

# Visual Python, Numpy, Matplotlib

田浦健次郎  
電子情報工学科

東京大学

# Contents

- 1 イントロ
- 2 Visual Python
- 3 Numpy と Scipy
- 4 Scipy
- 5 Matplotlib

# Contents

1 イン트로

2 Visual Python

3 Numpy と Scipy

4 Scipy

5 Matplotlib

## 3つの飛び道具

- Visual Python: 3D アニメーションを超お手軽に
- Numpy, Scipy: 高度な行列・ベクトル計算や数値計算を一言で呼び出し
- Matplotlib: 関数やデータの可視化 (グラフ化)

# Contents

1 イントロ

2 Visual Python

3 Numpy と Scipy

4 Scipy

5 Matplotlib

# シミュレーションから可視化まで

- 1 シミュレーション (数値計算) をする
- 2 vector で 3D 世界の計算にする
- 3 Visual Python のオブジェを作ればできあがり

# 1つの質点のシミュレーションのテンプレート

要するにこれだけ

```
1 n_steps = ステップ数
2 dt = (終了時刻 - 開始時刻) / n_steps # 時間の刻み幅
3 x = 初期位置
4 v = 初速
5 t = 開始時刻
6 for i in range(n_steps):
7     alpha = 力 / 質量 # 加速度
8     x += v * dt      # 位置 += 速度 * 時間
9     v += alpha * dt  # 速度 += 加速度 * 時間
10    t += dt
```

- 赤字が問題によって本質的に変わる部分. 力を,  $t$ ,  $x$ ,  $v$  の式で書ければ実質的な仕事は終了
- 青字は問題設定によって決まる
- ステップ数は, 欲しい精度に応じて決める

## 適用例：吊るされたバネ

- バネが自然長のときの重りの位置を  $y = 0$  とする
- 時刻  $t = 0$  から 10 までシミュレーションするとする
- 初期位置  $y = 0$ , 初速  $v = 0$  とする
- 力:  $-ky + mg$  ( $g = -0.98$ )
- 注: 以下の例で  $t$  は力の計算に不要なので, 計算から削除している

```
1 k = 1.0
2 g = -9.8
3 m = 1.0
4 n_steps = 1000
5 dt = (10.0 - 0) / n_steps
6 y = 0.0
7 v = 0.0
8 for i in range(n_steps):
9     alpha = -k * y + m * g
10    y += v * dt
11    v += alpha * dt
```



## 3D世界の計算にする

- 数値の代わりに **Visual Python の vector を使う** → ほとんど変更なく各値をベクトルにできる
- 注: もちろんこの例においては運動自身は一次元内の運動なので、本質的な意味はない

```
1 from visual import *
2 k = 1.0
3 g = vector(0.0, -9.8, 0.0)
4 m = 1.0
5 n_steps = 1000
6 dt = (10.0 - 0) / n_steps
7 y = vector(0.0, 0.0, 0.0)
8 v = vector(0.0, 0.0, 0.0)
9 for i in range(n_steps):
10     alpha = -k * y + m * g
11     y += v * dt
12     v += alpha * dt
```

# アニメ化

- 位置を Visual Python のオブジェクトの pos にセットするだけ
- 速度や加速度も適宜オブジェクトの属性にするとわかりやすい
- 注: 以下ではバネ (helix) は表示していない

```
1 from visual import *
2 k = 1.0
3 g = vector(0.0, -9.8, 0.0)
4 m = 1.0
5 scene.autoscale = 0
6 n_steps = 1000
7 dt = (10.0 - 0) / n_steps
8 s = sphere(pos=vector(0.0, 0.0, 0.0))
9 s.vel = vector(0.0, 0.0, 0.0)
10 for i in range(n_steps):
11     rate(1.0/dt)
12     s.alpha = -k * s.pos + m * g
13     s.pos += s.vel * dt
14     s.vel += s.alpha * dt
```

## 物体が複数の場合

- 基本: 物体一つにつき位置, 速度などの変数を 1 セット
- 簡単な場合は, 物体 1 つ = Visual Python のオブジェクト一つ
- 物体が増えてきたらリストなどを使う

# Contents

1 イントロ

2 Visual Python

**3 Numpy と Scipy**

4 Scipy

5 Matplotlib

# Numpy と Scipy

- SciPy
- NumPy (Numerical Python)
- NumPy  $\subset$  SciPy ということのような
- numpy で提供されている機能はそのまま, scipy でも提供されている
- なので scipy だけで押し通しても良さそうだが, 世の中の説明は numpy が主流なので, それに合わせて, 基本は numpy, scipy だけで提供されている機能は scipy を使う

# 有用なチュートリアル

- 以下では speed learning のために基本を駆け足で説明する
- 適宜, 以下のページなどを参照すると良い  
[http://wiki.scipy.org/Tentative\\_NumPy\\_Tutorial](http://wiki.scipy.org/Tentative_NumPy_Tutorial)

# array : numpy の基本データ

- numpy の中心データは, array
- array を作る:

```
1 import numpy as np
2 np.array(リスト)
```

- 例:

```
1 import numpy as np
2 x = np.array([2,0,1,4])
3 print x
4 print len(x)
5 print x[1]
```

- 出力:

```
1 [2 0 1 4]
2 4
3 0
```

# 多次元の array

## ■ 例:

```
1 import numpy as np
2 A = np.array([[1,2,3],[4,5,6]])
3 print A
4 print len(A)
5 print A[1][1]
```

## ■ 出力:

```
1 [[1 2 3]
2  [4 5 6]]
3 2
4 5
```

## ■ 想像通り, 3次元, 4次元, ... の array も作れる



## array の演算

```
1 import numpy as np
2 x = np.array([2,0,1,4])
3 y = np.array([5,6,7,8])
4 print x + y
5 print x * y # 注意
6 print x.dot(y)
```

```
1 [ 7 6 8 12]
2 [10 0 7 32] # 要素ごとの *
3 49          # 内積
```

## array の演算 (続)

### ■ 行列 × ベクトル:

```
1 import numpy
2 A = numpy.array([[1,2,3],[4,5,6]])
3 x = numpy.array([2,4,6])
4 print A.dot(x)
```

```
1 [28 64]
```

### ■ 行列 × 行列

```
1 import numpy
2 A = numpy.array([[1,2,3],[4,5,6]])
3 B = numpy.array([[2,3],[4,5],[6,7]])
4 print A.dot(B)
```

```
1 [[28 34]
2  [64 79]]
```

## matrix : '\*' で行列積がしたければ

- array は任意の次元のデータの集まりを表す、汎用的なデータで、「行列」専用というわけではない(その意味で,\*の動作は自然)
- 「行列」を使いたければ matrix を使う

```
1 import numpy as np
2 A = np.matrix([[1,2,3],[4,5,6]])
3 B = np.matrix([[2,3],[4,5],[6,7]])
4 print A * B
```

```
1 [[28 34]
2  [64 79]]
```

- そして, array を受け付ける関数の殆どは, matrix も受け付ける

# arrayの色々な作り方 (1)

## よく使う基本形

### ■ 等差数列 (公差を指定)

```
1 >>> numpy.arange(2,3,0.2)
2 array([ 2. ,  2.2,  2.4,  2.6,  2.8])
```

### ■ 等差数列 (点数を指定)

```
1 >>> numpy.linspace(2,3,6)
2 array([ 2. ,  2.2,  2.4,  2.6,  2.8,  3. ])
```

### ■ 0や1を並べる

```
1 >>> numpy.zeros((3,2))
2 array([[ 0.,  0.],
3        [ 0.,  0.],
4        [ 0.,  0.]])
```

```
1 numpy.ones((2,3))
2 array([[ 1.,  1.,  1.],
3        [ 1.,  1.,  1.]])
```

## arrayの色々な作り方 (2)

### つぶしの効くやり方

- ややこしい物を作りたければ, zerosなどで形だけ作り, 各要素に値を代入すれば良い

```
1 import numpy
2
3 def make_diag(n):
4     A = numpy.zeros((n,n))
5     for i in range(n):
6         A[i,i] = i + 1
7     return A
8
9 print make_diag(4)
```

```
1 [[ 1.  0.  0.  0.]
2  [ 0.  2.  0.  0.]
3  [ 0.  0.  3.  0.]
4  [ 0.  0.  0.  4.]]
```

## arrayの色々な作り方 (3)

### reshape

- 要素の並びはそのままにして, 形 (次元や縦横の要素数) を変える
  - ベクトル (1次元)  $\leftrightarrow$  行列 (2次元) など
  - $3 \times 5$  行列  $\leftrightarrow$   $5 \times 3$  行列など

```
1 A = numpy.arange(0, 15, 1).reshape(3, 5)
2 print A
3 B = A.reshape(5, 3)
4 print B
```

```
1 [[ 0  1  2  3  4]
2  [ 5  6  7  8  9]
3  [10 11 12 13 14]]
4 [[ 0  1  2]
5  [ 3  4  5]
6  [ 6  7  8]
7  [ 9 10 11]
8  [12 13 14]]
```

## array の色々な作り方 (4)

その他たまに便利なもの

### ■ 乱数

```
1 >>> numpy.random.random((3,3))
2 array([[ 0.1065879 ,  0.99028258,  0.05101186],
3        [ 0.33271903,  0.42683107,  0.88456947],
4        [ 0.52302174,  0.63007686,  0.84804093]])
```

### ■ 各成分を関数で指定

```
1 >>> def f(i,j):
2     ...     return i + j
3     ...
4 >>> numpy.fromfunction(f, (3,3))
5 array([[ 0.,  1.,  2.],
6        [ 1.,  2.,  3.],
7        [ 2.,  3.,  4.]])
```

# 要素, 行, 列の取り出し

- 特定の行, 特定の列など, 自由に切り出すことが可能

```
1 import numpy
2 A = numpy.arange(0, 15, 1).reshape(3, 5)
3 >>> A
4 array([[ 0,  1,  2,  3,  4],
5        [ 5,  6,  7,  8,  9],
6        [10, 11, 12, 13, 14]])
7 >>> A[1,2]
8 7
9 >>> A[1:3,2:4]
10 array([[ 7,  8],
11        [12, 13]])
```



## 要素, 行, 列の取り出し

```
1 >>> A[1:3,:] # = A[1:3,0:5]
2 array([[ 5,  6,  7,  8,  9],
3        [10, 11, 12, 13, 14]])
4 >>> A[:,2:4] # = A[0:3,2:4]
5 array([[ 2,  3],
6        [ 7,  8],
7        [12, 13]])
8 >>> A[:,2]
9 array([ 2,  7, 12])
10 >>> A[:,:]
11 array([[ 0,  1,  2,  3,  4],
12        [ 5,  6,  7,  8,  9],
13        [10, 11, 12, 13, 14]])
```

# Universal 関数 (1)

- 数値に対する演算の多くがひとりでに、array に拡張されている

```
1 >>> import numpy as np
2 >>> r = np.linspace(0, 0.5 * math.pi, 6)
3 >>> r
4 array([ 0., 0.31415927, 0.62831853, 0.9424778,
          1.25663706, 1.57079633])
5 >>> r + 2
6 array([ 2., 2.31415927, 2.62831853, 2.9424778,
          3.25663706, 3.57079633])
7 >>> r ** 2
8 array([ 0., 0.09869604, 0.39478418, 0.8882644,
          1.5791367, 2.4674011 ])
```

## Universal 関数 (2)

- また, `sin`, `cos` など一部の数学関数は, `math` で提供されているものは `array` に適用不可だが, `array` にも適用可能なものが `numpy` で同名で提供されている

```
1 >>> import numpy as np
2 >>> import math
3 >>> r = np.linspace(0, 0.5 * math.pi, 6)
4 >>> math.sin(r)
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7 TypeError: only length-1 arrays can be converted to
   Python scalars
8 >>> np.sin(0.5 * math.pi)
9 1.0
10 >>> np.sin(r)
11 array([ 0., 0.30901699, 0.58778525, 0.80901699,
         0.95105652, 1.          ])
```

# Contents

1 イントロ

2 Visual Python

3 Numpy と Scipy

**4 Scipy**

5 Matplotlib

- とにかく色々なことが一撃でできるようになっている
- 行列の計算, 関数の最大最小, etc.
- 一度見て見といて! 以上!
- <http://docs.scipy.org/doc/scipy/reference/>

# Contents

1 イントロ

2 Visual Python

3 Numpy と Scipy

4 Scipy

**5 Matplotlib**

- データの可視化
- Visual のような物体の可視化, アニメーションではなく, データの可視化 (グラフ表示) を得意とする
- Matplotlib にデータを渡すのに, numpy の array 形式で渡す
- 機能が豊富で, 十数枚のスライドで多くをカバーするのは無謀
- 基本概念 + 最小限の例を説明する

後は、以下のページなどを適宜参照. おそらくこのスライドよりもよほど有用.

- document 全体: <http://matplotlib.org/1.3.1/contents.html>
- user's guide:  
<http://matplotlib.org/1.3.1/users/index.html>
  - Pyplot tutorial (一変数関数  $y = f(x)$  の表示)
  - mplot3d (3D で表示したい場合のキーワード)
- Gallery (こんな絵はどう書くんだろうと思ったらここから探す)  
<http://matplotlib.org/1.3.1/gallery.html>
- Plotting Command Summary (色々なグラフの書き方の総覧)  
[http://matplotlib.org/1.3.1/api/pyplot\\_summary.html](http://matplotlib.org/1.3.1/api/pyplot_summary.html)



# 1 変数関数 ( $y = f(x)$ ) をプロットする

- 最低限覚えるべき関数: matplotlib.pyplot モジュールの, plot と show
- 長すぎるので普通は,

```
1 import matplotlib.pyplot as plt
2 plt.plot(...)
3 plt.show(...)
```

- 基本概念: プロットしたいデータを, リストや array として, plot 関数に渡し, show で画面に表示

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 x = np.arange(0, 10, 0.01)
4 y = np.sin(x) # universal 関数
5 plt.plot(x, y)
6 plt.show()
```

## もう少し柔軟なプロットの仕方

- 複数のグラフを表示したい, 3D で表示したいという時には, 以下のスタイルで

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(行数, 列数, 通し番号)
4 ax.plot(...)
5 fig.show(...)
```

- 例 (3x2 のタイルで表示)

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax0 = fig.add_subplot(3,2,1) # ≡ fig.add_subplot(321)
4 ax1 = fig.add_subplot(3,2,2) # ≡ fig.add_subplot(322)
5     ...
6 ax0.plot(...)
7 ax1.plot(...)
8     ...
9 fig.show(...)
```

# 表示のカスタマイズ

- とにかく色々なことができます
- グラフの種類: bar で表示, 散布図表示, ... etc.
  - plot 関数の代わりに色々な関数があります
  - pyplot tutorial, gallery を参照
- 線の色とかスタイルとか: 個々の関数のマニュアルを読む. ググる, もしくは `help`

```
1 >>> import matplotlib.pyplot as plt
2 >>> help(plt.plot)
```

## 2変数関数 ( $z = f(x, y)$ ) をプロットする

### ■ 基本概念:

1 plot に変わる, 2変数用の関数を呼ぶ

1 pcolor (色表示)

2 contour (等高線表示)

3 etc. (gallery や plotting command summary 参照)

2 プロットしたいデータ  $(x_i, y_i, z_i)$  を,  $x_i$  だけ,  $y_i$  だけ,  $z_i$  だけの2次元配列にして渡す

■  $(x, y) \in [0, 2] \times [0, 3]$  内の格子点上で,  $x^2 - y^2$  を色表示

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 X = np.array([[0,0,0,0],[1,1,1,1],[2,2,2,2]])
4 Y = np.array([[0,1,2,3],[0,1,2,3],[0,1,2,3]])
5 Z = X ** 2 - Y ** 2 # array([[0,-1,-4,-9],[1,0,-3,-8]])
6 plt.pcolor(X, Y, Z)
7 plt.show()
```

## 2変数関数 ( $z = f(x, y)$ ) をプロットする

- 格子点を作る便利な関数に、`np.meshgrid` がある
- 先と同じ例

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 X = np.array([0,1,2])
4 Y = np.array([0,1,2,3])
5 X,Y = np.meshgrid(X, Y)
6 Z = X ** 2 - Y ** 2 # array([[0,-1,-4,-9],[1,0,-3,-8]])
7 plt.pcolor(X, Y, Z)
8 plt.show()
```

## 2変数関数 ( $z = f(x, y)$ ) をプロットする. **ただし3Dで**

- 基本: plot にかわる「それ用の」関数を使う
  - plot\_surface
  - plot\_wireframe
  - etc.
- ただし, 二つほどまじないが必要

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import mpl_toolkits.mplot3d.axes3d
4
5 fig = plt.figure()
6 ax = fig.add_subplot(1,1,1, projection='3d')
7 X = np.linspace(0,1,11) # array([0,0.1,0.2,...,1.0])
8 Y = np.linspace(0,1,11)
9 X,Y = np.meshgrid(X, Y)
10 Z = X ** 2 - Y ** 2 # array([[0,-1,-4,-9],[1,0,-3,-8]])
11 ax.plot_surface(X, Y, Z)
12 fig.show()
```

- <http://matplotlib.org/1.3.1/gallery.html#mplot3d> 参照