Visual Python, Numpy, Matplotlib

田浦健次朗
電子情報工学科

東京大学

Contents

Contents

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

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

```
n_steps = ステップ数
dt = (終了時刻 - 開始時刻) / n_steps # 時間の刻み幅
x = 初期位置
v = 初速
t = 開始時刻
for i in range(n_steps):
    alpha = 力 / 質量   # 加速度
    x += v * dt         # 位置 += 速度 * 時間
    v += alpha * dt     # 速度 += 加速度 * 時間
    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 vpython 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 vpython 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  cv = canvas()
8  s = sphere(pos=vector(0.0, 0.0, 0.0))
9  s.vel = vector(0.0, 0.0, 0.0)
10 cv.autoscale = 0 # 場合によりけり
11 cv.autocenter = 0 # 場合によりけり
12 for i in range(n_steps):
13     rate(1.0/dt)
14     s.alpha = -k * s.pos /m + g
15     s.pos += s.vel * dt
16     s.vel += s.alpha * dt
```

アニメーション化する際のいくつかのトラップ

- rate(f)：これを時々よばないと (pos を変更しても) 画面は更新されない
  - f：更新頻度が 1 秒 f 程度になるよう時間調整
- カメラの自動追随機能により，動いているオブジェクトが動いているように見えないことがある (例えば 1 個しかオブジェクトがない場合に顕著)
  - 解 1: 動かないオブジェクト (板とか) も何か表示する
  - 解 2: 初期状態を表示し終えたところで，以下のおまじないをとなえる

```
1 cv = canvas()
2     ...
3 # 初期配置完了したあたりで
4 cv.autocenter = 0
5 cv.autoscale = 0
```

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

Numpy と Scipy

- SciPy
- NumPy (Numerical Python)
- NumPy $\subset$ SciPy ということのようだ
- 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`

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)

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

- 等差数列 (点数を指定)

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

- 0 や 1 を並べる

```
>>> np.zeros((3,2))
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
```

```
np.ones((2,3))
array([[ 1.,  1.,  1.],
       [ 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 as np
2
3 def make_diag(n):
4     A = np.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) ↔ 行列 (2 次元) など
  - 3×5 行列 ↔ 5×3 行列など

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

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

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

- 関数で作る

```
>>> def f(I,J):
...   return I + J
...
>>> np.fromfunction(f, (3,3))
array([[ 0.,  1.,  2.],
       [ 1.,  2.,  3.],
       [ 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 = np.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

```
>>> A[1:3,:]   # = A[1:3,0:5]
array([[ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> A[:,2:4]   # = A[0:3,2:4]
array([[ 2,  3],
       [ 7,  8],
       [12, 13]])
>>> A[:,2]
array([ 2,  7, 12])
>>> A[:,:]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [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 にも適用可なものが numpy で同名で提供されている

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

Numpy に備わる飛び道具

- 連立一次方程式 $(Ax = b)$ を解く

```
1 x = np.linalg.solve(A, b)
```

- 固有値，固有ベクトル

```
1 d,p = np.linalg.eig(A)
```

- 行列式

```
1 rank = np.linalg.det(A)
```

- ランク

```
1 rank = np.linalg.matrix_rank(A)
```

- その他いっぱい...

Contents

Scipy

- とにかく色々なことが一撃でできるようになっている
- 積分 (scipy.integrate)
- 常微分方程式 (scipy.odeint)
- 関数の最大最小 (scipy.optimize)
- ...
- 仮に今回使わなくても scipy という単語を覚えておくだけで，将来きっと役に立つ
- `http://docs.scipy.org/doc/scipy/reference/`

Contents

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

Matplotlib

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/contents.html`
- user's guide: `http://matplotlib.org/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)
  `https://matplotlib.org/gallery/`
- Plotting Command Summary (色々なグラフの書き方の総覧)
  `https://matplotlib.org/3.2.1/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 というモジュールの, plot と show
- 長すぎるので普通は,

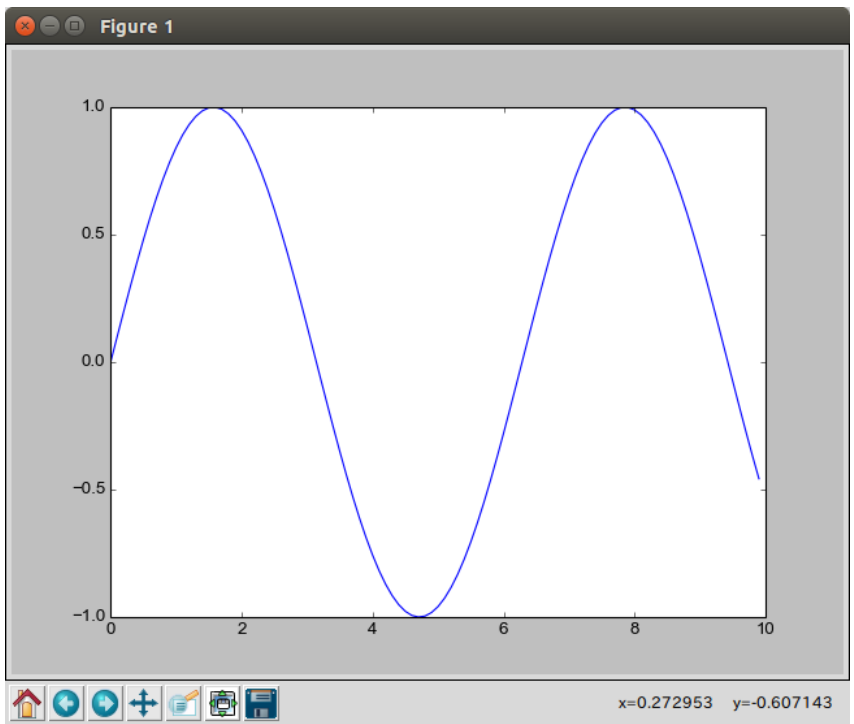
```
1 import matplotlib.pyplot as plt
2 plt.plot(...)
3 plt.show(...)
```

- 基本概念: プロットしたいデータの x 座標, y 座標をそれぞれリストや array として, plot 関数に渡し, show 関数で画面に表示

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 x = np.arange(0, 10, 0.1) # [0 0.1 0.2 0.3 .. ]
4 y = np.sin(x)  # universal 関数
5 # y = [sin(0) sin(0.1) sin(0.2) ... ]
6 plt.plot(x, y)
7 plt.show()
```

1変数関数 $(y = f(x))$ のプロット例

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 x = np.arange(0, 10, 0.1) # [0 0.1 0.2 0.3 .. ]
4 y = np.sin(x)  # universal 関数
5 # y = [sin(0) sin(0.1) sin(0.2) ... ]
6 plt.plot(x, y)
7 plt.show()
```



色々な表示の仕方

- とにかく色々な表示の種類 (棒グラフ, 散布図, . . . etc.) があります
- `plot` 関数の代わりに別の関数を使えば良い
  - 棒グラフ →`bar`, 散布図 →`scatter`
  - pyplot tutorial, gallery を参照
- 線の色とかスタイルを変えたい? 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 ($z = f(x, y)$) を表示する

- 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        # universal 関数
6 # Z = array([[0,-1,-4,-9],[1,0,-3,-8],[4,3,0,-5]])
7 plt.pcolor(X, Y, Z)
8 plt.show()
```

- X, Y の作り方が面倒 (冗長) なのでなんとかしたい

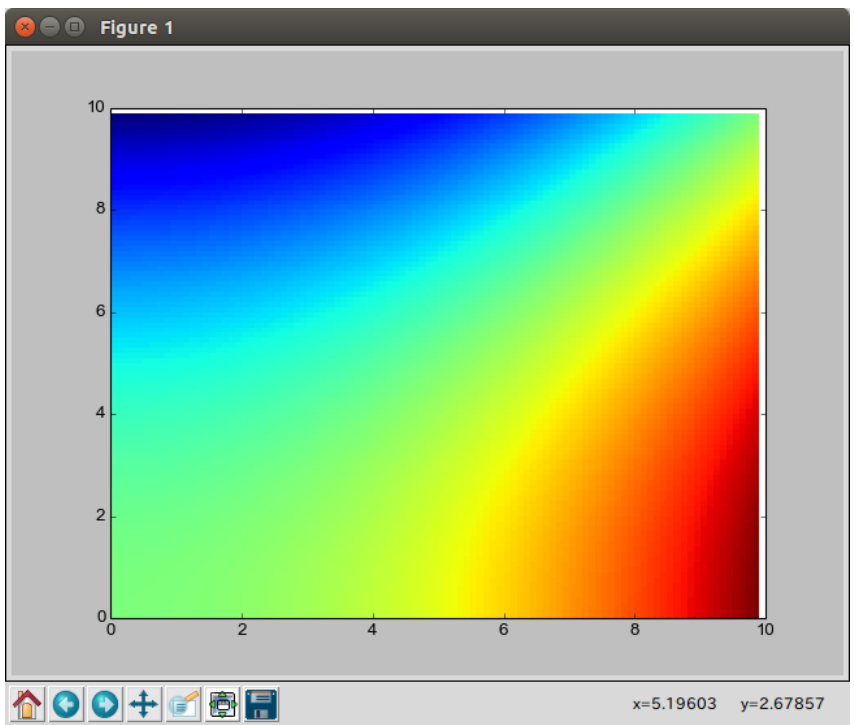
2変数関数 $(z = f(x, y))$ をプロットする

- 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
7 plt.pcolor(X, Y, Z)
8 plt.show()
```

2変数関数 $(y = f(x, y))$ のプロット例

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 X = np.arange(0, 10, 0.1)
4 Y = np.arange(0, 10, 0.1)
5 X,Y = np.meshgrid(X, Y)
6 Z = X ** 2 - Y ** 2
7 plt.pcolor(X, Y, Z)
8 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

```
import matplotlib.pyplot as plt
fig = plt.figure() # windows 全体
ax = fig.add_subplot(行数，列数，通し番号) # 一つのグラフ
ax.plot(...)
fig.show(...)
```

- 例 (3x2 のタイルで表示)

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

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

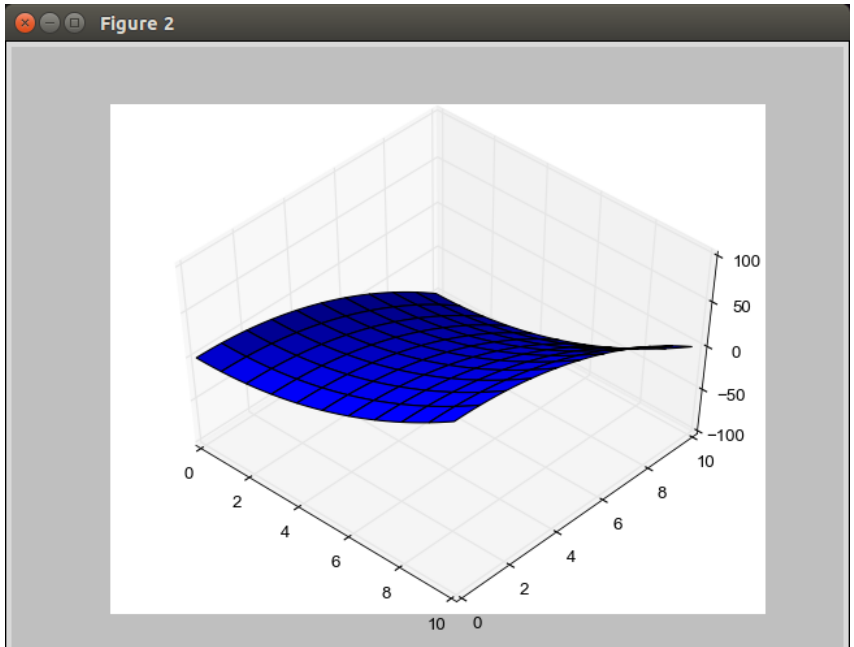
- 基本: plot にかわる「それ用の」関数を使う
  - plot_surface
  - plot_wireframe
  - etc.
- ただし，二つほどまじないが必要

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

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

3D でプロットの例

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



Matplotlib でのアニメーション

- 場が時間とともに発展するようなシミュレーションでは, 最終結果だけでなく, 途中も表示したい
- 最終結果がおかしい時, その原因を究明するためにも重要
- いくつか方法があるのだが, Jupyter 環境のシステム上の問題で, 動かないときが多い

例題

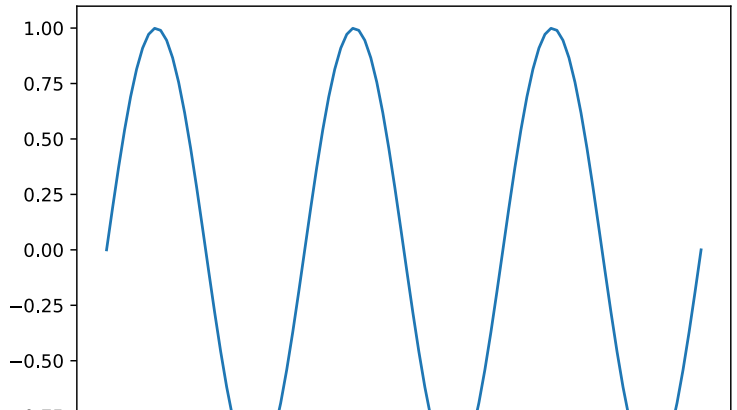
- 問題:

$$y = \sin kx \quad (0 \leq x \leq 2\pi)$$

のグラフを, $k = 1, 2, 3, \cdots 5$ まで変えながら表示する

- 復習: 固定した k $(= 3)$ について表示するだけでよいのなら以下

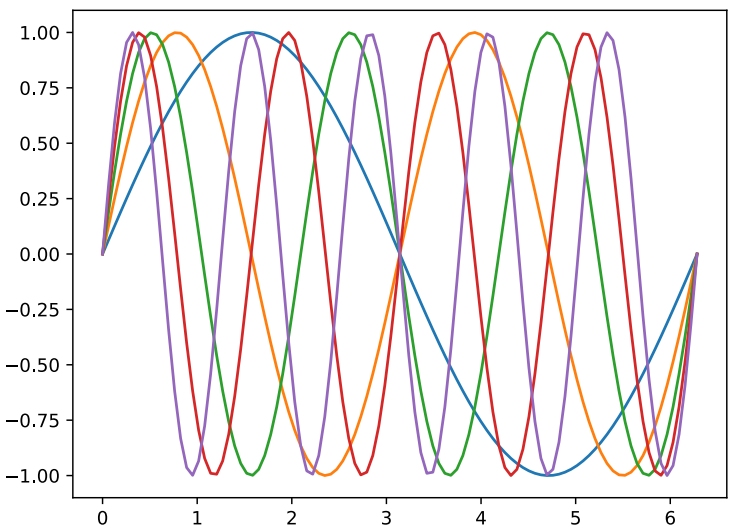
```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0.0, 2.0 * np.pi, 100)
5 plt.plot(x, np.sin(3 * x))
6 plt.show()
```



一見それらしいけど不正解な例

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0.0, 2.0 * np.pi, 100)
5 for k in range(1, 6):
6     plt.plot(x, np.sin(k * x))
7 plt.show()
```

- おこること: 5 つのグラフが全部 (重ねて) 表示される



アニメーションの方法

- 以下にアニメーションの方法を 3 つ述べるが, (田浦の調べた限り), Jupyter 内でアニメーションができるのは, 方法 1 だけ
- 他の方法は Jupyter 内では動かない
  - (授業ではやっていない) python コマンドで実行すれば動く

方法 1

- plt.plot(...) した結果は, 「曲線」(のリスト)
- それらをリストに全て溜め (∗), animation.ArtistsAnimation という関数に渡すと, それらを順にアニメーションしてくれる
- %matplotlib notebook というおまじないが必要

```
1  %matplotlib notebook
2  import matplotlib.pyplot as plt
3  import matplotlib.animation as animation
4  import numpy as np
5
6  x = np.linspace(0.0, 2.0 * np.pi, 100)
7  plots = []
8  for k in range(1, 6):
9      plots.append(plt.plot(x, np.sin(k * x)))
10 a = animation.ArtistAnimation(plt.gcf(), plots, repeat=0)
11 plt.show()
```

注意

- 表示したい絵の数を無闇に大きくできない
- 例えば 1 万ステップのシミュレーションで全ステップの絵を表示するのは無謀
- リストの長さが適度になるように間引きする必要がある. 例えば以下は表示する絵を max_frames 個以下に押さえる方法

```
1 n_steps = 123456
2 max_frames = 50
3 interval = math.ceil(n_steps / max_frames)
4 plots = []
5 for t in range(n_steps):
6     if t % interval == 0:
7         plots.append(...)
```

- また, この方法の欠点: 計算が全て完了してから結果が初めて表示される
- 計算しながら表示するには方法 2, 3 が必要だが, 残念ながら Jupyter では動かない

(参考) 方法 2

- 最初に一度 `plt.plot(...)` して結果 (曲線) を変数に入れる (∗)
- その曲線のデータを `set_data` で変更 (†)
- 必要に応じて `pause` で変更間隔を調整 (††)
- (田浦の知る限り) Jupyter 内では動かない (python のコマンドラインでなら動く. 詳細不明)

```
1 x = np.linspace(0.0, 2.0 * np.pi, 100)
2 for k in range(1, 6):
3     if k == 1:
4         [ line ] = plt.plot(x, np.sin(k * x)) # (∗)
5     else:
6         line.set_data(x, np.sin(k * x)) # (†)
7     plt.pause(0.1) # (††)
```

- 注: `plt.plot(...)` は, 描かれた曲線のリスト (普通は 1 要素) を返す; 実は一度に複数の曲線を描ける

(参考) 方法 3

- (Jupyter では動かない上, 難しいのであまり気にしないで)
- 方法 2 に似た方法. 但し以下のように「関数」にして, 書き直したいものを `yield` する (∗)

```
def generate_plots():
    x = np.linspace(0.0, 2.0 * np.pi, 100)
    for k in range(1, 6):
        if k == 1:
            [line] = plt.plot(x, np.sin(k * x))
        else:
            line.set_data(x, np.sin(k * x))
        yield [line] # (∗)
```

- その上で,

```
animate_iterator(generate_plots(), interval=100)
```

- ただし `animate_iterator` は次ページ

animate_iterator

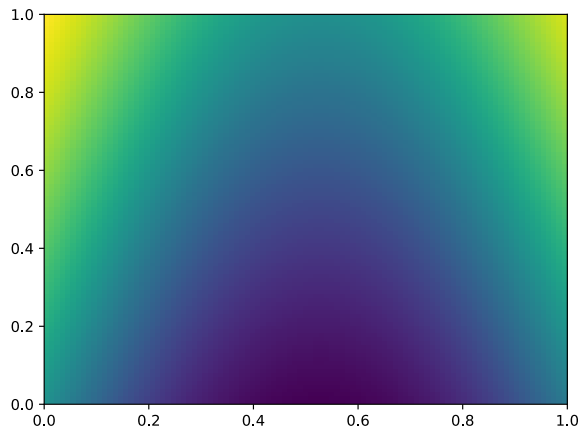
- 以下はコピペでよい (説明は難しいので省略)

```
1  import matplotlib.pyplot as plt
2  import matplotlib.animation as animation
3
4  def animate_iterator(iterator, **kwargs):
5      def fun(*args):
6          try:
7              return next(iterator)
8          except StopIteration:
9              return []
10     ani = animation.FuncAnimation(plt.gcf(), fun,
11                                   **kwargs)
12     plt.show()
```

2次元 (pcolor) の場合

- 復習: アニメーションなしで静止画 ($z = y - \sin 3x$) を書くだけの例

```
1 %matplotlib notebook
2 import matplotlib.pyplot as plt
3 import numpy as np
4 x = np.linspace(0.0, 1.0, 100)
5 y = np.linspace(0.0, 1.0, 100)
6 x,y = np.meshgrid(x, y)
7 plt.pcolor(x, y, np.sin(y - np.sin(3 * x)))
8 plt.show()
```



2次元 (pcolor) 方法1

- 1次元と考え方は同じ (pcolor の結果をリストにためて, animation.ArtistAnimation を呼び出す)

```
1  %matplotlib notebook
2  import matplotlib.pyplot as plt
3  import matplotlib.animation as animation
4  import numpy as np
5  x = np.linspace(0.0, 1.0, 100)
6  y = np.linspace(0.0, 1.0, 100)
7  x,y = np.meshgrid(x, y)
8  pcolors = []
9  for k in range(1, 20):
10     pcolors.append([ plt.pcolor(x,y,y-np.sin(k*x)) ])
11 a = animation.ArtistAnimation(plt.gcf(), pcolors,
12                                repeat=0)
13 plt.show()
```

- plt.plot の結果はリストだったが, (ややこしいことに)plt.pcolor の結果はリストではないので, 自分で1要素のリストにする (*)

(参考) 2 次元 (pcolor) 方法 2

- 注: Jupyter では動かない
- 1 次元 (plot) では set_data を呼んだが, 2 次元 (pcolor) では set_array を呼ぶ
- ややこしいことに, z 座標のデータを 1 ずつ削り, かつ 1 次元配列に直して渡さないといけない (理由不明)

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  x = np.linspace(0.0, 1.0, 100)
4  y = np.linspace(0.0, 1.0, 100)
5  x,y = np.meshgrid(x, y)
6  for k in range(1, 20):
7      if k == 1:
8          f = plt.pcolor(x, y, y - np.sin(k * x))
9      else:
10         f.set_array(shrink1(y - np.sin(k * x)))
11     plt.pause(0.1)
```

- shrink1 は次ページ

shrink1

```
1 def shrink1(z):
2     m,n = z.shape
3     return z[:m-1,:n-1].flatten()
```

(参考) 2 次元 (pcolor) 方法 3

- (Jupyter では動かない上, 難しいのであまり気にしないで)

```
def generate_pcolors():
    x = np.linspace(0.0, 1.0, 100)
    y = np.linspace(0.0, 1.0, 100)
    x,y = np.meshgrid(x, y)
    for k in range(1, 20):
        if k == 1:
            f = plt.pcolor(x, y, y - np.sin(k * x))
        else:
            f.set_array(shrink1(y - np.sin(k * x)))
        yield [ f ]
```

- その上で,

```
animate_iterator(generate_pcolors(), interval=100)
```

- animate_iterator は 1 次元の場合と同じ