# Fictitious Play 発表資料

大野 嵩侃

2014 年 6 月 28 日

# 目次

- Fictitious Play の説明
- コードの説明
- プログラムを用いた Fictitious Play のシミュレーション
  - Matching Pennies Game
  - Coordination Game
- まとめ

# 一般的な Fictitious Play の説明

- 一定の利得表のもとで、プレイヤー 0, プレイヤー 1 の 2 人が標準型ゲームを複数回繰り返す
- プレイヤーはそれぞれ「相手はこのような確率分布で行動する」という信念をもち、その確率分布の下で自らの期待利得が最大となるような行動をとると仮定する
- ある期における信念が、その期以前に相手が実際にとった行動の経験分布と一致する場合に、この動学モデルを *Fictitious Play* とよぶ
- 今回は厳密なそれとは少し異なるが、同様に相手の行動の経験分布が信念に大きな影響を与えるモデルを考えた

# 今回取り上げる Fictitious Play の説明

- プレイヤーは共に行動 0, 行動 1 という 2 つの選択肢をもつ
- $t$ 回目のゲームを行う直前の時点での確率分布に関する信念のうち、とくにプレイヤー $i$ の「相手はこのような確率で行動 1 をとる」という信念を $x_i(t)$ として表すことにする
- ゲームを 1 回も行っていない $t = 0$ での信念は初期信念 $x_i(0)$ として $[0, 1]$ 間の一様分布からランダムに定まる
- また、$t$ 期におけるプレイヤー $i$ の行動を $a_i(t)$ と表すことにする．たとえば
  - $t = 5$ でプレイヤー 0 が行動 1 をとったら $a_0(5) = 1$
  - $t = 8$ でプレイヤー 1 が行動 0 をとったら $a_1(8) = 0$

# 今回取り上げる Fictitious Play の説明

- 0 期から $k-1$ 期までのゲームを終えた時点での信念 $x_i(k)$ は

$$x_i(k) = \frac{x_i(0) + \sum_{j=0}^{(k-1)} a_{(1-i)}(j)}{k+1}$$

と表される
(初期信念が後の信念に影響する点が通常の fictitious play とは異なる)

- たとえば $x_0(0) = 0.5$ で、$t=0$ から $t=8$ までの 9 回でプレイヤー 1 が 7 回行動 1 を選択していたとしたら $t=9$ 時点のプレイヤー 0 の信念 $x_0(9)$ は以下のようになる

$$x_0(9) = \frac{0.5+7}{1+9} = \frac{7.5}{10} = 0.75$$

# 今回取り上げる Fictitious Play の説明

- すると、
$$x_i(k+1) = x_i(k) \times \frac{k+1}{k+2} + a_{(1-i)}(k) \times \frac{1}{k+2}$$
が成り立つことがわかる
- このことから、$x_0(t)$ を再帰的に
$$x_0(t+1) = x_0(t) + \frac{1}{t+2}(a_1(t) - x_0(t))$$
と書き表すことができる
- 上式より、直前の行動が信念に与える影響は次第に小さくなっていくため、各人の信念はいずれ何らかの値に収束してゆくのではないか、と考えられる。
- 次ページ以降ではこの $2 \times 2$ の fictitious play を表現したプログラムのコードを書き、実際にいくつかのゲームの利得表を代入してプログラムを実行してみることで、この仮説が正しいかを検証したい。

# コードの説明

- プログラムに入れた機能は次の 4 つである
- $n$ 回のゲームを行い, $t = 0, 1, \dots, n-1$ における $x_i(t)$ の推移をプロットした $t-x$ グラフを表示する
- 上記のグラフを PNG,PDF 形式で保存する
- $n$ 回のゲームからなる fictitious play を $m$ 回行い, $x_0(n-1)$ がとった値の度数分布をヒストグラムとして表示する
- 上記のヒストグラムを PNG,PDF 形式で保存する
- 次ページ以降でコードの説明を行う
- なお、インデントについては一部割愛しているので注意

# インポート

- ```
  import matplotlib.pyplot as plt
  import random
  import numpy as np
  ```
- ライブラリのインポートを行う
  - matplotlib.pyplot はグラフのプロットに使用
  - random は初期信念と行動を定めるのに使用
  - numpy は行列計算に使用
  ```

# FP クラスと init メソッド (1/2)

- `class FP:`

```
    def __init__(self, profits):
        self.pro = profits
        self.cu_xs = [0, 0]
        self.x0s = []
        self.x1s = []
```

- いくつかのメソッドをもつクラスを FP として定義した
  FP を使う際には `f = FP(profits)` のように打つ必要がある
- `profits` は利得表を表す行列である
  詳細は次ページで述べる
- init メソッド中でインスタンス変数をいくつか定義している
  これはクラス内の各メソッド中で共有される

# FP クラスと init メソッド (2/2)

- ```
  class FP:

      def __init__(self, profits):
          self.pro = profits
          self.cu_xs = [0, 0]
          self.x0s = []
          self.x1s = []
  ```

- インスタンス変数の説明
  - self.pro は profits と同じであり、たとえば利得行列

    $$\begin{pmatrix} (a,b) & (c,d) \\ (e,f) & (g,h) \end{pmatrix}$$

    は [[[a,b],[c,d]],[[e,f],[g,h]]] で表す
  - self.cu_xs は現在の $(x_0, x_1)$ を表すリスト
  - self.x0s, self.x1s は $t = 0, 1, 2, \ldots$ での $x_0, x_1$ をそれぞれ左から並べたリスト
  ```

# oneplay メソッド (1/5)

- ```
  def oneplay(self, ts_length):
          pro_cal = (np.transpose(self.pro)[0],
                     np.transpose(np.transpose(self.pro)[1]))
          self.x0s = []
          self.x1s = []
          self.cu_xs = [random.uniform(0, 1), random.uniform(0, 1)]
          cu_es = [[0, 0], [0, 0]]
  ```
- oneplay は ts_length の値の回数だけゲームを行い、$x_0(t), x_1(t)$ の値の推移を記録するメソッド
  最初に x0s，x1s を空にすることでこの oneplay を複数回行えるようにしている
  (これがないと何回も繰り返した時に要素が無尽蔵に増える)
- pro_cal に関しては次ページで詳しく述べる
- cu_xs はランダムに定まる初期信念 $x_0(0), x_1(0)$ の値になる
- cu_es はその時点で各プレイヤーが各行動をとったときの期待利得を表し、ここではリストの形だけを定めている
  ```

# oneplay メソッド (2/5)

- ```python
  def oneplay(self, ts_length):
           pro_cal = (np.transpose(np.transpose(self.pro)[0]),
                      np.transpose(self.pro)[1])
          self.x0s = []
          self.x1s = []
          self.cu_xs = [random.uniform(0, 1), random.uniform(0, 1)]
          cu_es = [[0, 0], [0, 0]]
  ```

- pro_cal は利得表をそのまま書き写したのに近い profits を計算しやすいよう転置を用いて書き換えたもの

  $$\begin{pmatrix} (a,b) & (c,d) \\ (e,f) & (g,h) \end{pmatrix}$$

  を表すリスト [[[a,b],[c,d]],[[e,f],[g,h]]] に対してこの操作を行うと

  [array[[a,c],[e,g]],array[[b,f],[d,h]] となる。
  すなわち

  $$\begin{pmatrix} a & c \\ e & g \end{pmatrix}, \begin{pmatrix} b & f \\ d & h \end{pmatrix}$$

  という形に変形される
  ```

# oneplay メソッド (3/5)

- ```
  for i in range(ts_length):
      self.x0s.append(self.cu_xs[0])
      self.x1s.append(self.cu_xs[1])
      exp = ((1-self.cu_xs[0], self.cu_xs[0]),
             (1-self.cu_xs[1], self.cu_xs[1]))
      cu_es[0] = np.dot(pro_cal[0], exp[0])
      cu_es[1] = np.dot(pro_cal[1], exp[1])
      cu_as = [0, 0]
  ```
- 1 回毎のゲームを ts_length の回数繰り返す
- まず x0s に cu_xs[0],x1s に cu_xs[1] を加えている
  n 回これを繰り返すと $t = 0, 1, \ldots, n-1$ における
  $x_0(t), x_1(t)$ の値がそれぞれ順番に記録される
- exp はプレイヤー 2 人が考える相手の行動の確率分布を行列の形で表したもの
  いま、cu_xs は $[x_0(t), x_1(t)]$ となっているので、exp は以下のようになる

$$\begin{pmatrix} 1 - x_0(t) & x_0(t) \\ 1 - x_1(t) & x_1(t) \end{pmatrix}$$
  ```

# oneplay メソッド (4/5)

- ```
  (for i in range(ts_length):)
        cu_es[0] = np.dot(pro_cal[0], exp[0])
        cu_es[1] = np.dot(pro_cal[1], exp[1])
        cu_as = [0, 0]
  ```
- 期待利得を表す cu_es の値を計算する
  np.dot は行列の積を表すので、ここでやっている計算は

  $$\begin{pmatrix} a & c \\ e & g \end{pmatrix} \cdot \begin{pmatrix} 1 - x_0(t) \\ x_0(t) \end{pmatrix}, \begin{pmatrix} b & f \\ d & h \end{pmatrix} \cdot \begin{pmatrix} 1 - x_1(t) \\ x_1(t) \end{pmatrix}$$

  の 2 つである
- cu_as はある時点での実際の行動を表すリストで、ここでは形だけを定めている
  ```

# oneplay メソッド (5/5)

- ```
  (for i in range(ts_length):)
      for j in range(2):
          if cu_es[j][0] == cu_es[j][1]:
              cu_as[j] = random.randint(0, 1)
          else:
              cu_as[j] = cu_es[j].argmax()
      for k in range(2):
          self.cu_xs[k] = (self.cu_xs[k]*(i+1)+cu_as[1-k])/(i+2)
  ```
- for j 以下で両プレイヤーの行動を定める
  - 行動 0 と行動 1 の期待利得が同じならランダムに定める
  - 異なる場合は argmax() によって期待利得が大きい方の行動を選択する
    たとえば cu_xs[0] = [3,-2] の場合に argmax() を使うと最大値 3 のリスト内の順番である 0 が返ってくる
- for k 以下で信念をアップデートする
  - 計算は Fictitious Play の説明通り
  ```

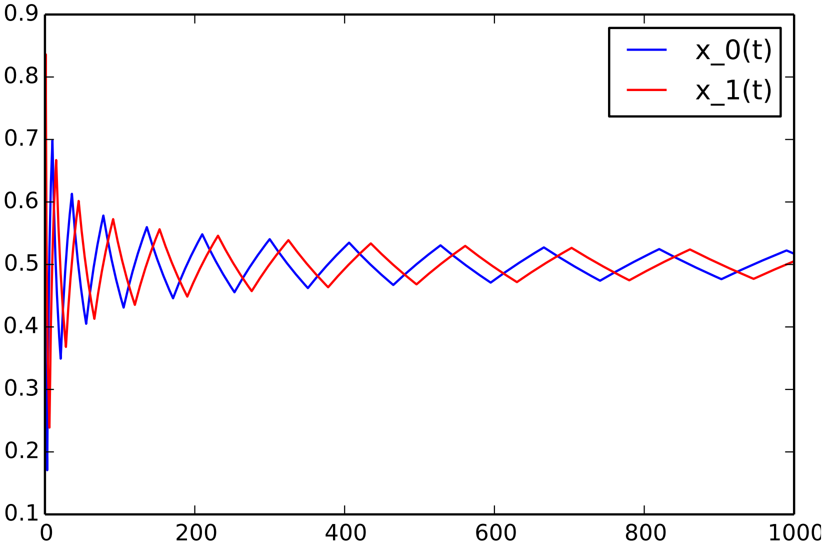
# playplot メソッド (1/2)

- ```
  def playplot(self, ts_length):
      self.oneplay(ts_length)
      plt.plot(self.x0s, 'b-', label='x_0(t)')
      plt.plot(self.x1s, 'r-', label='x_1(t)')
      plt.legend()
      plt.show()
  ```
- `oneplay` メソッドを実行して記録された $x_0(t), x_1(t)$ の推移をプロットするメソッド
- `playplot(1000)` のように入力すると実行される
  この場合は $t = 0, 1, \ldots, 999$ について記録されている
- たとえば次のような利得行列でこのメソッドを実行する

$$\begin{pmatrix} (1,-1) & (-1,1) \\ (-1,1) & (1,-1) \end{pmatrix}$$

すると次ページのようなグラフが得られる
  ```

# playplot メソッド (2/2)



# playsave メソッド

- ```
  def playsave(self, ts_length, name):
      self.oneplay(ts_length)
      plt.plot(self.x0s, 'b-', label='x_0(t)')
      plt.plot(self.x1s, 'r-', label='x_1(t)')
      plt.legend()
      plt.savefig(str(name)+'.png', bbox_inches='tight', pad_inches=0)
      plt.savefig(str(name)+'.pdf', bbox_inches='tight', pad_inches=0)
      plt.close()
  ```
- 前ページで表示したグラフを表示することなく PNG と PDF で保存するメソッド
- `playsave(1000,'fictplay')` のように入力して実行
  この場合は $t = 0, 1, \ldots, 999$ についての推移を表すグラフを"fictplay.png" および"fictplay.pdf" として保存している
- 保存するグラフを前もって見ることができないため、推移が何種類か考えられるようなものを漏らさず保存するのには不向き
  ```

# histogram メソッド (1/2)

- ```
  def histogram(self, n, ts_length):
      last_x0s = []
      for j in range(n):
          self.oneplay(ts_length)
          last_x0s.append(self.cu_xs[0])
  ```
- oneplay メソッドを何回も実行し、各回における最終的な $x_0$ の値を記録するメソッド
- histogram(100,1000) のように入力して実行
  この場合 $x_0(1000)$ を 100 回分記録することになる
- リストである last.x0s に記録が行われる
  oneplay が終わるごとにその時点での cu_xs[0] が追加される
  ```

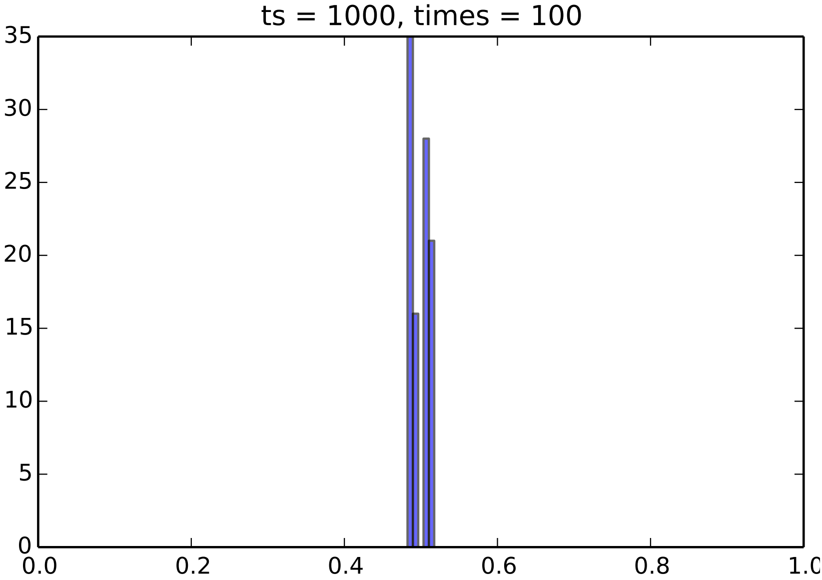
# histogram メソッド (2/2)

- ```
  ax = plt.subplot(111)
  ax.hist(last_x0s, alpha=0.6, bins=5)
  ax.set_xlim(xmin=0, xmax=1)
  t = 'ts = '+str(ts_length)+', times = '+str(n)
  ax.set_title(t)
  ```
- `last_x0s` からヒストグラムを作り、表示せずにバックグラウンドでプロットしている
- 確率分布としてわかりやすいよう $x$ の範囲は 0 から 1 まで
- `last_x0s` の最大値から最小値を引いた値を 5 等分したものがヒストグラムの 1 本 1 本の幅となる
  ```

# histplot メソッド (1/2)

- ```
  def histplot(self, n, ts_length):
      self.histogram(n, ts_length)
      plt.show()
  ```
- `histogram` メソッドでバックグラウンドにプロットしたものを表示するメソッド
- `histplot(100,1000)` のように入力して実行する
  このとき表示されるグラフは 100 回分の $x_0(1000)$ の値の度数を表すヒストグラムである
  これによって次ページのようなグラフが表示される
  ```

# histplot メソッド (2/2)



# histsave メソッド

- ```
  def histsave(self, n, ts_length, name):
      self.histogram(n, ts_length)
      plt.savefig(str(name)+'.png', bbox_inches='tight', pad_inches=0)
      plt.savefig(str(name)+'.pdf', bbox_inches='tight', pad_inches=0)
      plt.close()
  ```
- 前ページで表示したグラフを表示することなく PNG と PDF で保存するメソッド
- `histsave(100,1000,'ficthist')` のように入力して実行
  この場合は 100 回分の $x_0(1000)$ の値についてのヒストグラムを"ficthist.png" および"ficthist.pdf" として保存している
  ```

# 具体的なゲームにおけるプログラムの実行

- ここまでで説明したコードを用いて、特定の利得表で表されるゲームの様子を実際に求めてみる
- 今回行うのは次の 2 つのゲームである
  - Matching Pennies Game
  - Coordination Game

# Matching Pennies

- Matching Pennies Game は以下のような利得表をもつ単純なゲームである

| | 行動 0 | 行動 1 |
|---|---|---|
| 行動 0 | 1,-1 | -1,1 |
| 行動 1 | -1,1 | 1,-1 |

- このゲームに純粋戦略ナッシュ均衡は存在しない
- また、混合戦略ナッシュ均衡は両者ともに $(0.5, 0.5)$(のみ) である
- このため、$x_i(t)$ が収束するのであればそれは $0.5$ と予測される
- 実際にプログラムを実行し、描画したグラフおよびヒストグラムはコードの説明で用いたものに等しい
- 図より、たしかにこのゲームでの fictitious play では両者の混合戦略は $(0.5, 0.5)$ に収束するとわかる

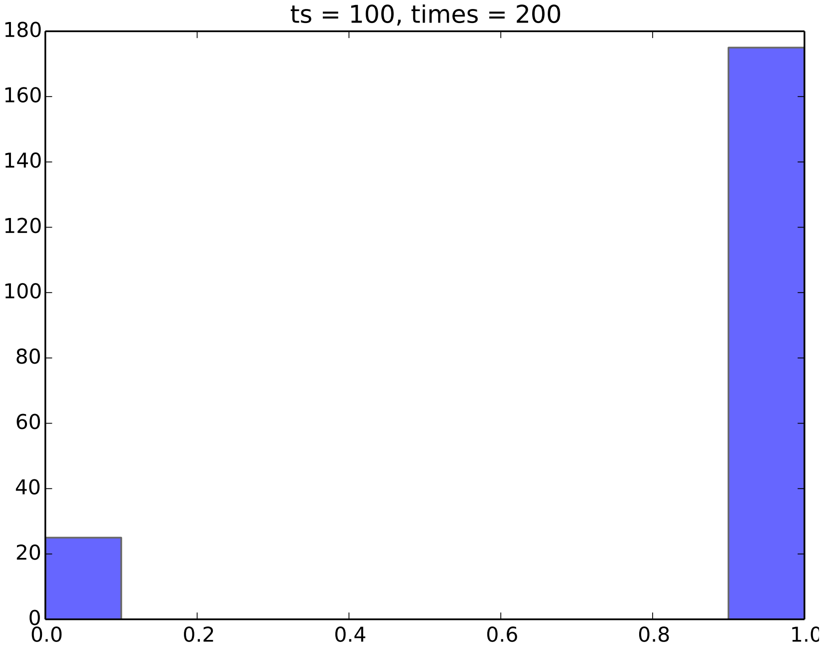
# Coordination

- Coordination Game は以下のような利得表をもつゲームである

| | 行動 0 | 行動 1 |
|---|---|---|
| 行動 0 | 4,4 | 0,3 |
| 行動 1 | 3,0 | 2,2 |

- このゲームの純粋戦略ナッシュ均衡は $(a_1, a_2) = (0, 0), (1, 1)$ の 2 つである
- また、混合戦略ナッシュ均衡は両者ともに $(\frac{2}{3}, \frac{1}{3})$ のみである
- この 3 つのナッシュ均衡点への収束が何らかの比率で実現すると考えられるが……?
- 次ページに coordination game で $t = 100$ の fictitious play を 200 回行ったヒストグラムを示す

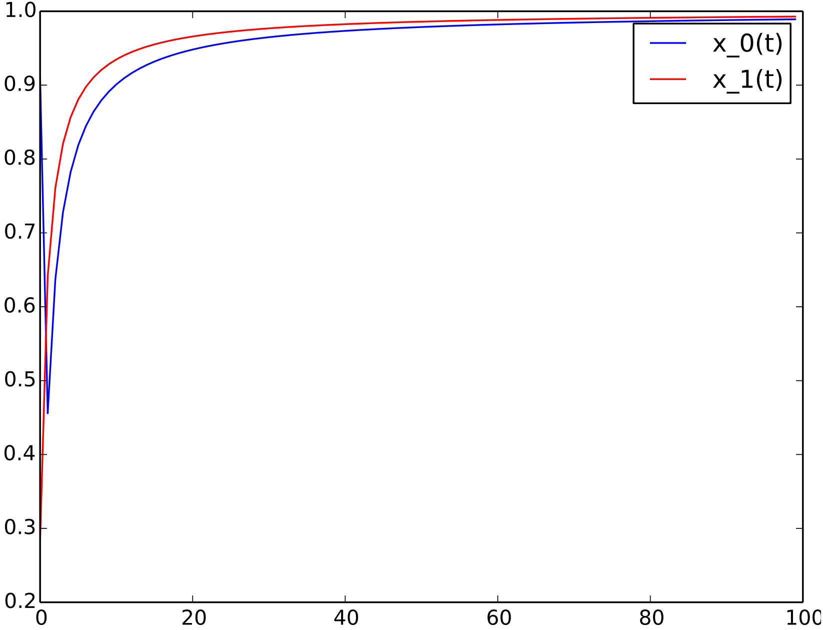
# Coordination



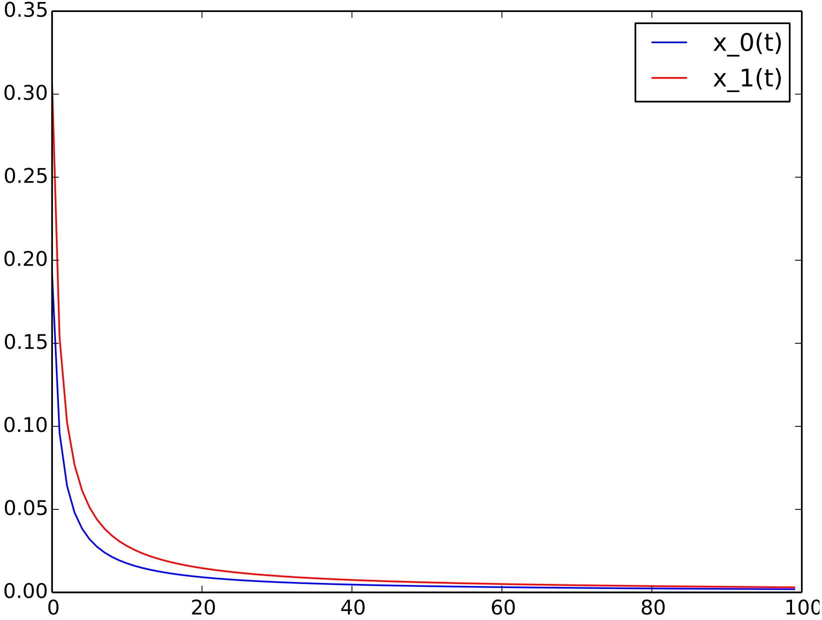
# Coordination

- 前ページのヒストグラムより、3 つのナッシュ均衡点のうち混合戦略ナッシュ均衡は実現していないことがわかる
- これはこのゲームが「相手と同じ戦略を取るのがよい」という性質を持つゲームであり、どちらかの $x_i(t)$ が 0 あるいは 1 に一定以上近づいた時点でもう片方もその行動を優先するようになるためであると考えられる
- $x_i(t)$ が 1 に収束したパターンと 0 に収束したパターンのグラフを次に記す
  いずれも matching pennies に比べ収束が速いことがわかる

# Coordination



# Coordination



# まとめ

- fictitious play ではナッシュ均衡点に収束してゆく
- だが、ナッシュ均衡点の全てが収束先となるわけではない

# よくわかっていない点・今後の課題

- coordination game で混合戦略ナッシュ均衡が実現しないことの証明
- np.transpose() を利得行列に使った際に行われている処理
  おそらく利得行列は 3 次元配列の行列として扱われている
  3 次元配列での転置とは何をどうしているのか……?
- ヒストグラムをプロットする際、現在は値が出た区間を 5 等分しているが、これを結果にかかわらず $[0, 1]$ の区間を 10 等分するように変えたい