スクリプト言語プログラミング
Pythonによる数値解析

第12回講義資料

箕原辰夫

Numpyを使った多次元配列（ndarray）

- **import** numpy **as** np
- np.array(リスト) → numpy上の配列のオブジェクトに変換される
- np.array(２次元リスト) → numpy上の２次元配列のオブジェクトに変換される

Numpyの配列の要素の型指定

- np.array(リスト, dtype=numpyの型)
- 標準では、実数はfloat64になっている。ただし表示は10桁程度。整数はint64で対応、複素数はcomplex128、文字列はunicodeになっている
- numpyの型一覧
  - 整数（ビット数およぼ符号無し指定）：int8, int16, int32, int64, uint8,uint16, uint32, uint64
  - 実数（ビット数指定）：float16, float32, float64, float128
  - 複素数（ビット数指定）：complex64, complex128, complex256
  - その他：bool, unicode, object
- 配列.astype(numpyの型)で、各要素の型を変換したものが返される
  - 例：a.astype(np.float128)
- 参照：https://note.nkmk.me/python-numpy-dtype-astype/

Numpyの多次元配列のインデックス参照

- インデックス参照
  - 配列名[インデックスの整数式]　例：a[5], a[-6]
- 多次元配列のインデックス参照
  - 配列名[1次元目, 2次元目, ...]　例：a[3, 4], a[-2, -1, 3]
- ファンシーインデックス参照（リストのインデックスの要素だけがフィルタリングされたものが返される）
  - 配列名[インデックスのリスト]　例：a[[3, 7, 2]]
- ブールインデックス参照（論理リストのTrueに対応する要素だけがフィルタリングされて返される）
  - 配列名[論理値リスト]　例：a[[True, False, True, True]]

- 参照：https://docs.pyq.jp/python/pydata/numpy/math/index_ref.html

Numpyの配列のスライスと論理式によるフィルタリング

- スライス
  - 配列変数[開始位置 : 終了位置+1]
  - 配列変数[開始位置 : 終了位置+1 : 差分]

- 論理式によるフィルタリング
  - 配列変数[配列変数名を含む論理式]
  - 例： xa = np.array([-3, -4, -1, 2, 3, 4, 6])

```
even = xa[ xa % 2 == 0 ]
plus = xa[  xa >= 0 ]
```

多次元配列のスライス

- 2次元配列の場合には、スライスとインデックスを組み合わせて、列ごとに切り出した1次元配列を取り出すことができる。
  - 例：

```
import numpy as np
a = np.array( [ [1, 2], [3, 5], [4, 6], [7, 9] ] )
a[ :, 1 ]
# → array([2, 5, 6, 9])
a[ 1:, 0 ]
# → array([3, 4, 7])
```

- また、2次元目もスライスで指定した場合、部分的な2次元配列を取り出すことがが可能になっている

```
b = np.array( [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ] )
b[ 1:,1:]
# → array([[5, 6],
      [8, 9]])
b[ :, 2:]
# → array([[3],
      [6],
      [9]])
```

Numpyの配列生成関数

- np.arange(初期値, 上限, 差分)
- np.linspace(初期値, 上限, 分割数, endpoint=True)
- np.ones(個数)→要素が 1 から構成される配列を生成、np.ones((n, m))→n行m列の2次元配列で作成
- np.zeros(個数)→要素が 0 から構成される配列を生成、np.zeros((n, m))→n行m列の2次元配列で作成
- np.full(個数, 値)→要素が値から構成される配列を生成、np.full((n, m), value)→n行m列の2次元配列で作成
- np.eye(次数)→対角成分に 1 を持つ２次元配列を生成（正方行列の単位行列の２次元配列）
- np.diag(リスト)→リストの要素を対角成分にもつ２次元配列を生成（正方行列の２次元配列）
- np.random.rand(個数)→個数分一様乱数（0〜1）を発生した配列を返す
- np.random.randn(個数)→正規乱数（平均：0, 分散: 1）を発生した配列を返す
- np.random.randint(下限, 上限, 個数)→一様乱数（下限〜上限-1）を個数分発生
- rng = np.random.default_rng() # 標準の乱数発生器
  - rng.random(None)→1個だけ一様乱数を発生
  - rng.random(個数)→個数分の一様乱数を発生し、配列を返す
  - rng.random((個数, 個数))→個数分の一様乱数を発生し、2次元配列を返す

Numpyのarrayの属性

- ndim … 次元数（np.ndim(array)でも同じ）
- shape … 各次元の要素の個数をタプルで返す（np.shape(array)でも同じ）
- base…そのarrayのベース（共通）となっている配列を返す
- size…要素全体の個数（np.size(array)でも同じ）
- T…転置行列を配列として返す（２次元以上の場合、np.transpose(array)でも同じ））
- flat…各要素を巡回できるようなイテレータを返す（np.array(array.flat)にしないと、1次元配列にはならない、array.flatten()で1次元配列に変換可能）
- real/imag…実数・虚数部分を返す（要素が複素数の場合）
- len(配列) … 1次元目の大きさ
- len(配列[0])…2次元目の大きさ

Numpyのarrayのメソッド

- item(インデックス)…指定されたインデックスの要素の値を返す
- tolist()…Pythonのリストに変換したものを戻す
- tofile(ファイル名, sep=区切り文字列 , format=printfのフォーマット文字列)…テキストファイルに配列の内容を出力する
- copy()…その配列のコピーを返す
- view()…その配列を共有するビューを作成する
- fill(値)…指定された値ですべての要素を埋め尽くす

Numpyの配列変更の関数

- 要素の追加・挿入・削除
  - delete: インデックスの位置の要素を削除した配列を生成する（axis=Noneだと2次元以上の配列でも結果は1次元配列になる）
    - a = np.delete(a, index, axis=None)
    - https://note.nkmk.me/python-numpy-delete/
  - insert: 配列のインデックスの位置に要素やリストを挿入した配列を生成する（axis=Noneだと2次元以上の配列でも結果は1次元配列になる）
    - a = np.insert(a, index, value or list, axis=None)
    - https://note.nkmk.me/python-numpy-insert/
  - append: 配列の最後に要素を追加した配列を生成する（axis=Noneだと2次元以上の配列でも結果は1次元配列になる）
    - a = np.append(a, value, axis=None)
    - 2次元配列だと、aとvalueが同じ行数列数でないとエラーになる
    - https://note.nkmk.me/python-numpy-append/
- 繰返しで要素を作っていく
  - tile: 配列の並びが指定回繰り返された配列ができる
    - a = np.tile(a, times) # timesは整数あるいはタプル
    - timesがタプルになっていると、各要素分、その次元に繰り返される
  - repeat: 各要素が指定回繰り返した配列ができる（axis=Noneだと2次元以上の配列でも結果は1次元配列になる）
    - a = np.repeat(a, repeats, axis=None) # timesは整数あるいは整数を要素として持つ1次元リスト
  - 参考：
    - https://numpy.org/doc/stable/reference/generated/numpy.repeat.html#numpy-repeat
    - https://note.nkmk.me/python-numpy-tile/

配列の要素・次元の変更

- resize/reshape/flip/flatten/ravel
  - reshape: 配列の次元を変更する
    - a = np.reshape(a, (各次元のサイズ))
    - 同じ要素の個数でなければならない、1次元に変更する場合は、タプルでなくて良い
    - reshapeは、ビューとしての配列を返すので、ビューから元の配列の要素が変更される
  - resize: 配列の次元を変更する
    - a = np.resize(a, (各次元のサイズ))
    - 要素の個数は増やしたり、減らしたりできる（増やした場合は、元の要素の値が繰り返される）、1次元の場合は、タプルでなくて良い
    - resizeは変更された配列のコピーを返す
  - flatten: 配列を 1次元にする
    - a = a.flatten( )
    - flattenは1次元された配列のコピーを返す
  - ravel: 1次元にする
    - a = np.ravel(a)
    - ravelは1次元にされた配列のビューを返す
  - 参考：
    - https://note.nkmk.me/python-numpy-reshape-usage/
    - https://note.nkmk.me/python-numpy-ravel-flatten/

配列の分割

- 配列の分割
  - split: 配列を分割する
    - ビューとして分割された配列のリストができる
    - 書式：np.split(a, 個数またはリスト)
    - 個数だと、個数ごとに分割される
    - リストだと、各リストの要素が次に分割されるリストのインデックスの先頭と見なされる
    - 例：np.split(a, 2)
    - 例：np.split(a, [1,2,3]) # 4xNの行列
  - hsplit: 2次元以上の配列を横に分割する
    - 例 ：np.hsplit(a, 2) # 横に2分割（2次元だと結果は、列方向に2つ分かれた2次元配列のリスト）
  - vsplit: 2次元以上の配列を縦に分割
    - 例：np.vsplit(a, 4) # 縦に4分割（2次元だと結果は、行方向に4分割された2次元配列のリスト）
  - https://note.nkmk.me/python-numpy-split/

配列の結合

- 配列の結合（配列の結合はすべてコピーが生成される）
  - concatenate: 配列を結合する
    - 例：np.concatenate([a, b]) # 標準は行方向
    - np.concatenate([a, b], axis=0) # 行方向
    - np.concatenate([a, b], axis=1) # 列方向
  - stack: 配列を新たな軸を指定して結合する
    - 例：np.stack([a, b]) # a, bは1次元だと、2次元配列になる、a, bが2次元配列だと3次元配列になる
  - vstack: 2次元以上の配列を縦に結合する
    - 例：np.vstack([a, b])
  - hstack: 2次元以上の配列を横に結合する
    - 例：np.hstack([a, b])
  - block: 配置をリストで記述して結合
    - 例：np.block([a, b]) # 横結合

      np.block([[a],[b]]) # 縦結合

      np.block([[[a]],[[b]]]) # 3次元結合
  - https://note.nkmk.me/python-numpy-concatenate-stack-block/

Numpyのr_およびc_オブジェクト

- r_は、Row方向
- 配列の結合
  - import numpy as np
    a, b = np.array([1, 2, 3]), np.array([4, 5, 6])
    np.r_[a, b]
    ⇒ np.array([1, 2, 3, 4, 5, 6])
- スライスによる配列の作成
  - np.r_[1: 10]
    ⇒ np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

    np.r_[: 5]
    ⇒ np.array([0, 1, 2, 3, 4])

    np.r_[2: 10: 2]
    ⇒ np.array([2, 4, 6, 8])

    np.r_[1:5 , 0, 4, np.array([3,2])]
    ⇒ np.array([1, 2, 3, 4, 0, 4, 3, 2])
- c_は、Column方向
  - a = np.arange(1, 4)
    b = np.arange(4, 7)
    np.c_[a, b]
    ⇒

    [[1 4]
     [2 5]
     [3 6]]
  - np.c_[1:5,5:9]
    ⇒

    [[1 5]
     [2 6]
     [3 7]
     [4 8]]

r_, c_の結合の仕方を文字列で指定する

- 数値の文字列(string)による軸や次元の指定
  - "a,b,c"で指定、標準は"0,0,-1"
  - aは、どの軸（axis）の方向に沿って配列を結合するのか
  - bは、できあがる配列の次元数の最小値
  - cは、次元数の少ない配列の次元数の拡張を行った際、形状（shape）の表記として、どこに配列の最後の次元が置かれるべきか
- 例：

```
a = np.ones((2, 2))
b = np.zeros((2, 2))
```

```
np.r_["1", a, b]
⇒

[[ 1.,  1.,  0.,  0.],
 [ 1.,  1.,  0.,  0.]]

np.r_["0", a, b]
⇒

[[ 1.,  1.],
 [ 1.,  1.],
 [ 0.,  0.],
 [ 0.,  0.]]
```

- 参考資料：https://deepage.net/features/numpy-cr.html

Numpyの演算

- array同士の演算
  - ベクトルとしての演算ライブラリになる
  - スカラー値との演算の場合は、各要素に適用され、新しい配列が生成される
  - 内部的にはndarrayで実装されている
  - 行列として乗算するときは、@演算子を用いる
- matrixの演算（現在はndarrayを使うように推奨されている）
  - array(ndarray)クラスのサブクラスになっているので、arrayクラスで使えるものは、すべて使える
  - 行列としての演算ライブラリになる

- arrayとmatrixは、演算の方式が異なる部分もあるので、どちらを使うかは、目的に応じて

Numpyのarray同士の演算

- 配列 ± 配列…要素同士の加減算
- 配列 * 数値 あるいは 数値 * 配列…各要素のスカラー倍
- 配列 * 配列…要素同士の乗算（アダマール積：Hadamard product）
- 配列 @ 配列…行列の乗算（1次元の場合は、要素同士の乗算の総和になる）

  Python 3.5より
- 配列 / 配列…要素同士の除算
- 配列 ** 数値…各要素のべき乗
- 配列 ** 配列…各要素の後ろの要素のべき乗

Numpyのブロードキャスティング

- 加算や減算を2つの配列に対して行なうとき
- ルール1：次元数を揃える
  - 2つの配列の次元数が異なる場合、次元数が少ない方の配列の先頭にサイズ（長さ）が1の新しい次元を追加して次元数を揃える。
- ルール2：各次元のサイズ（長さ）を揃える
  - 2つの配列の各次元のサイズが一致しない場合、サイズが1である次元は他方の配列の次元のサイズに引き伸ばされる（値が繰り返される）。
  - 2つの配列のどちらのサイズも1ではない次元が存在するとき、ルール1が適用できないため、ブロードキャストできずにエラーとなる。
- なお、配列ndarrayの次元数はndim属性、形状はshape属性で取得できる。
- 参照：https://note.nkmk.me/python-numpy-broadcasting/

Numpyのarrayのベクトル演算

- np.linalg.norm(配列)…ノルム（大きさ）を求める
- np.inner(配列, 配列)あるいはnp.dot(配列, 配列)…内積を求める
- np.outer(配列, 配列)…外積を求める（各要素同士の積の総当たりとなる２次元配列が生成）
- np.cross(配列, 配列)…クロス積を求める

ベクトルの直積

- 直積あるいはデカルト積（Cartesian product）は、ベクトルの各成分の組合わせを求めるものである。
- numpyには、用意されておらず、scikit-learnのextmathモジュールにcartesianが用意されている。
  - **from** sklearn.utils.extmath **import** cartesian
  - cartesian((リスト, リスト, ...)) → 直積の2次元リストが返される
  - リストは、2つ以上必要になる。タプルとして渡すので、外側に丸括弧が必要。
- 参考：

  https://funmatu.wordpress.com/2018/09/02/numpyで直積（デカルト積, cartesian-product），順列（permutations），組

numpyの行列・ベクトル用関数

- np.dot(a, b[, out])　2つの配列のドット積（行列の場合は通常の行列の積：dot product)
- np.vdot(a, b)　2つのベクトルのドット積
- linalg.multi_dot(arrays, *[, out])　2つ以上の配列のドット積を求める、なおどのドット積～求めるかは、自動的に最速の評価順序で決定される
- np.inner(a, b)　2つの配列の内積 (inner product)
- np.outer(a, b[, out])　2つのベクトルの外積 (outer product)
- np.matmul(x1, x2, /[, out, casting, order, …])　2つの配列の行列積（Matrix product)
- np.tensordot(a, b[, axes])　指定された軸（axes）に添ってのテンソル・ドット積（tensor dot product)
- np.linalg.matrix_power(a, n)　正方行列のn乗（整数）
- np.kron(a, b)　2つの配列のクロネッカー積 (Kronecker product)

Numpyの数学関数演算

- ndarrayを引数に持つと、すべての要素に対して関数が適用され、結果のndarrrayが生成される。計算は整数, 実数, 複素数は、それぞれ、int64, float64, complex128が基本
- 三角関数
  - sin( ), cos( ), tan( )
  - arcsin( ), arccos, arctan( )
  - radians( ), degrees( ), deg2rad( ), rad2deg( )
- 指数・対数関数
  - power( ), exp( ), sqrt( )
  - log( ), log2( ), log10( ), log1p( )
- 整数・絶対値変換
  - ceil( ), floor( ), trunc( ), round( ), around( ), rint( ), fix( )
  - fabs( ), absolute( ), abs( )
- 定数
  - pi, e
- 参考：https://deepage.net/features/numpy-math.html

Numpyの行列

- 現在、使用は推奨されていない
- **import** numpy **as** np

- np.matrix(２次元リスト) → numpy上の行列のオブジェクトに変換される
- np.mat(２次元リスト) → numpy上の行列のオブジェクトに変換される
- np.mat("文字列") → 文字列は、「要素 … ; 要素 …」の形になっていること
  （例：np.mat("1 2; 3 4") ）
- np.bmat(データ)→小行列を足し合わせて行列を作る

行列では演算子が使える

- 行列 + 行列…行列の和
- 行列 - 行列…行列の差
- 行列 * 行列　あるいは　行列 @ 行列…行列の積
- 行列 / 行列 … 単に要素同士を除算した商を成分に持つ行列が生成
- スカラー値 * 行列…成分がスカラー倍される
- 行列.T…転置行列
- 行列.I…逆行列

matrixとarrayの乗算

- arrayとmatrixの積は、matrixとして求められる。ただし、matrixと対応する次元のサイズとarrayのサイズが一致していること
  - 例：a=np.array([1,2])　…　配列

    m= np.matrix([[2,4,],[5,6]])　…行列

    a * m　⇒　matrix([[12, 16]])　…行列として
- matrixとmatrixの積で計算する場合は、１次元配列は、転置をしないと計算されない
  - 例：amat = np.matrix(a).T

    m * amat ⇒ matrix([[10], [17]])

scipyの使用方法

- scipyは、numpyをベースにしているが、上書きしている部分もある
- 計算精度などは、scipyの方が優れており、優秀なアルゴリズムを利用していることが多い
- ライブラリの導入は、**import** scipy **as** spで、利用する
- arrayやmatrixは、numpyの方を踏襲しているが、arrayについては、numpy.arrayを使えと言われるし、matrixもnumpyのものなので、これもnumpy.arrayを使う方が無難である
- サブパッケージに関しては、以下のように、パッケージごとに名前でimportする。
  - 例：**from** scipy **import** linalg # 線形代数パッケージ

scipyのサブパッケージ

- cluster　クラスタリング・アルゴリズム
- constants　物理数学定数
- fftpack　高速フーリエ変換
- integrate　積分および常微分方程式の解法
- interpolate　内挿（補間）とスプラインによるスムーズ化
- io　入出力
- linalg　線形代数…一般に線形代数パッケージをLAPACK（Linear Algebra Package）と呼ぶ
- ndimage　N次元画像処理
- odr　直交距離回帰（Orthogonal distance regression）
- optimize　最適化および解探索ルーチン
- signal　信号処理
- sparse　疎行列および関連ルーチン
- spatial　空間データ構造とアルゴリズム
- special　特殊関数
- stats　統計分布および統計関数

- 参照：https://docs.scipy.org/doc/scipy/reference/tutorial/general.html

numpyおよびscipyで行列式を求める

- np.linalg.det(行列または２次元配列)…行列式を求めた結果を返す
- 行列式は、numpyのlinalgよりも、scipyのlinalgパッケージの方が精度が良い場合がある
- **from** scipy **import** linalg…scipyで使うとき
- linalg.det(行列または２次元配列)…行列式を返す

scipyとnumpyで逆行列

- numpyの場合
  - m = np.matrix([[1,2,3],[5,6,7],[4,9,8]])
  - m.I
  - m.I * m　# 誤差で単位行列にならない

- scipyの場合（内部でnumpyのmatrixが使われる）
  - m = sp.matrix([[1,2,3],[5,6,7],[4,9,8]])
  - **from** scipy **import** linalg
  - im = linalg.inv(m)
  - im * m　# やはり誤差で単位行列にならない

scipyによる定積分

- integrateパッケージに入っている
- romberg関数…ロンバーグ積分で求めている
  - romberg(関数, 下限, 上限, show= 論理値)

    showがTrueになっていると、途中の計算値も表示される
- quad関数…FORTRANのQUADPACKライブラリから来ている
  - 積分値, 絶対誤差 = quad(関数, 下限, 上限)

- 例：

```
from scipy import integrate
def f( x ): return x ** 3 + x ** 2 + x + 3
result = integrate.romberg( f , -4, 3, show=True)
print( result )
```

scipyによる方程式の求値

- scipy.optimizeパッケージに、割線法・ニュートン＝ラフソン法、ハレー法で求めるnewton関数などがある、
- func(x) = 0のときのxの値を求める、x0はxの初期値
  - **from** scipy **import** optimize
  - optimize.newton(func, x0) … 割線法
  - optimize.newton(func, x0, fprime=f') … ニュートン法
  - optimize.newton(func, x0, fprime=f', fprime2=f'') …ハレー法
  - オプションパラメータ「tol=許容誤差」も指定できる
  - 例：

```
from scipy import optimize
def func( x ) : return x**2 - 2
def ffunc( x ): return 2
print( optimize.newton( func, 1 ) )
print( optimize.newton( func, 1,
fprime=ffunc ) )
print( optimize.newton( func, 1,
fprime=ffunc, tol=1.0e-15 ) )
```

- その他に、scipy.optimizeパッケージに、FORTRANのLAPACKから由来のfsolve関数もある
  - optimize.fsolve(func, x0)

matplotlibのライブラリ

- **import** matplotlib.pyplot **as** plt
- 定義域 = np.linspace(開始値, 終了値, 個数, endpoint=終了値を含めるか)あるいは、np.arange( )を使っても良い
- グラフ = plt.plot(定義域の値リスト, 値域の値リスト)…グラフを表示する。

  plt.plotは、オプションパラメータとして、color="色名"や、label="系列名"などを用いることができる
- plt.setp(グラフ, color="色名")…後から属性を変えたいとき
- plt.show( )…グラフ表示ウィンドウを出す

matplotlibグラフのタイトルなど

- plt.xlabel("x軸のラベル")
- plt.ylabel("y軸のラベル")
- plt.title("グラフのタイトル")
- plt.plot(定義域, 値域, label="系列のタイトル", color="色名")
- plt.plot(定義域, 値域, マーカーの文字)
  - # {'marker': 'マーカーの文字'}で指定することも可能

    'x', 'o', '^', '--', 'bs' : 四角, '*' などの文字を指定できる
- plt.legend()　# 系列のタイトル表示
- plt.axis([xmin, xmax, ymin, ymax]) # 軸の最小値・最大値

- 参照：https://matplotlib.org/tutorials/introductory/pyplot.html

matplotlibでサブグラフを記述したいとき

- 以下のどちらかの関数で描画領域を確保する。figsize, facecolorはオプションのキーワード引数なので省略可能（横比率と縦比率は、インチサイズ）
  - fig = plt.figure(figsize=(横比率, 縦比率), facecolor="l色名")
- add_subplotで、個別のグラフ領域（ax）を追加する
  - ax = fig.add_subplot(行数, 列数, 番号) # 番号は、1番始まり
- 個別のグラフ領域（ax）に対して、plotなどでグラフを描く
  - ax.plot(x値リスト, y値リスト)
- plt.show( )で描画
- 参考：https://python-academia.com/matplotlib-multiplegraphs/

matplotlibでのサブグラフを記述法（別方法）

- subplotsを用いたサブグラフの記述
  - fig, axes = plt.subplots(行数, 列数, figsize=(横比率, 縦比率), facecolor="色名")
  - axes（個別のグラフ領域配列）は、行数×列数の2次元配列になる
- 各ax（個別のグラフ領域）に、plot関数で値をセットする
  - 概形：axes[row, col].plot(x値リスト, y値リスト)
  - 例：axes[0, 1].plot(x_array, y_array, color="blue", label="sample")
- plt.show( )で描画

個別のグラフ領域の属性設定

- 個別のグラフ領域（ax）の属性は以下のような設定が可能
  - ax.set_facecolor(色名または色指定)
    … 背景色の設定
  - ax.set_xlim([下限値, 上限値])
    … x軸の下限値と上限値の指定
  - ax.set_ylim([下限値, 上限値])
    … y軸の下限値と上限値の指定
  - ax.set_xticks(リストまたは配列)
    … x軸の目盛りの値
  - ax.set_yticks(リストまたは配列)
    … y軸の目盛りの値
  - ax.set_xticklabels(リストまたは配列)
    … x軸の目盛りのラベルの値
  - ax.set_xticklabels(リストまたは配列)
    … y軸の目盛りのラベルの値
  - ax.set_xlabel(タイトル文字列)
    … x軸のタイトル
  - ax.set_ylabel(タイトル文字列)
    … y軸のタイトル
  - ax.set_title(タイトル文字列)
    …グラフ領域のタイトル
  - ax.set_legend( )
    … 系列の凡例の表示
    loc=配置文字列　も指定可能（"upper left", "upper right", "lower left", "lower right", "upper center", "lower center", "center left", "center right", "center", "best"のいずれか)

matplotlibで使える色

- 使える色名は、右の図の通り
  - https://matplotlib.org/stable/gallery/color/named_colors.html
- タプルで指定する
  - 0以上, 1以下 の float 値の RGB または RGBA のタプル
    - 例: (0.1, 0.2, 0.5) または (0.1, 0.2, 0.5, 0.3))。
- 文字列で指定する
  - 16 進数の RGB または RGBA 文字列 (例: '#0F0F0F' または '#0F0F0F0F')。
- 文字列の縮小表現
  - 各文字を複製することで得られる 16 進数の RGB または RGBA 文字列
    - 例: '#abc'は、'#aabbcc' と同等、 '#abcd'は、'#aabbccdd' と同等
- グレーレベルは、0以上,1以下 の浮動小数点値の文字列表現
  - 例: '0.5
- 単一の文字列で色合いの短縮表記である
  - {'b'、'g'、'r'、'c'、'm'、'y'、'k'、'w'} は、それぞれ
    青、緑、赤、シアン、マゼンタ、黄、黒、白に対応する

matplotlibで使えるフォント

- matplotlibがインストールされたときに入っているフォントしか使えない
- matplotlibのフォントキャッシュを更新する方法も書かれているが、一番無難なのは、一度アンインストールして、再インストールする方法
- fontが使えるかどうかは、以下でチェックすることができる
  - **import** matplotlib.font_manager
  - matplotlib.font_manager.findfont(フォント名, rebuild_if_missing=True)
- グラフのタイトルなどを設定するときは、**fontdictを指定する
  - fontdictには、次のような指定ができる。

    {'font name': フォント名の文字列, 'fontsize': サイズ, 'fontweight': ウェイト, 'color': カラー,

    'verticalalignment': 'baseline', 'horizontalalignment': loc}
  - locはいかのどれかから{'center', 'left', 'right'}、デフォルトは、'center'
- 設定例：
  - tfont = {'fontname':'Segoe UI Historic', 'fontsize': 24}
  - ax.set_title(t, **tfont)

他のグラフのスタイル

- fig, ax = plt.subplots()などで、ax（個別のグラフ領域の確保）を得ておく
- ax.plotの代わりに、以下のような関数で描画ができる
  - 散布図

    ax.scatter(x値のリスト, y値のリスト, s=大きさリスト, c=色リスト, vmin=最小値, vmax=最大値)
  - 棒グラフ

    ax.bar(x値のリスト, y値のリスト, width=幅, edgecolor="色名", linewidth=線幅)
  - 茎図

    ax.stem(x値のリスト, y値のリスト)
  - 垂直水平だけの折れ線図

    ax.step(x値のリスト, y値のリスト, linewidth=線幅)
  - 幅のある領域図（透明度は0.0〜1.0）

    ax.fill_between(x値のリスト, 上限のy値のリスト, 下限のy値のリスト, alpha=透明度, linewidth=0)
  - 重ね図

    ax.stackplot(x値のリスト, ２次元のy値のリスト)

    2次元のy値のリストは、複数のy値のリストからnp.vstack関数を使って作る感じ
- 軸の設定（背景に8x8の線が描かれる）
  - ax.set(xlim=(0, 8), xticks=np.arange(1, 8), ylim=(0, 8), yticks=np.arange(1, 8))
- 参照
  - https://matplotlib.org/stable/plot_types/basic/index.html

matplotlibでグラフ領域に複数の範囲を持つグラフを表示

- twinx()関数を用いて、2つのグラフに対して、関連性を持たせる
  - 例：fig, ax1 = plt.subplots()

    ax2 = ax1.twinx()

- それぞれのグラフを設定する
  - 例：ax1.plot(x_list, ax1_y_list, label="plot graph")

    ax2.bar(x_list, ax2_y_list, width=25, label="bar graph") # 2つ目は棒グラフ

    ax1.set_ylim([ax1_min_value, ax1_max_value]) # 軸の範囲の設定

    ax2.set_ylim([ax2_min_value, ax2_max_value]) # 軸の範囲の設定

- 参照：https://datumstudio.jp/blog/matplotlib-2軸グラフの書き方/

sympyによる数式の指定

- **import** sympy **as** sym
- symを頭につけたくない場合は、**from** sympy **import** *

- 変数を指定する
  - x = symbols("x") … xを数式の変数に指定　varでも良い
  - var("x y z") … x, y, zを数式の変数に指定
- 数式（関数）を指定する
  - f = x ** 2 + 3 * x + 1 … 指定された変数を使って
- 用いることができる基本的な数学関数や値
  - pi, E … 円周率、自然対数の底
  - sin(x), cos(x), tan(x), cot(x), sec(x), csc(x)…三角関数
  - asin(x), acos(x), atan(x), acot(x), asec(x), acsc(x)… 逆三角関数
  - sinh(x), cosh(x), tanh(x), coth(x), sech(x), csch(x) … 双曲線関数
  - asinh(x), acosh(x), atanh(x), acoth(x), asech(x), acsch(x)… 逆双曲線関数
  - log(x), log(x, b), …対数関数
  - exp(x), Pow(x, n), sqrt(x), root(x, b)…指数関数
  - Abs(x), sign(x) , floor(x), ceiling(x), factorial(n), binomial(n, k)… 絶対値、 符号関数（-1, 0, 1 を返す）、 床関数、 天井関数、 階乗、二項係数

sympyによる方程式・不等式の指定

- 方程式
  - Eq(式, 式)…式 = 式であるような方程式を定義することができる
    - 例：eq1 = Eq(x**2 + 3, 2) # x**2 + 3 == 2
  - Ne(式, 式)…式 ≠ 式であるような方程式を定義することができる
- 不等式
  - Gt(式, 式)…式 < 式であるような不等式を定義することができる
  - Ge(式, 式)…式 ≧ 式であるような不等式を定義することができる
  - Lt(式, 式)…式 > 式であるような不等式を定義することができる
  - Le(式, 式)…式 ≦ 式であるような不等式を定義することができる

sympyによる数式演算

- xの値によって、fの値を求める
  - f.subs(x, 2)　…x = 2のときのfの値
- fの値によって、xの値を求める
  - solve(sym.Eq(f, value), x) …f = valueのときのxの値
  - solve(f, x)…f = 0のときのxの値
- 因数分解する
  - f.factor()…fの因数分解
- 式の展開
  - f.expand() …因数分解の形で書かれた式を展開する
- テイラー展開
  - f.series(n=12)…fのテイラー展開を12の次数まで求める
  - O(次数)の誤差を表示してくれるが、取るためには、removeO( )関数を使う

sympyによる数式の簡単化

- 数式を簡単にする関数…simplify( )
- 分母に平方根とか入っている分数を有理化する関数…radsimp( )
- 小数を分数の形で表現させる…nsimplify(実数値)

sympyの数式描画

- jupyterなどでは、sympyでの数式を数学の式のように描画させることが可能になっている
- sympyでの表示準備
  - **import** sympy
  - sympy.init_printing()
- IPythonでの表示関数display
  - **from** IPython.display **import** display
  - display(数式など)
- 参照：https://qiita.com/ceptree/items/3668ca52f8621b13bbc2

sympyあるある

- Qiita「sympyあるある」より
  - 参照 Webページ

    https://qiita.com/tehen_/items/b86730bfd0d98236d056

  - **from** sympy **import** * が行なわれていると仮定
- 平方根と二乗の展開をさせる
  - x = symbols("x"); print(sqrt(x ** 2))
    - sqrt(x**2)と表示されてしまう
  - y = symbols("y", positive=True);
    print(sqrt(y ** 2))
    - yと表示される
- 分数をそのまま表示させる
  - Rational(整数, 整数) あるいは

    Rational(実数)
  - print(1/2) # 0.5となってしまう
  - print(Rational(1, 2)) # 1/2と表示される
  - print(Rational(0.625)) # 5/8と表示される
  - print(Rational(0.4)) # 実数は、誤差があるため、うまく表示されない

    3602879701896397/9007199254740992

sympyによる微分・積分・定積分

- 微分形を求める
  - f.diff() …fの微分形
  - 例：

```
f1 = (x**2+3*x+2)**3 + 4*x**4 + 2*x**5

print( f1.expand() )
# x**6 + 11*x**5 + 37*x**4 + 63*x**3 + 66*x**2 + 36*x + 8
print( f1.expand().diff() )
# 6*x**5 + 55*x**4 + 148*x**3 + 189*x**2 + 132*x + 36
print( f1.diff() )
# 10*x**4 + 16*x**3 + (6*x + 9)*(x**2 + 3*x + 2)**2

f2 = sin( x ) **2 + sqrt( 3 ) *cos( x )**2
print( f2.diff() )
# -2*sqrt(3)*sin(x)*cos(x) + 2*sin(x)*cos(x)
```

- 積分形を求める
  - integrate(f, x)…fの積分形
  - 例：

```
print( integrate( f1, x ) )
# x**7/7 + 11*x**6/6 + 37*x**5/5 + 63*x**4/4 + 22*x**3 +
18*x**2 + 8*x

print( integrate( f2, x ) )
# x/2 + sqrt(3)*(x/2 + sin(x)*cos(x)/2) - sin(x)*cos(x)/2
```

- 定積分の値を求める
  - integrate(f, (x, a, b))…xが下限aから上限bまでの間のfの定積分値を求める

$$\int_a^b f(x)dx$$

  - 例：

```
print( integrate( f1, (x, 2, 4 ) ) )
# 780468/35
print( integrate( f2, (x, rad(0), rad(90) ) ) )
# pi/4 + sqrt(3)*pi/4
```

sympyによる定積分と積分について

- integrate関数を呼び出すときに、２番目のパラメータを(変数, 下限, 上限)の３つの要素から成るタプルにして求める
- 例：

```
import  sympy as sym
x = sym.symbols( "x" )
f =  x ** 3 + x ** 2 + x + 3
print( sym.integrate( f, (x, -4, 3 )))
```

- また、不定積分であれば、数式の、integrateメソッドを呼び出すことでも、実行できる
  - f.integrate( ) …xを指定する必要はない
- 不定積分は、簡単そうに見える関数でも、特殊関数を必要とする場合がある
- 例：sym.integrate(sym.cos(x**2), x)　…flesnelc関数とgamma関数を必要とする

$$\frac{\sqrt{2}\sqrt{\pi}C\left(\frac{\sqrt{2}x}{\sqrt{\pi}}\right)\Gamma\left(\frac{1}{4}\right)}{8\Gamma\left(\frac{5}{4}\right)}$$

  - fresnelc関数…https://docs.sympy.org/latest/modules/functions/special.html#sympy.functions.special.error_functions.fresnelc
  - gamma関数…https://docs.sympy.org/latest/modules/functions/special.html#module-sympy.functions.special.gamma_functions

方程式の解・連立方程式の解

- 方程式の解

```
from sympy import *
var('x, a, b')
sol=solve (a*x+b, x)
init_printing()
display(sol[0])
```

- Eq関数で、方程式を作成することが可能
  - 例：

```
eq = Eq( 2*x**2+2*x+3, 4 )
```

$2x^2 + 2x + 3 = 4$

- 連立方程式の解
  - solve(方程式のリスト, 変数のリスト　)で求めることができる
  - 例：

```
import sympy as sym
sym.var('x, y, a, b, c, d, e, f')
eq1=sym.Eq(a*x+b*y, e)
eq2=sym.Eq(c*x+d*y, f)
sym.solve ([eq1, eq2], [x, y])
```

$\left\{x : \frac{-bf + de}{ad - bc}, y : \frac{af - ce}{ad - bc}\right\}$

- 連立方程式は、一次でなくても構わない。

```
eqs = [ Eq( 3*x**2+2*y**2, 72), Eq(x+y,16) ]
print( solve( eqs, [x, y] ) )
```

$\left[\left(\frac{32}{5} - \frac{14\sqrt{6}i}{5}, \frac{48}{5} + \frac{14\sqrt{6}i}{5}\right), \left(\frac{32}{5} + \frac{14\sqrt{6}i}{5}, \frac{48}{5} - \frac{14\sqrt{6}i}{5}\right)\right]$

- 参考Webページ

  https://pianofisica.hatenablog.com/entry/2019/04/04/233515

不等式の解・連立不等式の解

- 条件・範囲を示す関数がいくつか用意されている
  - Gt(a, b), Lt(a, b), Ge(a, b), Le(a, b), Ne(a, b) …比較関数
  - And(x, y), Or(x, y), Not(x) …論理関数
- 例：

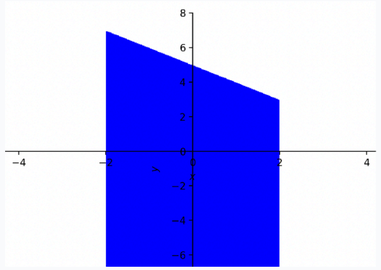
```
ineq1 = Ge(x, 2)  # x >= 2
ineq2 = Le(y, 5)  # y <= 5
ineq3 = Eq(x + y, 7)  # x + y = 7
ineq4 = Ne(x, y)  # x ≠ y
```

- 連立不等式は、1つの変数の解しか求められない
- 例：

```
ineq_xy = [ Gt( 2*x+7*y, 12 ), Lt( -3*x+2*y, -12) ]
print( solve(ineq_xy, [x] ) )
# (x > 6 - 7*y/2) & (x > 2*y/3 + 4)
```

- 連立不等式で、解の範囲を求めるには、sympy.plotting.plot_implicitを利用する
- 例：

```
from sympy.plotting import plot_implicit
ineq1 = x**2 - 4 < 0  # x > -2 and x < 2
ineq2 = y + x < 5     # y < 5 - x

# 解の範囲のプロット

plot_implicit(And(ineq1, ineq2), (x, -5, 5), (y, -10, 8))
```



sympyで偏微分・3次元ベクトル

- 偏微分は、diff(変数)あるいはdiff(変数, 変数, ...)で求めることができる。
- 例：
  - tf1 = sin(x) * cos(y) * z**4
  - print(tf1.diff(x))
  - print(tf1.diff(z))
  - print(tf1.diff(x, y))
  - print(tf1.diff(x, y, z))
- 以下の3次元ベクトルで用いられる勾配、発散、回転は、3次元座標系において求めるので、3次元座標を定義する必要がある。
  - 例： N = CoordSys3D('N')
  - # 3次元の座標系を定義（文字列は何でも良い）
- 3次元の座標系を定義すると、単位ベクトルと成分値が記述できる
  - N.i, N.j, N.k…それぞれの方向の単位ベクトル
  - N.x, N.y, N.z…それぞれの方向の成分値
- 勾配は、sympy.vectorモジュールのgradient(f)で求まる、

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

- 発散は、sympy.vectorモジュールのdivergence(vector)で求まる。

$$\nabla \cdot \mathbf{F} = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z}$$

- 回転（rot）は、sympy.vectorモジュールのcurl(vector)で求まる。

$$\nabla \times \mathbf{F} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ F_x & F_y & F_z \end{vmatrix}$$

- 例：

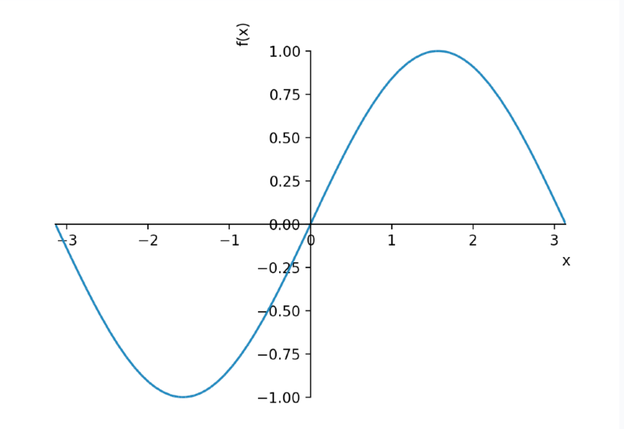
```
from sympy import *
from sympy.vector import *
N = CoordSys3D('N')  # 3次元の座標系

x, y, z = symbols('x y z')

f = N.x**2 + N.y**2 + N.z**2  # 関数を定義

print(gradient(f))
# 2*N.x*N.i + 2*N.y*N.j + 2*N.z*N.k

v = N.i*N.x*N.y+N.j*N.z*N.y+N.k*N.x*N.z # ベクトルを定義

print( divergence(v) )
# N.x + N.y + N.z
print( curl(v) )
(-N.y)*N.i + (-N.z)*N.j + (-N.x)*N.k
```

sympyの関数のプロット

- plottingモジュールのplot関数でプロットすることができる
- plot(関数, (変数, 下限, 上限))

- 例：

```
import sympy as sym
import sympy.plotting as plt
x = sym.symbols( "x" )
f = sym.sin( x )
plt.plot( f, (x, -sym.pi, sym.pi ) )
```



陰関数・パラメトリック関数曲線のプロット

- import指定
  - **from** sympy.plotting **import** * を仮定
- 陰関数
  - x**2 + y**2 = r**2より、r=2として

    f = x ** 2 + y ** 2 - 2 ** 2
  - plot_implicit(f, (x, -2, 2), (y, -2, 2))
  - 次の関数もプロットしてみる
  - f = x ** 2 + y ** 2 + x/y - 2 ** 2
    …弧状のグラフになる
  - f = 2 * x ** 2 + 5 * y ** 2 - 3 * x*y - 2 ** 2
    …斜めの楕円グラフになる
- パラメトリック関数
  - t = symbols("t")
  - plot_parametric(cos(t), sin(t), (t, 0, 2*pi))

sympy.plottingで複数の関数のグラフを表示する方法

- 変数 = plt.plot(...., show= False)にして代入しておく、代入後はリストになっている
- 変数にappend( )で追加する（追加するグラフもリストなので、先頭の要素をインデックスで指定する）
- 最後に、最初のグラフをshow()関数で表示させる

- 例：

```
import sympy.plotting as plt
p1 = plt.plot( sin( x ) + cos( x ), (x, -pi, pi), show=False )
p2 = plt.plot( sin(x), (x, -pi, pi ), line_color="r", show=False )
p1.append(p2[0])
p1.show()
```

sympyでサブグラフとして描画する

- 同じように、plotする際に、show=Falseの指定をしておく
- PlotGrid関数を用いる
  - PlotGrid(行数, 列数, プロットオブジェクトのシーケンス)
- 例：

```
import sympy.plotting as plt
p1 = plt.plot( sin( x ) + cos( x ), (x, -pi, pi), show=False )
p2 = plt.plot( sin(x) * cos( x ), (x, -pi, pi ), line_color="r", show=False )
plt.PlotGrid(1, 2, p1, p2)
```

sympyによる行列クラスの演算

- **from** sympy **import** *
- Matrixクラスのオブジェクトを作成
  - 書式：
    - Matrix(入れ子型のリストやタプル)
    - Matrix(行数, 列数, 1次元のリストやタプル)
  - 例：
    - Matrix([[1,0,0], [0,0,0]])
    - Matrix([[1, 2, 3]])
    - Matrix(2, 3, [1, 2, 3, 4, 5, 6])
    - Matrix(3, 4, **lambda** i, j: 1 - (i+j) % 2)
- 正方行列でzero行列、1行列、単位行列を作成
  - zeros(次数), ones(次数), eye(次数)
  - diag(対角成分の並び) … 対角行列を作成、並びの成分にはMatrixのオブジェクトも可能
- 基本的には、scipy, numpyと同様の関数が利用することができるが、詳しくは、以下を参照
  - https://showa-yojyo.github.io/notebook/python-sympy/matrices.html

sympyの行列の要素のアクセス

- 以下で、Mは、sympyのMatrixオブジェクトを示す
- インデックスによるアクセス
  - 1次元：M[idx]
  - 2次元：M[row, col]
- スライスによるアクセス
  - M[start: end] -- start endは省略可能
  - M[start: end: step] -- stepがある場合
  - 右辺にある→コピー、左辺にある→ビュー
- 各行・各列にアクセス
  - M[row, :] … row番目の行ベクトル
  - M[:, col] … col番目の列ベクトル
  - M[r1:r2, c1:c2] … 部分的な行列
- 各要素に関数を適用
  - M.applyfunc(関数名あるいは無名関数)
- 要素の個数を求める
  - len(M) …すべての要素の総数
  - len(M[:, 0])…行数
  - len(M[0,:])…列数

sympyの行列に対する変更演算

- 以下で、M, Nは、sympyのMatrixオブジェクトを示し、n, mは列や行のインデックスを示す
- 連結
  - 複数の行列を横に連結した行列を生成：M.hstack(M, N, ...)
  - 複数の行列を縦に連結した行列を生成：M.vstack(M, N, ...)
  - 行列に列を挿入した行列を生成：M.col_insert(n, N)
  - 行列に行を挿入した行列を生成：M.row_insert(m, N)
  - 行列同士を縦に連結した行列を生成（下に連結）：M.col_join(N)
  - 行列同士を横に連結した行列を生成（右に連結）：M.row_join(N)
  - 行を指定の置換で入れ替えた行列を生成：M.elementary_row_op(op="n<->m", row1=m, row2=m))
  - 列を指定の置換で入れ替えた行列を生成：M.elementary_col_op (op="n<->m", col1=m, col2=m)
- 列空間・行空間
  - 行列を列ベクトルに分解した列空間（行列の列ベクトルによって張られるベクトル空間）のリストを返す：M.columnspace( )
  - 行列を行ベクトルに分解した行空間（行列の行ベクトルによって張られるベクトル空間）のリストを返す：M.rowspace( )

sympyの行列演算子・ベクトル演算

- 以下で、M, Nは、sympyのMatrixオブジェクトを示す
- 演算子
  - 要素ごとの加算・減算：M + N, M - N
  - 行列の積：M*N, M@N　-- 演算子では要素ごとの積（アダマール積）はない
  - べき乗：M**n
  - 逆行列：M**-1
- ベクトル演算（ベクトルは、行/列ベクトルでベクトルを表現）
  - クロス積（外積）：M.cross(N)
  - ドット積（内積）：M.dot(N)
  - 行ベクトルと列ベクトルの積：（行列の積）*, M.multiply(N)
  - 列ベクトルと行ベクトルの積：（行列の積）*, M.multiply(N)
  - 要素ごとの積：M.multiply_elementwise(N)
  - 正規化（大きさを1に）：M.normalized( ) -- Mは行/列ベクトルに限る
  - 直交化基底：GramSchmidt([M, N, ...], orthonormal=False) -- M, Nは行/列ベクトルに限る, orthonormal=Trueにすると基底ベクトルが正規化（大きさを1に）される

sympyの行列演算

- 以下で、M, Nは、sympyのMatrixオブジェクトを示し、i, jは、行列のインデックスを示す。
- 転置行列
  - M.T あるいはM.transpose( )
- 乗算
  - 行列の積：M.multiply(N), M*N, M@N
  - 要素ごとの積：M.multiply_elementwise(N)
- ノルム
  - ノルム：M.norm( )
    - 行列はFrobeniusノルム
    - ベクトルの場合は、2ノルム
- 行列式
  - 正方行列：M.det( )
  - Bareisのアルゴリズムで：M.det(method="bareiss")
  - LU分解で行列式を求める：M.det_LU_decomposition()
- 余因子
  - 指定がないと、Berkowitzのアルゴリズムで求める形になる。
  - 余因子行列：M.adjugate( )
  - 余因子行列（転置なし）：M.cofactorMatrix( )
  - 余因子：M.cofactor(i, j)
  - 小行列：M.minorMatrix(i, j)
- 逆行列
  - M.inv(method=None)
    - methodのキーワード
      - "GE": Gaussの消去法による（デフォルト）
      - "LU": LU分解による
      - "ADJ": 余因子行列と行列式で求める