Fictitious Play 発表資料

大野 嵩侃

2014 年 6 月 28 日

はじめに

- 項目 1
- 項目 2
  - 1 階層下の項目 1
  - 1 階層下の項目 2
- このページの最後の項目

次のスライド

- Ficititious play の説明 1
  利得表の中から、プレイヤー 0 は行を自由に選べ、プレイヤー 1 は列を自由に選べると考えて良い。
  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」という信念に従って、自らの期待利得が最大となるような行動をとる
  とくに「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 回も行っていない時の信念は初期信念 $x_i(0)$ として $[0,1]$ 間の一様分布からランダムに定まる
  k 回ゲームを終えた時点での信念 $x_i(k)$ は

$$x_i(k) = \frac{x_i(0) + (\text{実際に相手が行動 1 をとった回数})}{k+1}$$

  と表される
  たとえば $x_0(0) = 0.5$ で、9 回目までにプレイヤー 1 が 7 回行動 1 を選択していたとしたらこの時点でのプレイヤー 0 の信念 $x_0(9)$ は

次のスライド

- Ficititious play の説明 1
  利得表の中から、プレイヤー 0 は行を自由に選べ、プレイヤー 1 は列を自由に選べると考えて良い。
  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」という信念に従って、自らの期待利得が最大となるような行動をとる
  とくに「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 回も行っていない時の信念は初期信念 $x_i(0)$ として $[0, 1]$ 間の一様分布からランダムに定まる
  k 回ゲームを終えた時点での信念 $x_i(k)$ は

$$x_i(k) = \frac{x_i(0) + (\text{実際に相手が行動 1 をとった回数})}{k+1}$$

  と表される
  たとえば $x_0(0) = 0.5$ で、9 回目までにプレイヤー 1 が 7 回行動 1 を選択していたとしたらこの時点でのプレイヤー 0 の信念 $x_0(9)$ は

次のスライド

- Ficititious play の説明 1
  利得表の中から、プレイヤー 0 は行を自由に選べ、プレイヤー 1 は列を自由に選べると考えて良い。
  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」という信念に従って、自らの期待利得が最大となるような行動をとる
  とくに「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 回も行っていない時の信念は初期信念 $x_i(0)$ として $[0, 1]$ 間の一様分布からランダムに定まる
  k 回ゲームを終えた時点での信念 $x_i(k)$ は

$$x_i(k) = \frac{x_i(0) + (\text{実際に相手が行動 1 をとった回数})}{k+1}$$

  と表される
  たとえば $x_0(0) = 0.5$ で、9 回目までにプレイヤー 1 が 7 回行動 1 を選択していたとしたらこの時点でのプレイヤー 0 の信念 $x_0(9)$ は

次のスライド

- Fictitious play の説明 1
  利得表の中から、プレイヤー 0 は行を自由に選べ、プレイヤー 1 は列を自由に選べると考えて良い。
  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」という信念に従って、自らの期待利得が最大となるような行動をとる
  とくに「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 回も行っていない時の信念は初期信念 $x_i(0)$ として $[0,1]$ 間の一様分布からランダムに定まる
  k 回ゲームを終えた時点での信念 $x_i(k)$ は
  $$x_i(k) = \frac{x_i(0) + (\text{実際に相手が行動 1 をとった回数})}{k+1}$$
  と表される
  たとえば $x_0(0) = 0.5$ で、9 回目までにプレイヤー 1 が 7 回行動 1 を選択していたとしたらこの時点でのプレイヤー 0 の信念 $x_0(9)$ は

コードの説明

- ```
  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
  ```

コードの説明

- ```
  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
  ```

コードの説明

- ```
  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
  ```

コードの説明

- ```
  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
  ```

コードの説明

- ```
  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 (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 (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 (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 (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 (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

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

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

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
  ```

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 (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 (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 (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 (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)

- ```
  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]]
  ```

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

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

  という形に変形される
  ```

oneplay メソッド (2/5)

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

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

  という形に変形される
  ```

oneplay メソッド (2/5)

- ```
  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]]
  ```

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

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

  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 (3/5)

- for i in range(ts_length):
```
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 (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 (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 (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 (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

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

  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 (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

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

  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 (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

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

  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 (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

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

  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

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

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
  ```

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
  ```

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

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 (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 (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 (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 (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
  ```

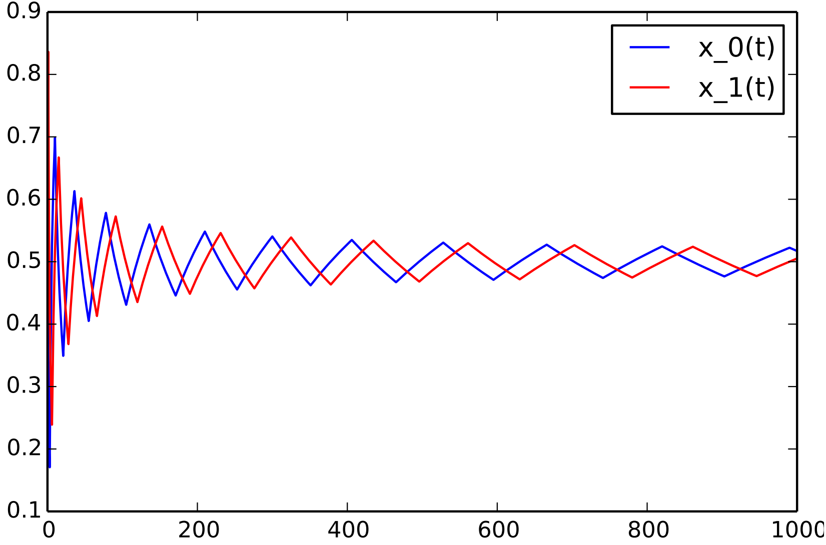
playplot メソッド (1/2)

- ```
  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
  ```

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
  ```

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
  ```

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
  ```

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
  ```

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 (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 (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 (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 (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 (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
  ```

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
  ```

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
  ```

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 (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
  ```

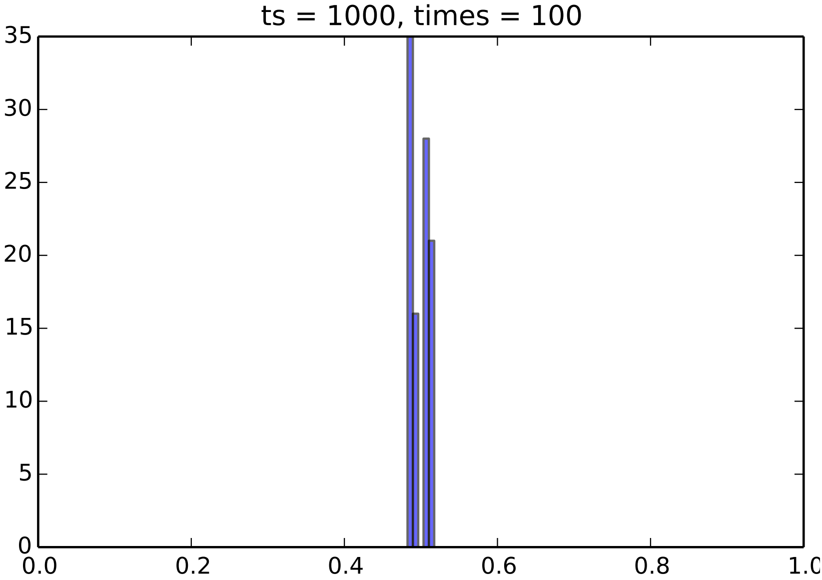
histplot メソッド (1/2)

- ```
  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 (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
  ```

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
  ```

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
  ```

まとめ

- まとめ
- よくわかっていない点とか
- 今後の課題とか