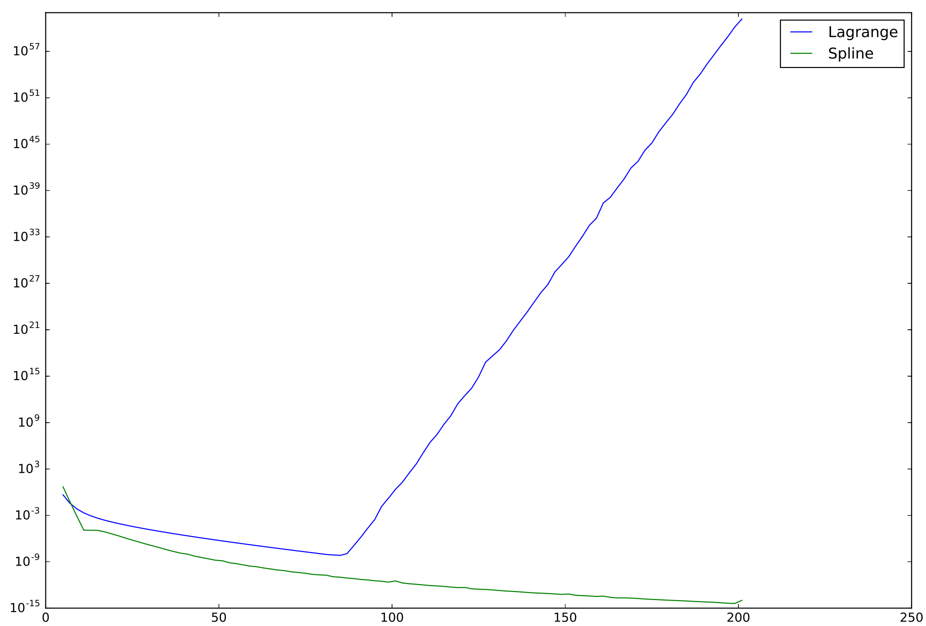

図 65　標本点の数による MAE と MSE の積の変化

4.8.3 考察

結果より, 標本点がおおよそ 10 以上ではスプライン補間の方が, ラグランジュ補間を改良した方法よりも精度が高くなった. また, ラグランジュ補間を改良した方法は補間を行う分割点に関係なくおおよそ標本点が 80 以上になると精度が急激に悪くなるが, スプライン補間は図 57, 図 58, 図 59 からわかるように, 精度の悪化が起こる境目が補間の分割の数に影響されるという特徴があることがわかった.

4.8.4　補間にかかる時間

また, 補間の分割点の数を 1000 にし, 標本点の数ごとに補間にかかる時間を計測した結果以下のようになった. 縦軸の単位は秒である (全部のグラフの縦横軸の名前入れ忘れました).

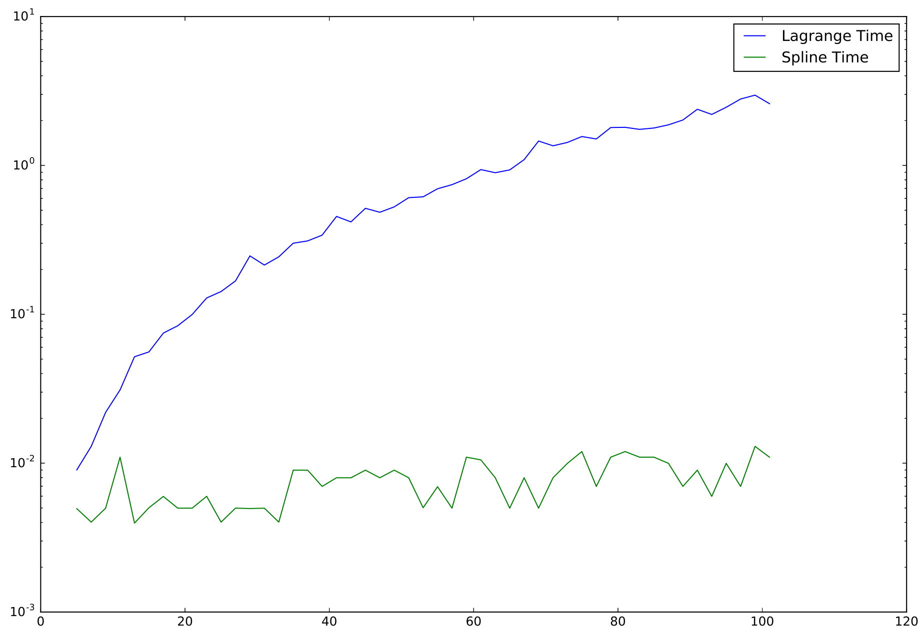

図 66　標本点の数による補間にかかる時間の変化

4.8.5　補間にかかる時間の考察

結果より, 標本点の数が増えるとラグランジュ補間は時間を必要とするが, スプライン補間はあまり関係がないということがわかる. これはラグランジュ補間の場合は, 補間を行う際に, 2 重の for 文ができているのに対して, スプライン補間の場合は 2 重の for 文がないことから処理の量に大きな差ができているからだと思われる. もう少し標本点の数を増やすとスプライン補間でも時間がかかるようになる可能性はあると思われる.

5 XGBoost での補間

3 つ目に XGBoost を用いて補間を行い, その特徴について考察していく.

5.1 XGBoost とは

XGBoost とは決定木モデルの勾配ブースティング木 (GBDT : Gradient Boosting Decision Tree) のモデルを作成するライブラリの一つで, 主に Python や R 言語で利用することができる. また, データ分析のコンペティションのプラットフォームである Kaggle, SIGNATE などではこれらの勾配ブースティング木を用いた予測モデルが上位入賞する例も多く, 非常に強力なデータ分析のライブラリである. 学習の流れは以下のようなものである.

1. 目的変数と予測値から計算される目的関数を改善するように, 決定木モデルを作成して, モデルに追加する.
2. 1. をハイパーパラメータで定めた決定木の本数だけ繰り返す.

このような繰り返しを行うため, 作成した決定木モデルは目的変数に近づいていく. また, XGBoost では過学習を防ぐために自動でクロスバリデーションを行うことができ, 最も精度の良かったモデルを採用することができる. 勾配ブースティング木のライブラリは XGBoost の他に LightGBM や CatBoost というライブラリがよく用いられる.

5.2 R 言語の実行環境

今回は時間の関係上 Python でプログラムを作成する時間が作れなかったため R 言語で行う. R 言語の環境は以下となる.

- R version 3.6.1 (2019-07-05)
- Rstudio 1.2.5001

5.3 プログラム

XGBoost を用いた補間のプログラムは以下のようになった.

ソースコード 8 XGBoost_1.R

```
1 ##########使用ライブラリ##########
2 library(MLmetrics)#評価指標計算
3 library(dplyr)
4 library(xgboost)
5 library(ggplot2)
6
7
8 #関数の作成
9 f <- function(x){
```

```
10   y = 1/(1 + 2 * x * x)
11   return(y)
12 }
13
14 M <- 100 #補完する点の分割点（補完点 - 1 個）
15 sample <- 10 #標本点の数
16
17 start <- -5 #開始点
18 end <- 5 #終了点
19 long <- end - start #範囲の長さ
20
21 data <- data.frame(x_in = seq(-5, 5, by = long / (sample - 1)))%>%
22                    mutate(f_in = f(x_in))
23
24 #目的変数作成
25 y_train <- data$f_in
26
27 #行列に変換
28 x_train <-as.matrix(data$x_in)
29
30 ###Hold Out
31 #構築データの割合
32 rate<-0.7
33
34 #構築データ数（小数の切捨て）
35 num<-as.integer(nrow(x_train)*rate)
36
37 #再現性のため乱数シードを固定
38 set.seed(17)
39
40 #sample(ベクトル，ランダムに取得する個数，復元抽出の有無)
41 row<-sample(1:nrow(x_train), num, replace=FALSE)
42
43 #構築データ
44 x_train_train<-as.matrix(x_train[row,])
45
46 #検証データ
47 x_train_test<-as.matrix(x_train[-row,])
48
49 #目的変数作成
50 y_train_train<- y_train[row]
51 y_train_test<- y_train[-row]
52
53 #パラメータの設定
54 set.seed(17)
55 param <- list(booster = 'gbtree',
```

```
56              objective = 'reg:linear',
57              eval_metric = 'mae',
58              eta = 0.1,
59              max_depth =1,
60              min_child_weight = 1,
61              subsample = 1,
62              colsample_bytree = 1
63              )
64
65 #CVによる学習数探索
66 xgbcv <- xgb.cv(param=param, data=x_train, label=y_train,
67                 nrounds=50000, #学習回数
68                 nfold=5, #CV数
69                 nthread=7, #使用するCPU数
70                 early_stopping_rounds = 10# ある回数を基準としてそこから100回以内
                      に評価関数の値が改善しなければ計算をストップ
71 )
72 #beep(sound = 3, expr = NULL)
73
74 ##モデル構築
75 set.seed(17)
76 model_xgb <- xgboost(param=param, data = x_train, label=y_train,
77                      nrounds=which.min(xgbcv$evaluation_log$test_mae_mean),
                            nthread=1, imprtance=TRUE)
78
79 #モデルを試しに適用する
80 pred_train<-predict(model_xgb, newdata = x_train_test)
81
82 #imp<-xgb.importance(model=model_xgb)
83 #print(imp)
84
85 mae_value <- RAE(pred_train, y_train_test)
86 print(mae_value)
87
88 ##########################################################################################

89
90 test_data <- data.frame(x_out = seq(-5, 5, by = long / (M - 1)))%>%
91   mutate(f_true_out = f(x_out))
92
93 #行列に変換
94 x_test <-as.matrix(test_data$x_out)
95 y_test <-as.matrix(test_data$f_true_out)
96
97 pred_test<-predict(model_xgb, newdata = x_test)
98
```

```
99  mae_value <- RAE(pred_test, y_test)
100 print(mae_value)
101
102
103 q_plot <- data.frame(
104   x = test_data$x_out,
105   complement = pred_test[1:M])
106
107 true_function <- data.frame(x =seq(-5, 5, by = 0.01))%>%
108   mutate(fx = f(x))
109
110 #可視化
111 g <- ggplot(true_function, aes(x = x, y = fx, color = "True Function"))+
112   geom_line()+
113   geom_line(q_plot, mapping = aes(x = x, y = complement, color = "XGBoost"))+
114   labs(color = "")+
115   xlab("")+
116   ylab("")
117 plot(g)
```

5.4　結果

補完する点の分割点を 100 にしてプログラムを実行した結果, 以下のようになった.

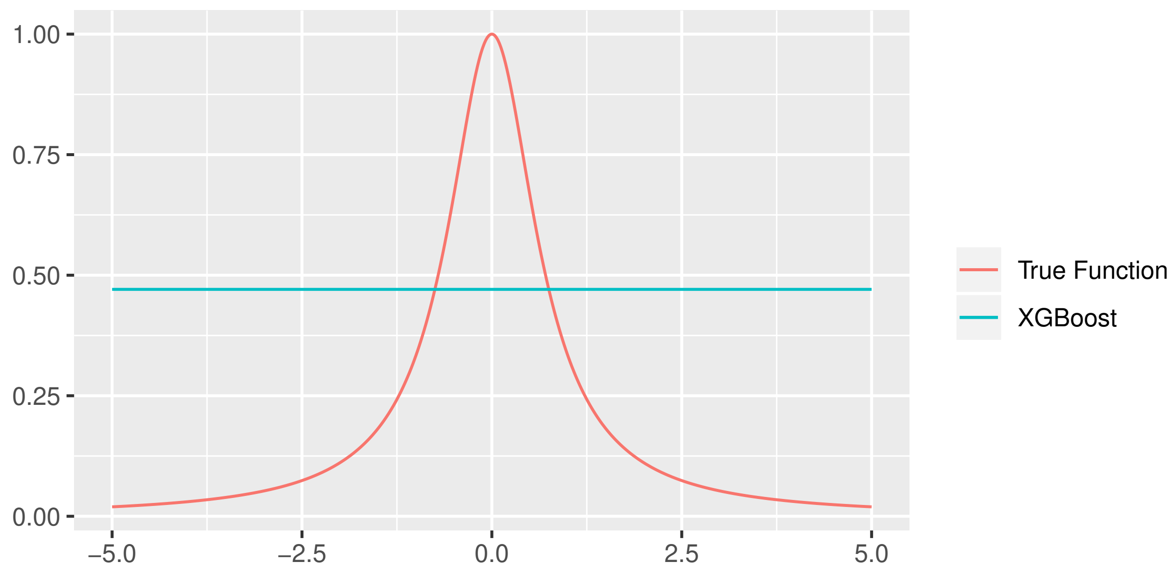

図 67　標本点の数が 10 の時の補間結果

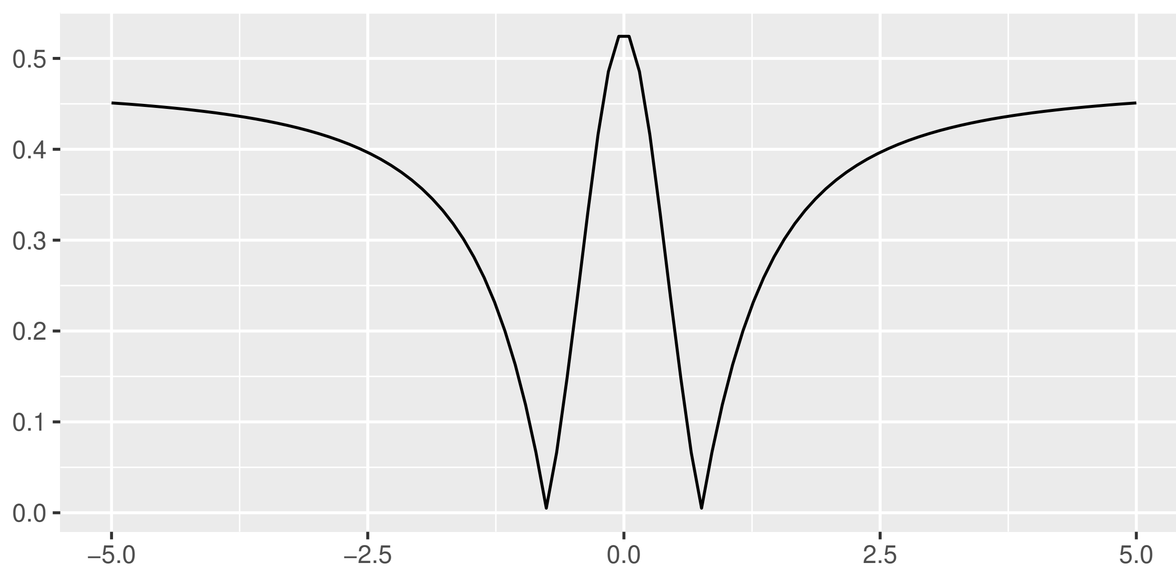

図 68　標本点の数が 10 の時の補間誤差

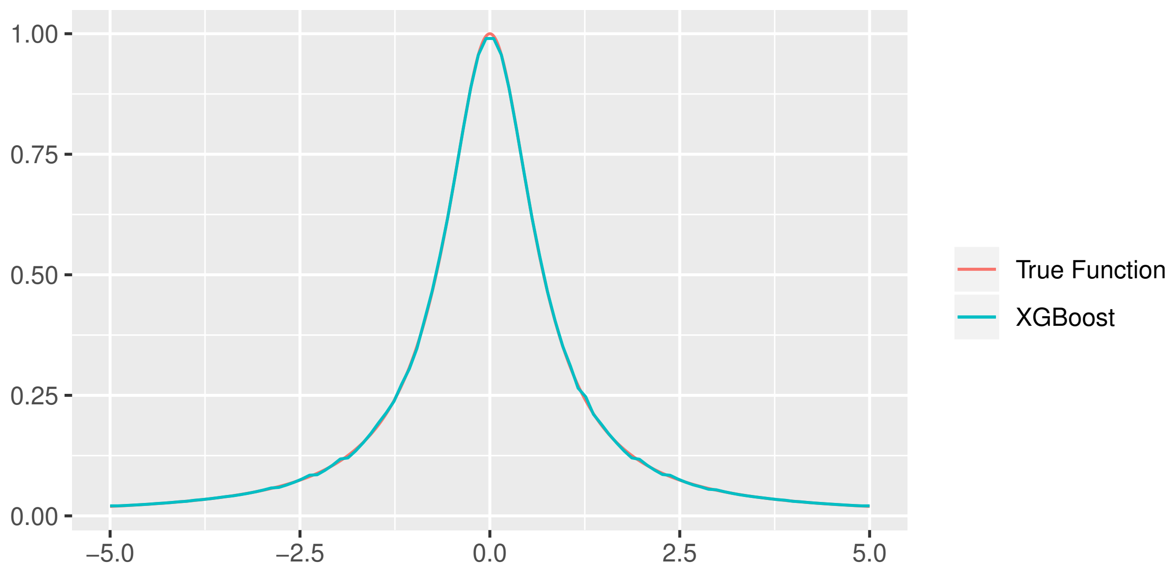

図 69　標本点の数が 100 の時の補間結果

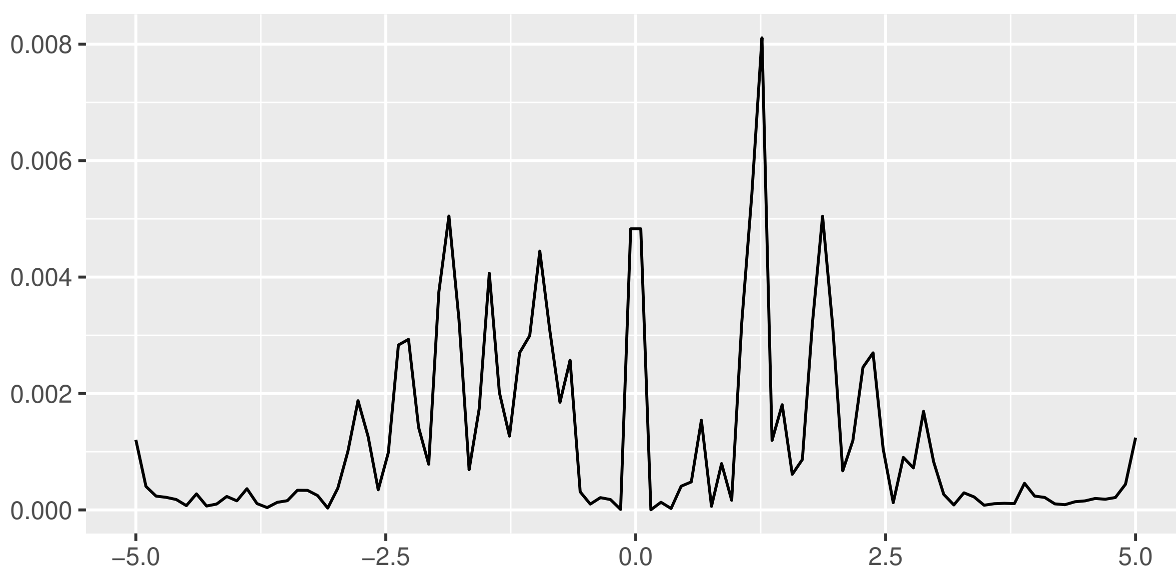

図 70　標本点の数が 100 の時の補間誤差

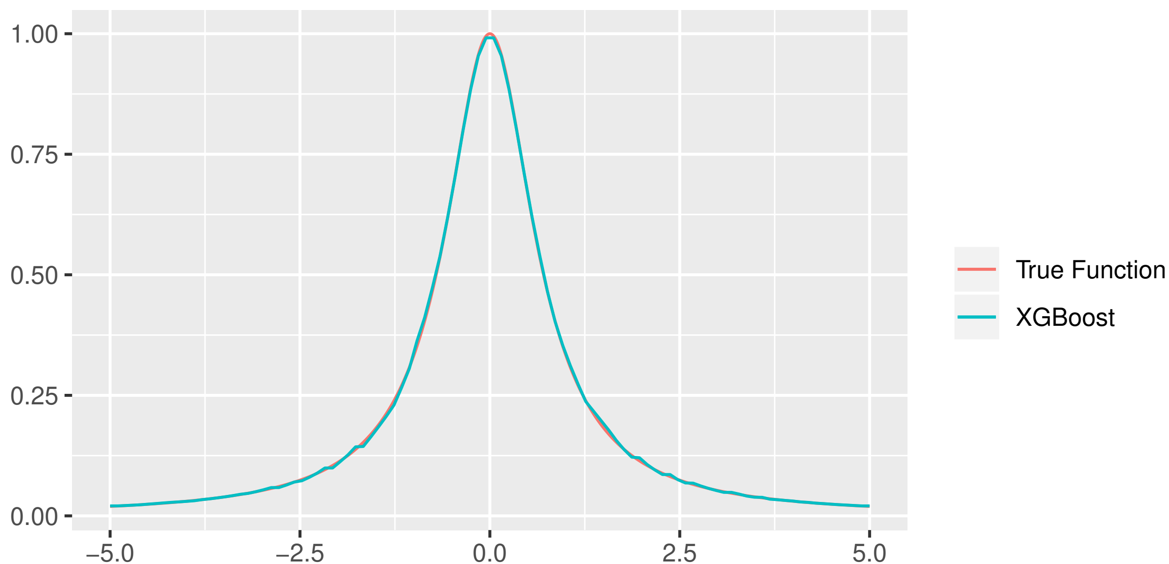

図 71　標本点の数が 1000 の時の補間結果

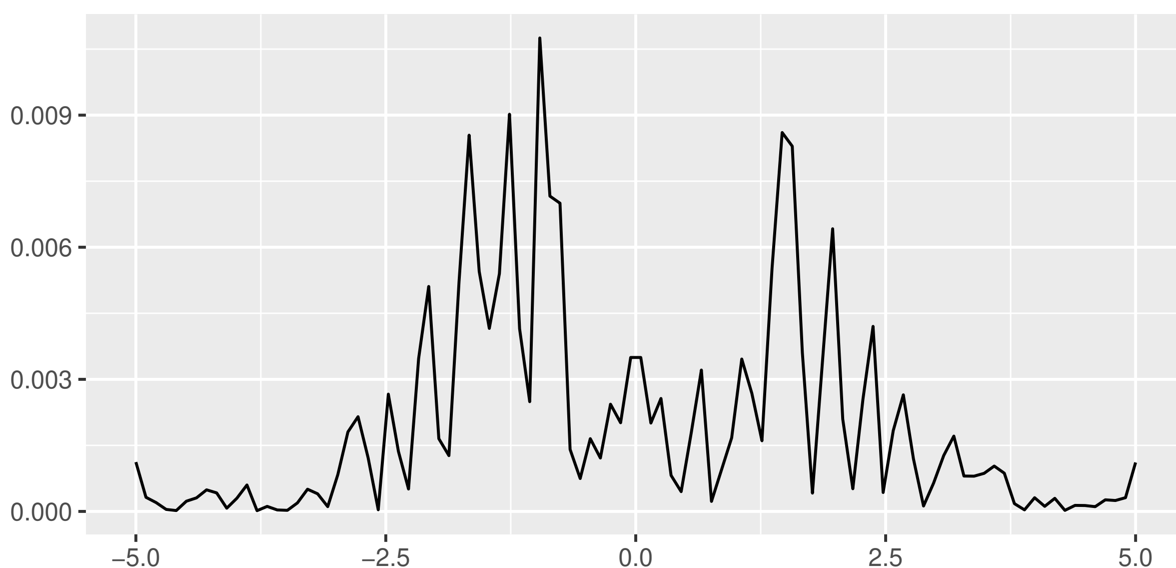

図 72　標本点の数が 1000 の時の補間誤差

5.5　考察

以上の結果より, 標本点が 10 の時は, 一定値となっていることがわかる. これは全く補間出来てるとはいえず, 改善点を探る必要があると思われる. また, スプライン補間とは異なり, 標本点が増え, 補間を行う点の数を大幅に超えても, 高い精度を維持できているように思える.

5.6 XGBoost 補間の改善

標本点の数が少ないときに全く補間が行われなかったということに関して, 精度の改善を試みる. 一般的には, 順に並んでいるデータは無作為な順に変更すると精度が改善することがあるので, それらを試す.

5.7 プログラム

それらの改善を行ったプログラムは以下のようになった.

ソースコード 9 XGBoost_2.R

```
1  ##########使用ライブラリ##########
2  library(MLmetrics)#評価指標計算
3  library(dplyr)
4  library(xgboost)
5  library(ggplot2)
6
7
8  #関数の作成
9  f <- function(x){
10   y = 1/(1 + 2 * x * x)
11   return(y)
12 }
13
14 M <- 100 #補完する点の分割点 (補完点 - 1 個)
15 sample <- 1000 #標本点の数
16
17 start <- -5 #開始点
18 end <- 5 #終了点
19 long <- end - start #範囲の長さ
20
21 data <- data.frame(x_in = seq(-5, 5, by = long / (sample - 1)))%>%
22                     mutate(f_in = f(x_in))
23
24 #データのシャッフル
25 set.seed(17)
26 data_2 <- data[sample(nrow(data)),]
27
28 #目的変数作成
29 y_train <- data_2$f_in
30
31 #行列に変換
32 x_train <-as.matrix(data_2$x_in)
33
34 ###Hold Out
```

```
35 #構築データの割合
36 rate<-0.7
37
38 #構築データ数（小数の切捨て）
39 num<-as.integer(nrow(x_train)*rate)
40
41 #再現性のため乱数シードを固定
42 set.seed(17)
43
44 #sample(ベクトル，ランダムに取得する個数，復元抽出の有無)
45 row<-sample(1:nrow(x_train), num, replace=FALSE)
46
47 #構築データ
48 x_train_train<-as.matrix(x_train[row,])
49
50 #検証データ
51 x_train_test<-as.matrix(x_train[-row,])
52
53 #目的変数作成
54 y_train_train<- y_train[row]
55 y_train_test<- y_train[-row]
56
57 #パラメータの設定
58 set.seed(17)
59 param <- list(booster = 'gbtree',
60               objective = 'reg:linear',
61               eval_metric = 'mae',
62               eta = 0.1,
63               max_depth =1,
64               min_child_weight = 1,
65               subsample = 1,
66               colsample_bytree = 1
67               )
68
69 #CV による学習数探索
70 xgbcv <- xgb.cv(param=param, data=x_train, label=y_train,
71                 nrounds=50000, #学習回数
72                 nfold=5, #CV 数
73                 nthread=7, #使用するCPU 数
74                 early_stopping_rounds = 10# ある回数を基準としてそこから 100回以内
                        に評価関数の値が改善しなければ計算をストップ
75 )
76 #beep(sound = 3, expr = NULL)
77
78 ##モデル構築
79 set.seed(17)
```

```
80 model_xgb <- xgboost(param=param, data = x_train, label=y_train,
81                      nrounds=which.min(xgbcv$evaluation_log$test_mae_mean),
                            nthread=1, imprtance=TRUE)
82
83 #モデルを試しに適用する
84 pred_train<-predict(model_xgb, newdata = x_train_test)
85
86 #imp<-xgb.importance(model=model_xgb)
87 #print(imp)
88
89 mae_value <- RAE(pred_train, y_train_test)
90 print(mae_value)
91
92 ##########################################################################################

93
94 test_data <- data.frame(x_out = seq(-5, 5, by = long / (M - 1)))%>%
95   mutate(f_true_out = f(x_out))
96
97 #行列に変換
98 x_test <-as.matrix(test_data$x_out)
99 y_test <-as.matrix(test_data$f_true_out)
100
101 pred_test<-predict(model_xgb, newdata = x_test)
102
103 mae_value <- RAE(pred_test, y_test)
104 print(mae_value)
105
106
107 q_plot <- data.frame(
108   x = test_data$x_out,
109   complement = pred_test[1:M])
110
111 true_function <- data.frame(x =seq(-5, 5, by = 0.01))%>%
112   mutate(fx = f(x))
113
114 #可視化
115 g <- ggplot(true_function, aes(x = x, y = fx, color = "True Function"))+
116   geom_line()+
117   geom_line(q_plot, mapping = aes(x = x, y = complement, color = "XGBoost"))+
118   labs(color = "")+
119   xlab("")+
120   ylab("")
121 plot(g)
```

5.8　結果

補完する点の分割点を 100 にしてプログラムを実行した結果, 以下のようになった.

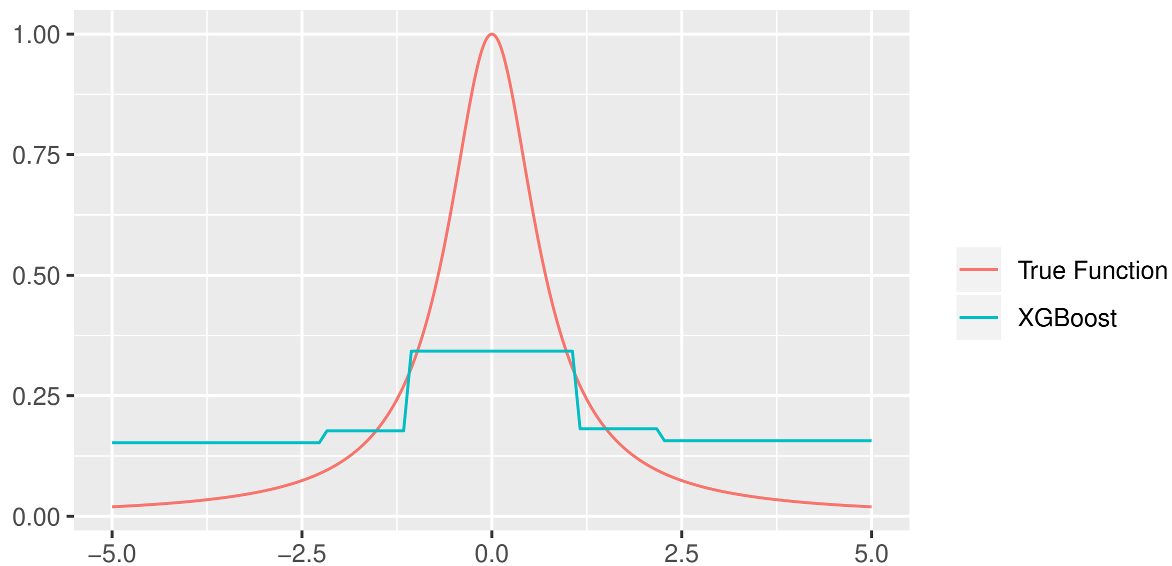

図 73　標本点の数が 10 の時の補間結果

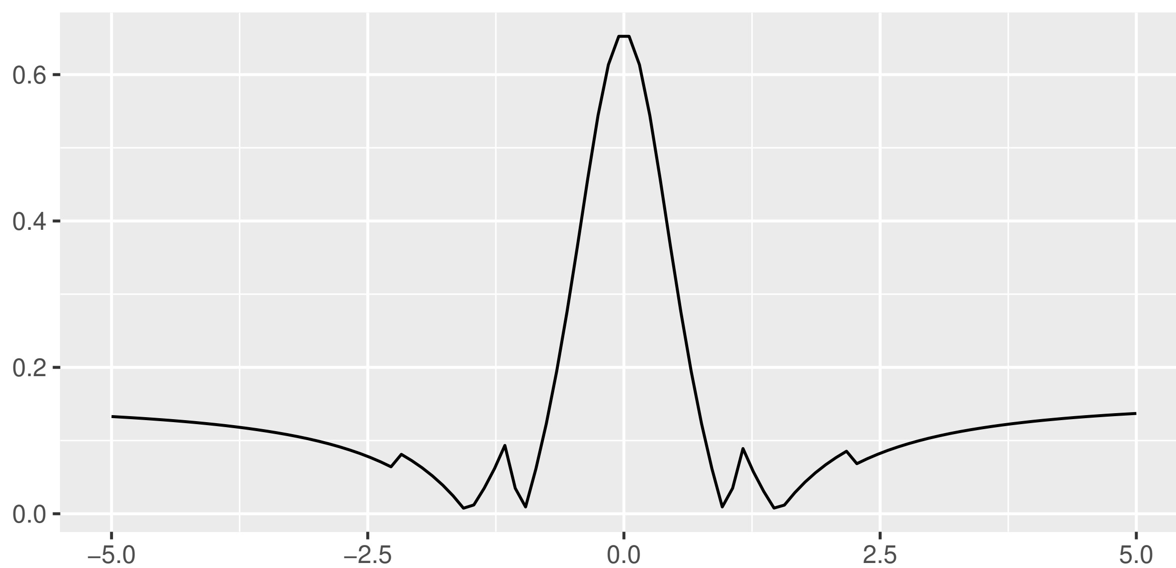

図 74　標本点の数が 10 の時の補間誤差

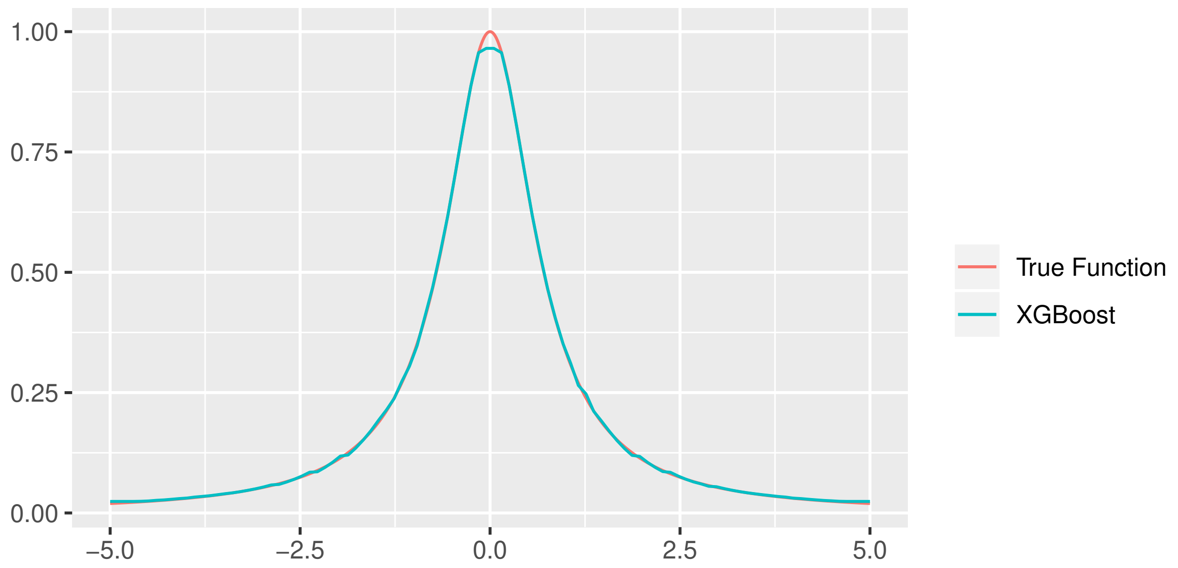

図 75　標本点の数が 100 の時の補間結果

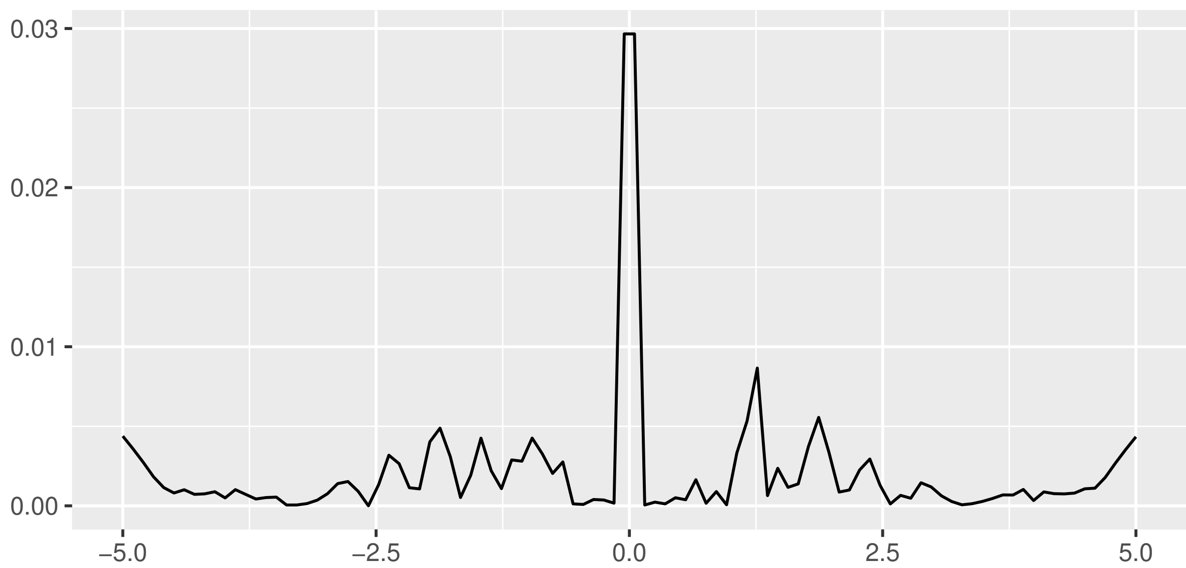

図 76　標本点の数が 100 の時の補間誤差

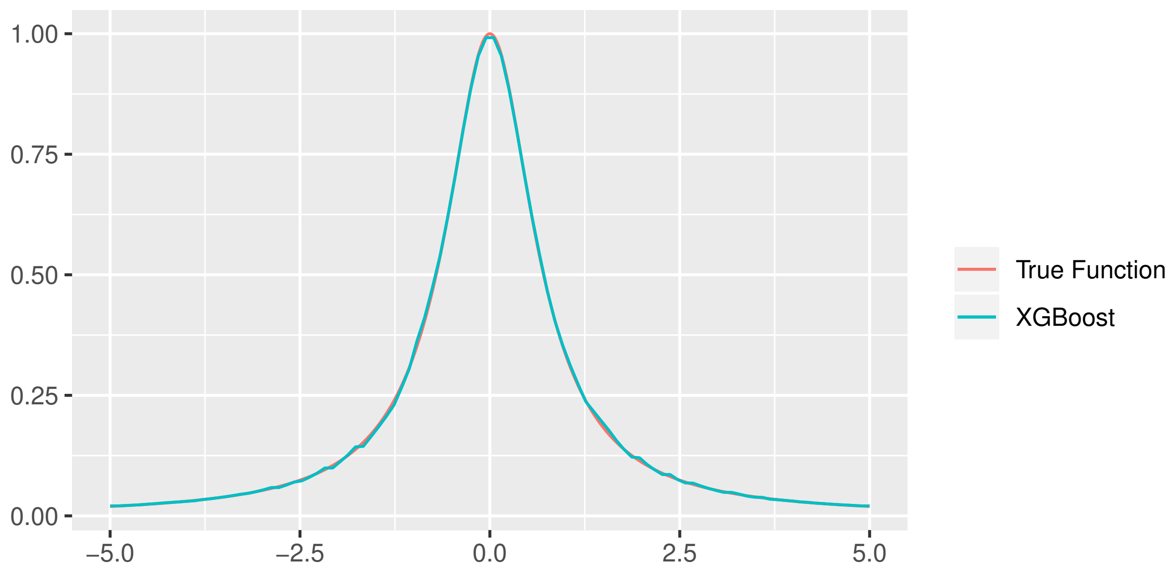

図 77　標本点の数が 1000 の時の補間結果

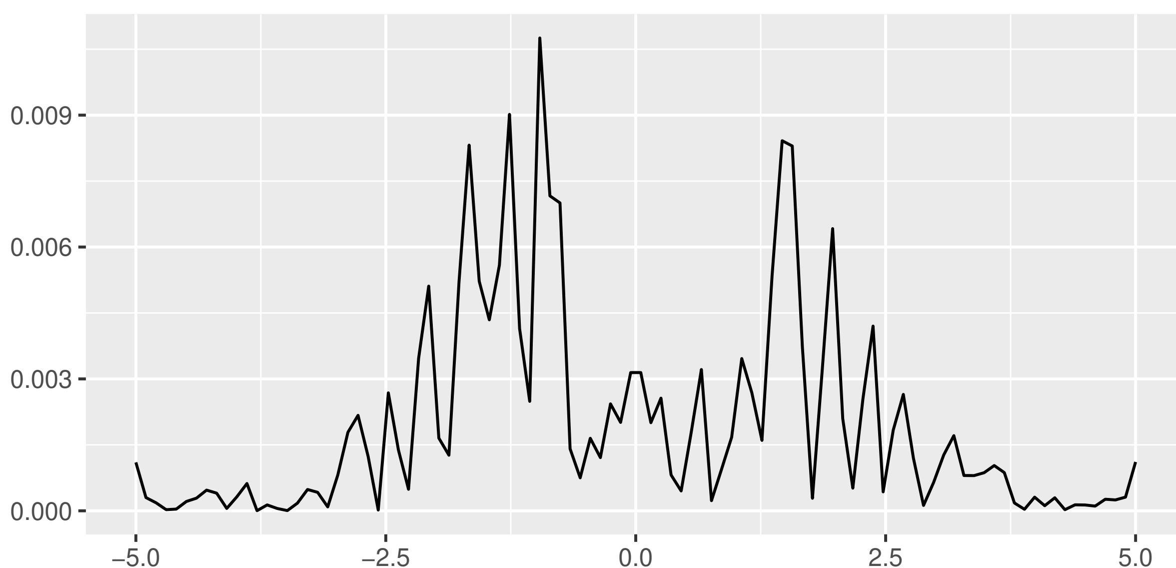

図 78　標本点の数が 1000 の時の補間誤差

5.9　考察

先ほどのプログラムと比較して, 標本点の数が 10 の時の精度が改善したように思える. 標本点の数が増えるとその影響は全くと言っていいほどないということも結果よりわかった.

5.10　スプライン補間と XGBoost での補間精度の比較

スプライン補間と XGBoost の補間の分割店を 100 にし, 標本点の数に対しての補間精度の結果は以下のようになった.

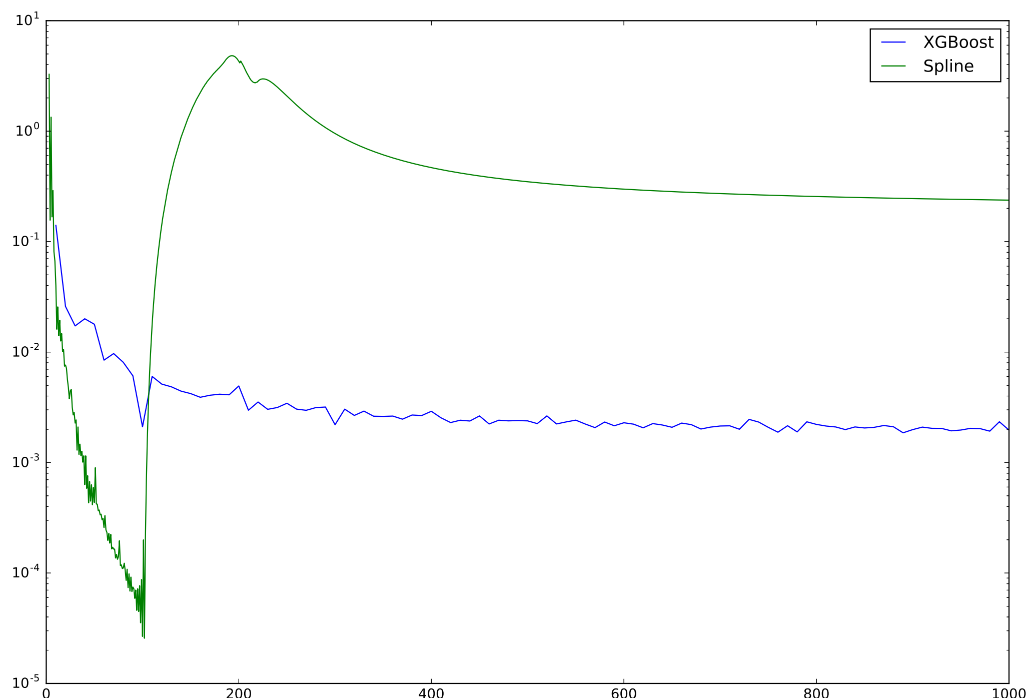

図 79　標本点の数に対する補間精度

5.11　考察

スプライン補間は標本点が 100 程度までは XGBoost よりも精度が高く, 精度の最も良かった時の XGBoost のもっともよい精度の点よりも精度がよいことがわかる. XGBoost はスプライン補間より標本点が増加した際の精度の向上は安定しているようにみえる. また, 今回は XGBoost で補間を行う際に用いたデータの種類が 1 種類であったため, 他に x 座標のデータから作成できるデータがあれば精度が向上するように思われる. また, 時間に関しては圧倒的にブースティングを行うため XGBoost は時間がかかってしまい, こういった関数の補間には向いていないことがわかる.

6　終わりに

本研究ではラグランジュ補間, スプライン補間, XGBoost を用いた補間について色々なものを比較したが, 全ての図表に縦軸横軸のラベルを入れ忘れてしまい, 単位などがわかりにくくなってしまったのが心残りである. しかし, ラグランジュ補間やスプライン補間, 補間では用いられないであろう勾

配ブースティングを用いた補間の特徴が少しわかったような気がする．本来はニューラルネットを用いた補間なども検証しようと考えていたが, 実装にてこずってしまい研究提出の期日までに間に合わせることができなかった．今度時間ができた際にでも実装を行い様々な検証を行いたい.

参考文献

[1] 福田 亜希子, 数値解析 I【第 10 回】：2020.07.16

[2] 福田 亜希子, 数値解析 II【第 3 回】：2020.10.23

[3] 福田 亜希子, 数値解析 II【第 4 回】：2020.10.20

[4] 福田 亜希子, 数値解析 II【第 5 回】：2020.10.27

[5] 門脇大輔 他, Kaggle で勝つデータ分析の技術：2019.10.22