

# Fortran, C ユーザに向けた 実践的 Python プログラミング (2018年4月8日改訂)

本田 宏明<sup>1,2)</sup>

<sup>1)</sup> 九州大学 情報基盤研究開発センター

<sup>2)</sup> JST-CREST

# 環境設定、配布ソフトについて

1. 66, A1-A5 頁を確認下さい.
2. Bourne 系シェルであることを前提にしています.  
A1 頁のセットアップ作業中に,  
“`..profile`” を実行することを忘れないようにして下さい.

# あらすじ

- 講習会の目的
- Python 言語について
- Python の基本的なプログラミング
- 逐次プログラミング
- 並列プログラミング
- Fortran や C 言語による拡張
- 付録
  - 配布ソフト
  - Python, 他のライブラリのインストール

# 講習会の目的

1. Fortran や C の知識をベースにして, Python の基本的な文法を知る.
2. Python を利用したテキスト処理や数値計算が出来るようになる.
3. MPI 並列計算が簡単に実行可能であることを知る.
4. 自身の Fortran, C プログラムを Python から利用できるようになる.

## □ 以下は自習にお任せします<sup>1)</sup>

- 種々のエラーメッセージの理解
- モジュールと名前空間
- オブジェクト指向, Python でのクラスの手取り
- ctypes 以外の種々の Fortran, C コード呼び出し方法
- Cython を利用した高速計算
- インタラクティブモードや IPython での計算
- Python2 と Python3 の違い (本講習ソフトは 2.7 と 3.6 両方にて確認)

- Ruby と Python は兄弟みたいなものなので、本は必要ないのではないかと思います。
- 以下のリンクを参照下さい。

1) <http://docs.python.jp/2.7/reference/index.html>  
<http://docs.python.jp/3.6/reference/index.html>

# 1. Python 言語について

## □ Python 言語について

- 汎用プログラミング言語
- 動的型付け, ガベージコレクション, オブジェクト指向
- ポータブル (Unix系, OSX, Windows)
- オープンソース
- 利点
  - コード可読性に配慮
    - インデントによるコードブロック形成
  - 豊富な標準ライブラリ
    - sys, re, subprocess, ...
  - 豊富なThird party によるライブラリ
    - NumPy, SciPy, mpi4py, ...

## 2. Python の基本的なプログラミング

# 基本2.1: Hello World 型プログラム

## hello.0.py プログラム内容

```
myname = "Kyudai Taro"
age     = 25
height  = 1.705
weight  = 60.16
formatted_weight = "weight: %.1f" % weight

print( weight )
print( formatted_weight )
print( "Hello, %s." % myname )
print( "Your age, height, and weight are %2d, and %.3f m, %.1f kg, respectively."
      % (age, height, weight) )
```

- 変数を書式付き文字列へ変換  
val = “書式” % 変数 or (変数, ... )

## プログラム実行

```
$ python hello.0.py
```

## 出力結果

```
60.16
weight:  60.2
Hello, Kyudai Taro.
Your age, height, and weight are 25,  1.705 m, and  60.2 kg, respectively.
```

- 変数には種々の型のデータを保持可能
- 出力には print 関数を利用
  - 変数をそのまま引数に渡す or
  - 書式つき文字列変数に変換して渡す
    - %d: 整数
    - %f, %e: 浮動小数点数
    - %s: 文字列 or オブジェクト内容

# 基本2.2: 繰り返し

hello.1.py プログラム内容

```
for i in range( 5 ):
    print( "%16.4e" % (float( i ) / 3) )

array = [5.7, 7.8, 9.10, 7.95]
print( "Length of array: %d" % len( array ) )
for v in array:
    print( "%16.4f" % (v - array[ 0 ]) )
```

出力結果

```
0.0000e+00
3.3333e-01
6.6667e-01
1.0000e+00
1.3333e+00
Length of array: 4
0.0000
2.1000
3.4000
2.2500
```

- range() 関数は整数引数値未満までの整数リストを作成

range( 5 ) => [ 0, 1, 2, 3, 4 ]

- 繰り返しは for で作成 (while もあり)
- in 以下の内容から一つずつ要素を取り出して添字変数に設定する
- **for の行末に ":" コロンを忘れない !**
- **ループボディ部はインデントする !**

- リストは [..., ..., ] により初期化
- array[ i ] の様にアクセス
- len() 関数により要素数を取得可能



## 基本2.3: 条件分岐

hello.2.py プログラム内容

```
i = int( input('Input an integer: ') )  
if ( ( i % 2 ) == 0 ):  
    print( "Even: %d" % i )  
else:  
    print( "Odd : %d" % i )
```

出力結果

```
Input an integer: 89  
Odd:      89
```

- input() 関数によりキーボード入力を取得
  - 関数戻り値は文字列
- 条件分岐文：  
    if ... :  
    else:
- 条件分岐ボディ部はインデントする！

キーボードから入力した

# 基本2.4: 関数の定義

hello.3.py プログラム内容

```
def sum1( arr ):  
    s_even = 0  
    s_odd  = 0  
    for i in range( len( arr ) ):  
        if ( (i % 2) == 0 ):  
            s_even = s_even + arr[ i ] * 2  
        else:  
            s_odd  = s_odd  + arr[ i ] * 3  
    return s_even, s_odd  
  
array = [5.7, 7.8, 9.10, 7.95]  
s_even, s_odd = sum1( array )  
  
sys.stdout.write( "test: " )  
print( "sum1 of %s -> %8.2f,%8.2f" % ( array, s_even, s_odd ) )
```

出力結果

```
test: sum1 of [5.7, 7.8, 9.1, 7.95]  
->    29.60,    47.25
```

- 関数の定義：

```
def 関数名( 引数1, ... ):  
    return 変数値
```

- プログラム構造が複雑な時には、**インデントルール**に従った記述を間違えないように**注意！**

- 関数からは、  
**return 変数1, 変数2, ...**  
と、複数の値（多値）を返し、  
呼び出し側で複数の値を受ける事が可能

# インデントルール

- コードブロックはインデントにより表現！！！！

```
def sum1( arr ):  
    s = 0  
    for i in range( len( arr ) ):  
        if ( (i % 2) == 0 ):  
            s = s + arr[ i ] * 2  
        else:  
            s = s + arr[ i ]  
    return s
```

- 空白部分にタブが入っている場合があるので注意！
- Emacs ではタブを使わない設定にし、vi ではエディタに空白を挿入するコマンドを用意することをオススメ

### 3. 逐次プログラミング

1. 簡単 sed プログラム
2. 簡単 awk プログラム
3. NumPy を利用した線形計算

# 課題3.1: 簡単 sed

## 課題

ファイル名 in.template 中の “\_\_ZZ\_\_” の文字列を,  
”7.4”, “8.4”, “9.4” の3種類の文字列に置換し,  
それぞれ, in.7.4, in.8.3, in.9.4 のファイルに出力する

in.template

```
3
7.8 4.9 8.9
5.3 3.4 __ZZ__
9.8 7.9 -2.9
```

in.7.4

```
3
7.8 4.9 8.9
5.3 3.4 7.4
9.8 7.9 -2.9
```

in.9.4

```
3
7.8 4.9 8.9
5.3 3.4 9.4
9.8 7.9 -2.9
```

in.8.3

```
3
7.8 4.9 8.9
5.3 3.4 8.3
9.8 7.9 -2.9
```

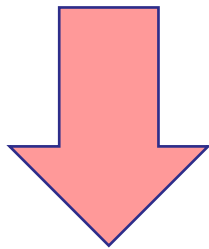
# プログラム作成の全体方針

## 全体方針

in.template を入力として、1つの置換を実施するための sbst.py を作成

簡易 sed (sbst.py) の作成：

1. コマンドライン引数を2 or 3つ取得  
\$ python sbst.py src dst [infile]
  1. 2つ未満の場合にはエラー
  2. 2つの場合には、標準入力を入力ファイルとする。  
3つある場合には、3つ目を入力ファイル名とする
2. ファイルを1行1行読み込み
3. 指定された文字列を検索
4. 指定された文字列が見つかった場合、指定された置換文字列に



sbst.py により、以下のように 7.4 のファイルを作成可能とする

```
$ python sbst.py __ZZ__ 7.4 in.template > in.7.4
```

上の sbst.py を利用して複数(7.4, 8.3, 9.4) の置換出力を作成

- 方法 1 : Shell から sbst.py を利用  
方法 2 : Python から sbst.py を利用

# sbst.py プログラム (1/3)

## sbst.py 全体プログラム

```
import sys, re

if len( sys.argv ) <= 2:
    sys.exit( "Error: len(args) <= 2: args: %s" % sys.argv )

rstr = sys.argv[ 1 ]
sstr = sys.argv[ 2 ]

if len( sys.argv ) <= 3:
    fin = sys.stdin
else:
    filename = sys.argv[ 3 ]
    fin = open( filename, "r" )

#####
for line in fin:
    iline = line[:-1]
    oline = re.sub( rstr, sstr, iline )
    print( oline )

#####
if len( sys.argv ) > 3:
    fin.close()
```

青：0. モジュールの読み込み  
1. 引数処理  
2. ファイル処理  
赤：1. 標準入力もしくは  
ファイルから行を読み込み  
2. 検索文字列を見つけ、置換

行の '#' 以降はコメント

# sbst.py プログラム (2/3)

## sbst.py 前半部分

```
import sys, re

if len( sys.argv ) <= 2 :
    sys.exit( "Error: len(args) <= 2: args: %s" % sys.argv )

rstr = sys.argv[ 1 ]
sstr = sys.argv[ 2 ]

if len( sys.argv ) <= 3:
    fin = sys.stdin
else:
    filename = sys.argv[ 3 ]
    fin = open( filename, "r" )

# fin (ファイルオブジェクト) を
# 利用してデータを取得

if len( sys.argv ) > 3:
    fin.close()
```

sys, re モジュールの読み込み

- sys: システム関係, 今回は引数の取り扱い
- re: 正規表現

コマンドライン引数の取り扱い:  
以下の様に実行した場合は,

```
$ python sbst.py "__ZZ__" 7.4 in.template
```

sys.argv 文字列リストに以下の値が設定される

```
sys.argv[ 0 ] => "sbst.py"
sys.argv[ 1 ] => "__ZZ__"
sys.argv[ 2 ] => "7.4"
sys.argv[ 3 ] => "in.template"
len( sys.argv ) => 4
```

- ファイルオブジェクト取り扱い  
標準入力先 : sys.stdin  
ファイルオープン : fileobj = open( "ファイル名", "r" )  
ファイルクローズ : fileobj.close()
- sys.stdin はファイルオブジェクトとしても利用可能



# sbst.py プログラム (3/3)

for と ファイルオブジェクトを利用  
1. ファイルから一行一行読み込んで,  
2. line 変数に格納  
3. ファイル終端にて停止

sbst.py 後半部分

```
for line in fin:  
    iline = line[:-1]  
    oline = re.sub( rstr, sstr, iline )  
    print( oline )
```

line[:-1] にて, line 文字列変数の最後の文字以外を取得,  
iline 変数に代入 (perl の chop() に対応)

re 正規表現ライブラリを利用  
re.sub( “検索文字列”, “置換文字列”, “置換対象文字列” )  
=> “置換後文字列”

よくわからない場合には,  
line, iline, oline の変数をループ内で出力してみると良い

# Shell からの sbst.py の利用

Bourn シェルによる 3 種類の置換ファイル生成

```
#!/bin/sh

PYTHON=python2

Zs='7.4 8.3 9.4'

for Z in $Zs
do
    echo "$PYTHON sbst.py __ZZ__ $Z in.template > in.$Z"
    $PYTHON sbst.py __ZZ__ $Z in.template > in.$Z
done
```

Bourn シェルによる繰り返し：

```
for 変数 in 変数候補
do
    $変数 を利用したシェルプログラム
done
```

# Python からサブプロセスを実行

Python からのサブプロセス実行し sbst.py により 3 種類の置換ファイル生成

```
import subprocess
```

```
Zs = [ "7.4", "8.3", "9.4" ]
```

```
comm = []
```

```
comm.append( "python" )
```

```
comm.append( "sbst.py" )
```

```
comm.append( "__ZZ__" )
```

```
comm.append( "" )
```

```
comm.append( "in.template" )
```

```
for i in range( len( Zs ) ):
```

```
    comm[ 3 ] = Zs[ i ]
```

```
    print( "comm: %s" % comm )
```

```
    filename = "in.%s" % Zs[ i ]
```

```
    os = open( filename, "w" )
```

```
    subprocess.call( comm, stdout=os )
```

```
    os.close()
```

subprocess モジュールをインポート

- サブプロセス実行のための、コマンド文字列リストを作成.
- 以下のコマンドを実行したい場合,

```
$ python sbst.py __ZZ__ 7.4 in.template
```

以下の文字列リスト comm を作成する

```
[ "python", "sbst.py", "__ZZ__", "7.4", "in.template" ]
```

- 今回は、comm を作成する際に空リストを作成した後、append() 関数を利用して要素を追加

- Comm の 3 番目の要素となるはずの、  
"7.4", ... の文字列はループの中で Zs から作成し、挿入

- subprocess.call( comm, stdout=os ) により、comm のコマンドを実行、この際、stdout=os により、os ファイルオブジェクトに出力可能
- 出力ファイルオブジェクト os は、open( "ファイル名", "w" ) により生成

# 課題3.2: 簡単 awk

## 課題

ファイル "input" の中から、以下の3つの情報を取得し、2. と 3. の値を出力する。

1. NUMB\_AO 値
2. TOTAL ENERGY 値
3. EIGEN VECTOR 1～4 (NUMB\_AO 次元)

"input"

```
...
NUMB_AO, NSIZE_FOCK :      39      780
...
TOTAL ENERGY =      -227.9533603374
...
EIGEN VECTOR:
          1          2          3          4
E -20.62037342 -20.53632775 -20.53231522 -1.41791218
1  0.00000537  0.99579190 -0.00044285 -0.00773777
2  0.00004793  0.02221927 -0.00006103  0.01727343
...
39  0.00035687  0.00030065 -0.00228570 -0.00553780
```

NUMB\_AO 次元行読込  
( 39 行 )

標準出力へ



```
ene: -2.2795e+02
1:  0.00000537  0.00004793 ...  0.00035687
2:  0.99579190  0.02221927 ...  0.00030065
3: -0.00044285 -0.00006103 ... -0.00228570
4: -0.00773777  0.01727343 ... -0.00553780
```

NUMB\_AO 列出力  
( 39 列 )

# awk 型プログラム main パート (1/3)

プログラム main パートの全体 (awk.py)

```
line = find_line( sys.stdin, "NUMB_AO, NSIZE_FOCK" )
strs = line[:-1].split()
na    = int( strs[ 3 ] )
print( "na:  %s : %d" % ( strs, na ) )
```

“NUMB\_AO” 値の読み込み

```
#####
strs = find_line( sys.stdin, "TOTAL ENERGY =" )[:-1].split()
ene  = float( strs[ 3 ] )
print( "ene: %s : %e" % ( strs, ene ) )
```

“TOTAL ENERGY” 値の読み込み

```
#####
```

```
line = find_line( sys.stdin, "EIGEN VECTOR:" )
line = skip_lines( sys.stdin, 2 )
#strs = line[:-1].split()
#print( "%s" % ( strs ) )
```

“EIGEN VECTOR” の読み込み

```
lines = read_split_lines( sys.stdin, na )
#for i in range( len( lines ) ):
#    print( "%s" % lines[ i ] )
```

# awk 型プログラム main パート (2/3)

“input”

```
...  
NUMB_AO, NSIZE_FOCK :      39      780  
...  
    TOTAL ENERGY =    -227.9533603374  
...
```

プログラム main 部分の前半 (awk.py)

```
line = find_line( sys.stdin, "NUMB_AO, NSIZE_FOCK" )  
strs = line[:-1].split()  
na    = int( strs[ 3 ] )  
print( "na:  %s : %d" % ( strs, na ) )
```

```
#####  
strs = find_line( sys.stdin, "TOTAL ENERGY =" )[:-1].split()  
ene  = float( strs[ 3 ] )  
print( "ene: %s : %e" % ( strs, ene ) )
```

1. 標準入力を読み込みつつ、自作関数 `find_line()` を用いて、“NUMB\_AO, ...” を検索、その行を `line` 変数値に返す
2. `line` の改行文字の削除後に空白で分割、3 番目の文字列 “39” を `int` で変数 `na` に保存

1. “TOTAL ENERGY =” を検索、その行を読み込み、空白で分割
2. 同じく 3 番目の文字列を `float` で保存

# awk 型プログラム main パート (3/3)

“input”

```
...
EIGEN VECTOR: ----->
      1          2          3          4
E -20.62037342 -20.53632775 -20.53231522 -1.41791218
1  0.00000537  0.99579190 -0.00044285 -0.00773777
2  0.00004793  0.02221927 -0.00006103  0.01727343
...
39 0.00035687  0.00030065 -0.00228570 -0.00553780
```

“EIGEN VECTOR” を検索  
2行スキップ

na 行読み込みながら進む。  
読み込んだデータを2次元リストに

プログラム main パートの後半 (awk.py)

```
line = find_line( sys.stdin, "EIGEN VECTOR:" )
line = skip_lines( sys.stdin, 2 )
#eigen_values_str = line[:-1].split()
#print( "%s" % eigen_values_str )

eigen_vectors_str = read_split_lines( sys.stdin, na )
for i in range( 4 ):
    sys.stdout.write( "%4d:" % ( i + 1 ) )
    for j in range( na ):
        sys.stdout.write( "%13s" % eigen_vectors_tr_str[ j ][ i + 1 ] )
    sys.stdout.write( "¥n" )
```

1. “EIGEN VECTOR:” を検索
2. 自作関数 skip\_lines() にて2行スキップ、スキップ後の行を line に返す
3. 自作関数 read\_split\_lines() を利用し、na 変数値分だけ行を読み込み、読み込んだ行を全て分割して、2次元リスト lines に返す
4. lines を出力

# awk 型プログラム各関数の定義 (1/2)

awk.py の関数定義部前半

```
import re

#
def find_line( fin, str ):
    r = re.compile( str )
    for line in fin:
        if ( r.search( line ) ):
            break
    return line
```

1. `re.compile` 関数で検索文字列をコンパイルし、コンパイル済み検索文字オブジェクトとして `r` 変数に返す (高速検索のため)
2. ファイルオブジェクト `fin` から 1 行 1 行読み込むループを形成
3. ループ内において、  
現在読み込まれた行に検索文字列が含まれるか検索、  
この際に `r` に付随する関数 `search()` を利用
4. 文字列が存在していた場合には `break` し、  
その行を返す



# awk 型プログラム各関数の定義 (2/2)

## awk.py の関数定義部後半

```
def skip_lines( fin, n ):  
    for i in range( n ):  
        line = fin.readline()  
    return line  
  
#  
def read_split_lines( fin, n ):  
    lines = []  
    for i in range( n ):  
        line = fin.readline()  
        strs = line[:-1].split()  
        lines.append( strs )  
    return lines
```

1. N 回ループを繰り返し
2. ループ内において、ファイルオブジェクト fin から 1 行ずつ読み込み、line 変数に返す
3. N 行読み終わったら、現在の line 変数を返す

1. lines リストを空で初期化
2. n 回ループを繰り返し
3. ループ内において、ファイルオブジェクト fin から 1 行ずつ読み込み、line 変数に返す。  
line を分割する。
4. lines リストに分割された line を追加
5. n 行読み終わったら、現在の line 変数を返す

# ファイルオブジェクトからの入力まとめ

## □ 文字列の空白での分割

`strs = str.split()` : "aaa bbb ccc" => [ "aaa", "bbb", "ccc" ]

## □ ファイルオブジェクトから 1 行読み込み, 分割

入力

aaa 10 10.76

```
line = fileobj.readline()
str = line[:-1]
strs = str.split()
C1 = int( strs[ 1 ] )
C2 = float( strs[ 2 ] )
```

 以下の様にも書ける

```
strs = fileobj.readline()[:-1].split()
C1 = int( strs[ 1 ] )
C2 = float( strs[ 2 ] )
```

ファイルから全ての行を読む場合には以下の方が良い  
for line in fileobj:  
 strs = line[:-1].split()

" aaa 10 10.76¥n" (str, strs 変数内容)  
" aaa 10 10.76¥n" => " aaa 10 10.76"  
" aaa 10 10.76" => [ "aaa", "10", "10.76" ]

- 標準入力から読む場合には：  
`fileobj => sys.stdin`
- ファイル名からファイルを開いて、  
ファイルオブジェクトを取得するには：  
`fileobj = open( filename, "r" )`
- `open` をした場合には、  
`fileobj.close()` を忘れない。

# ファイルオブジェクトへの出力まとめ

## ■ 基本の改行付き出力

- `print( obj )`
- `obj` として大抵の変数, リストを対象
  - 自身で作成したクラスでは `__repr__` 関数を定義する必要あり
  - 改行を抑える方法もあり

## ■ ファイルオブジェクトへの出力 (`printf` に似てる)

- `fileobj.write( “%s¥n” % obj )`  
`fileobj.write( “%s¥n%s¥n”, % (obj1, obj2) )`
- : `obj` を出力し改行  
: `obj1` を出力し改行し  
   `obj2` を出力し改行
- `write()` 関数の引数は文字列が一つ
  - 改行は付かないため, 自分で “¥n” を記述
  - 書式と変数の間は “,” でなくて “%” !!!

- 標準出力, 標準エラー出力へは上記の `fileobj` 変数を以下の様に変更する:  
標準出力: `sys.stdout`, 標準エラー出力: `sys.stderr`
- ファイル名からファイルを開いて, ファイルオブジェクトを取得するには  
`fileobj = open( filename, “w” )`.
- `open` した場合には `fileobj.close()` を忘れない

# NumPy と SciPy

## □ NumPy

- 配列や行列の演算を高速に実行可能とするライブラリ
- Python のリストに比較して大量のデータに対し、高速なアクセスが可能

## □ SciPy

- NumPy の配列や行列演算に加え、種々の科学技術計算を可能とするライブラリ
- 数学アルゴリズムや統計計算などの関数群を NumPy の拡張として提供

- これらのライブラリは標準では入っていない
- 自身でインストールが必要
- (付録参照)

# 課題3.3.1: 線形計算 (固有値問題)

## 課題

以下の行列 A, B から, 実対称正定値一般固有値問題 ( $Ax = \lambda Bx$ ) を計算する.

入力 input\_eig

A	4			
	0.24	0.39	0.42	-0.16
	0.39	-0.11	0.79	0.63
	0.42	0.79	-0.25	0.48
B	-0.16	0.63	0.48	-0.03
	4.16	-3.12	0.56	-0.10
	-3.12	5.03	-0.83	1.09
	0.56	-0.83	0.76	0.34
	-0.10	1.09	0.34	1.18

対象となる行列

A:				
	0.24	0.39	0.42	-0.16
		-0.11	0.79	0.63
			-0.25	0.48
				-0.03
B:				
	4.16	-3.12	0.56	-0.10
		5.03	-0.83	1.09
			0.76	0.34
				1.18

出力

	0	1	2	3
E	-2.2254	-0.4548	0.1001	1.1270
0	-0.0690	-0.5740	-1.5428	1.4004
1	0.3080	0.5329	-0.3496	-0.6211
2	-0.4469	-0.0371	0.0505	0.4743
3	-0.5528	-0.6766	-0.9276	0.2510

# 固有値問題解法プログラム (1/3)

## 全体プログラム構成

```
#1
# モジュール sys, numpy, scipy を読み込み
# モジュール linalg を scipy から更に読み込み

#2
# スカラー整数値 ndim の読み込み
# ndim * ndim 次元の NumPy の 2 次元配列として空の A, B を生成
# 標準入力から入力データを読み込み, A, B に設定

#3
# SciPy のライブラリ linalg.eigh() を利用して対角化,
# 返り値として固有値と固有ベクトルを取得

#4
# 出力
```

# 固有値問題解法プログラム (2/3)

## プログラム前半

```
#1
# モジュール sys, numpy, scipy を読み込み
# モジュール linalg を scipy から更に読み込み
import sys, numpy, scipy
from scipy import linalg

#2
# スカラー整数値 ndim の読み込み
strs = sys.stdin.readline()[:-1].split()
ndim = int( strs[ 0 ] )
print( "ndim = %4d" % ( ndim ) )

# ndim * ndim 次元の NumPy の 2 次元配列として空の A, B を生成
A = numpy.empty( [ ndim, ndim ], dtype=numpy.float64 )
B = numpy.empty( [ ndim, ndim ], dtype=numpy.float64 )

# 標準入力から入力データを読み込み, A,B に設定
for icol in range( ndim ):
    strs = sys.stdin.readline()[:-1].split()
    for irow in range( ndim ):
        A[ icol ][ irow ] = float( strs[ irow ] )

... (B についても同じ)
```

# 固有値問題解法プログラム (3/3)

## プログラム後半

#3

# SciPy のライブラリ `linalg.eigh()` を利用して対角化<sup>1)</sup>,  
# 返り値として固有値と固有ベクトルを取得

```
evals, evecs = scipy.linalg.eigh( A, b=B, lower=False, type=1 )
```

固有値問題が簡単に解ける！

#4

# 出力

```
print( "eigen values and vectors:" )
sys.stdout.write( "%5s" % ' ' )
for i in range( ndim ):
    sys.stdout.write( "%16d" % i )
print( "" )
sys.stdout.write( "%4s:" % 'E' )
for i in range( ndim ):
    sys.stdout.write( "%16.4e" % evals[ i ] )
print( "" )
for j in range( ndim ):
    sys.stdout.write( "%4d:" % j )
    for i in range( ndim ):
        sys.stdout.write( "%16.4e" % evecs[ i ][ j ] )
    print( "" )
```

1) <http://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eigh.html>



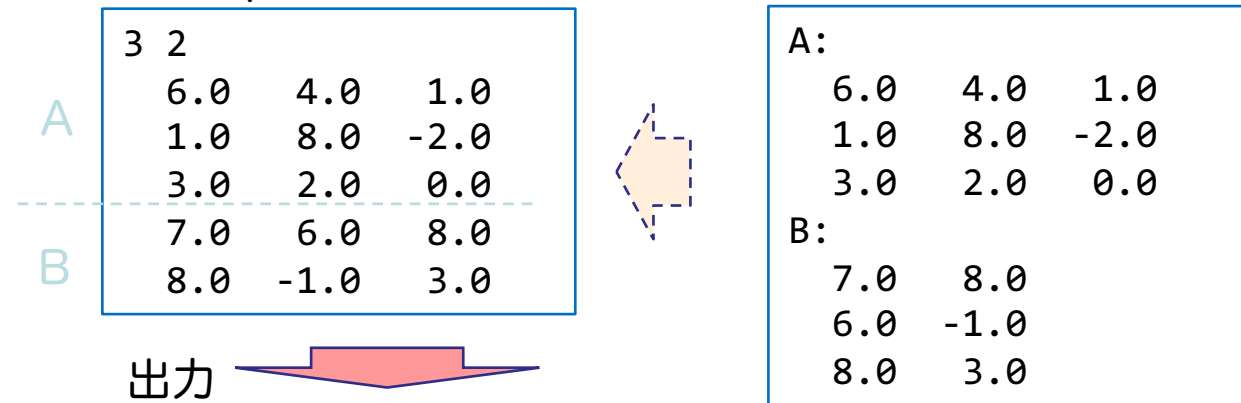
# 課題3.3.2: 線形計算 (連立方程式)

## 課題

以下の行列 A, B から, 連立方程式 ( $Ax = B$ ) を計算する.

入力 input\_lu

対象となる行列



出力

A:

6.0000	4.0000	1.0000
1.0000	8.0000	-2.0000
3.0000	2.0000	0.0000

B:

7.0000	8.0000
6.0000	-1.0000
8.0000	3.0000

ANS:

4.0000	0.8182
-2.0000	0.2727
-9.0000	2.0000

# 連立方程式解法プログラム (1/3)

全体プログラム構成 (LU 分解ライブラリを利用)

```
#1
# モジュール sys, numpy, scipy を読み込み
# モジュール linalg を scipy から更に読み込み

#2
# スカラー整数値 ndim, nsol の読み込み
# ndim * ndim 次元の NumPy 2次元配列 A と,
# nsol * ndim 次元の NumPy 2次元配列 B, X を生成
# 標準入力から入力データを読み込み, A, B に設定

#3
# SciPy のライブラリ linalg.lu_factor() を利用して LU 分解,
# 返回值として L, U, pivot を取得

#4
# nsol 回繰り返しループ
# ループボディにおいて, SciPy のライブラリ linalg.lu_solve() を利用

#5
# 出力
```

# 連立方程式解法プログラム (2/3)

プログラム：

NumPy 配列 A, B から LU 分解により連立方程式を解く

# 3

# SciPy のライブラリ linalg.lu\_factor() を利用して A を LU 分解,  
# 返回值として L, U を重ねた行列 LU, PIVOT を得る<sup>1)</sup>

```
LU, PIVOT = scipy.linalg.lu_factor( A )
```

# 4

# nsol 回繰り返しループ

# ループボディにおいて, SciPy のライブラリ linalg.lu\_solve() を利用し,  
# 連立方程式を解く<sup>2)</sup>

```
for isol in range( nsol ):
```

```
    X[ isol ] = scipy.linalg.lu_solve( (LU, PIVOT), B[ isol ] )
```

LU 分解が簡単に解ける！

連立方程式が簡単に解ける！

1) [http://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu\\_factor.html#scipy.linalg.lu\\_factor](http://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_factor.html#scipy.linalg.lu_factor)

2) [http://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu\\_solve.html#scipy.linalg.lu\\_solve](http://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_solve.html#scipy.linalg.lu_solve)

# 連立方程式解法プログラム (3/3)

## 全プログラム

```
import sys, numpy, scipy
from scipy import linalg

#####
strs = sys.stdin.readline()[::-1].split()
ndim = int( strs[ 0 ] )
nsol = int( strs[ 1 ] )
print( "ndim, nsol = %4d %4d" % ( ndim, nsol ) )

A = numpy.empty( [ ndim, ndim ], dtype=numpy.float64 )
B = numpy.empty( [ nsol, ndim ], dtype=numpy.float64 )
X = numpy.empty( [ nsol, ndim ], dtype=numpy.float64 )

for icol in range( ndim ):
    strs = sys.stdin.readline()[::-1].split()
    for irow in range( ndim ):
        A[ icol ][ irow ] = float( strs[ irow ] )

for icol in range( nsol ):
    strs = sys.stdin.readline()[::-1].split()
    for irow in range( ndim ):
        B[ icol ][ irow ] = float( strs[ irow ] )

#####
LU, PIVOT = scipy.linalg.lu_factor( A )
#print( LU )
for isol in range( nsol ):
    X[ isol ] = scipy.linalg.lu_solve( (LU, PIVOT), B[ isol ] )
```

#続き

```
print( "A:" )
for icol in range( ndim ):
    for irow in range( ndim ):
        sys.stdout.write( "%16.4f" % A[ icol ][ irow ] )
        sys.stdout.write( "¥n" )
print( "B:" )
for irow in range( ndim ):
    for isol in range( nsol ):
        sys.stdout.write( "%16.4f" % B[ isol ][ irow ] )
        sys.stdout.write( "¥n" )

print( "ANS:" )
for irow in range( ndim ):
    for isol in range( nsol ):
        sys.stdout.write( "%16.4f" % X[ isol ][ irow ] )
        sys.stdout.write( "¥n" )
```

# NumPy の使い方まとめ

- NumPy 配列の生成

- 空の NumPy 配列を生成<sup>1)</sup>

```
numpy_A = numpy.empty( [ 10, ndim ], dtype=numpy.float64 )
```

スカラー値の型<sup>2)</sup>

32 ビット整数: `numpy.int32`  
倍精度実数 : `numpy.float64`

- C 言語型データ並びでの **2 次元配列**
- n 要素の **1 次元配列**を指定する場合は, **[ n ]** と指定

- Python リストから NumPy 配列へ変換

```
numpy_A = numpy.array( [ A ], dtype=numpy.float64 )
```

- 多次元リストを渡すことも可能

```
numpy_A = numpy.array( [ 5 ], dtype=numpy.float64 )
```

スカラー値を渡すと、その値で初期化された 1 要素の 1 次元配列が生成

- NumPy 配列から Python リストへ変換

```
A = numpy_A.tolist()
```

1) <http://docs.scipy.org/doc/numpy/reference/routines.html>  
<http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html>

2) <http://docs.scipy.org/doc/numpy/reference/arrays.scalars.html#arrays-scalars-built-in>

他にも多くの関数

## 4. 並列プログラミング

1. MPI による出力プログラム
2. MPI による配列和プログラム

- saroma では一応実験可能ですが、ここはスキップ

# 課題4.1: MPI による Hello World 型プログラム

## 課題

MPI を用いた並列計算の 出力のみのプログラムを作成する.

1. MPI の ランク数取得関数, プロセス数取得関数, ノード名取得関数により各種情報を取得する.
2. 各ランク毎にランク番号とプロセス数, ノード名を出力する.

出力例 : 8 ノード 8 プロセス計算

```
nproc:      8 -> id:      0 : pcj0065
nproc:      8 -> id:      1 : pcj0066
nproc:      8 -> id:      4 : pcj0069
nproc:      8 -> id:      5 : pcj0070
nproc:      8 -> id:      3 : pcj0068
nproc:      8 -> id:      2 : pcj0067
nproc:      8 -> id:      6 : pcj0071
nproc:      8 -> id:      7 : pcj0072
```

8 行

# MPI による Hello World 型プログラム

## プログラム内容

```
# 1
# モジュール mpi4py から MPI モジュールを読み込み
from mpi4py import MPI

# 2
# COMM_WORLD のコミュニケータを取得
# そのコミュニケータでのランク番号とランク数を取得
# 現在のランクが動いているプロセッサのホスト名前を取得
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
name = MPI.Get_processor_name()

# 3
# 出力
print( "nproc: %4d -> id: %4d : %s" % ( size, rank, name ) )
```

MPI\_Init() と MPI\_Finalize() の処理は不要！



# 課題4.2: MPI による配列和の計算

## 課題

1. 全てのプロセスにおいて、10000 要素を持つ倍精度実数の NumPy の 1次元配列  $x$  を生成し、要素番号と各ランク番号に依存した初期値を設定する.
2. 次に、Root ランク(Root = 0) の  $x$  に、全てのプロセスからの  $x$  を加算する.
3. Root ランクの  $x[0]$ ,  $x[9999]$  を出力する.

出力例：8 ノード 8 プロセス計算

```
Nproc =          8
Before reduce:
  4.00039380e+00,   4.00000004e+04
After  reduce:
  1.44003150e+02,   1.44000000e+06
```

# MPI による配列和プログラム (1/2)

## プログラム前半

```
# 1
# モジュール numpy を読み込み
# モジュール mpi4py から MPI モジュールを読み込み
import numpy
from mpi4py import MPI

# 2
# COMM_WORLD のコミュニケータを取得
# COMM_WORLD 中のランク番号を取得
# root として 0 を設定
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
root = 0

# 3
# 配列要素数を 10000 とする
# 配列要素数サイズのNumPy 1次元配列を生成
# 配列の全要素に値を設定 (任意の値)
nsize = 10000
array = numpy.empty( [ nsize ], dtype=numpy.float64 )
for i in range( nsize ):
    array[ i ] = 3.938e-04 + ( i + 1 ) * 4 * ( rank + 1 )
```

# MPI による配列和プログラム (2/2)

## プログラム後半

```
# 4
# ランク番号が root の場合に配列を出力
if ( myrank == root ):
    print( "Nproc = %10d" % nprocs )
    print( "Before reduce:" )
    print( "%16.8e, %16.8e" % ( array[ 0 ], array[ nsize - 1 ] ) )

# 5
# コミュニケータ comm に対して, reduce を計算
# 返り値として reduce 後の配列を取得
garray = comm.reduce( array, MPI.SUM, root )

# 6
# ランク番号が root の場合に reduce で取得した配列を出力
if ( myrank == root ):
    print( "After reduce:" )
    print( "%16.8f, %16.8f" % ( garray[ 0 ], garray[ nsize - 1 ] ) )
```

## 5. Fortran や C 言語による拡張

1. 自作 sort プログラムの利用
2. LAPACK ライブラリの利用

# 課題5.1: 自作 sort プログラムの利用

## 課題

コンパイル型言語 (Fortran77, Fortran95, C, C++) により記述された自作のバブルソート関数を Python から利用する.  
この際に, 以下の A, B の2種類の関数を対象とする.

- A. 配列型 (Fortran, Fortran95, C, C++) :  
Keys と Values の2つの配列について, Keys をキーとして  
Keys と Values の2つの配列を昇順に並び替える関数
- B. 構造体型 (Fortran95, C, C++) :  
key と value のメンバーを持つ構造体配列について,  
key をキーとして昇順に並び替える関数

# Fortran77 言語の対象関数

sort\_f.f プログラム

整数や実数の配列のみ

```
subroutine bsortf( n, keys, values )  
implicit none  
integer n  
integer keys( n )  
double precision values( n )  
...  
end
```

# Fortran95 言語の対象関数

## sort\_f95.f95 プログラム

```
module mysort
  type keyvalue
    integer :: key
    double precision :: value
  end type keyvalue

contains
  subroutine bsortf95_array( n, keys, array )
    implicit none
    integer n
    integer keys( n )
    double precision array( n )
  ...
end subroutine bsortf95
  subroutine bsortf90_struct( n, array )
    implicit none
    integer n
    type( keyvalue ), dimension( n ) :: array
  ...
end subroutine bsortf90_struct
end module mysort
```

整数や実数の配列のみの引数

構造体配列の引数

# C 言語の対象関数

## sort\_c.h プログラム

```
typedef struct {  
    int    key ;  
    double value ;  
} key_value_t ;  
  
Extern int bsortc_array( int n, int *keys, doule *values ) ;  
extern int bsortc_struct( int n, key_value_t *array ) ;
```

## sort\_c.c プログラム

```
#include "sort_c.h"
```

```
int bsortc_array( int n, int *keys, doule *values )  
{  
    ...  
}
```

整数や実数の配列のみの引数

```
int bsortc_struct( int n, key_value_t *array )  
{  
    ...  
}
```

構造体配列の引数



# C++ 言語の対象関数

sort\_cpp.cpp プログラム

```
#include "sort_c.h"
```

```
int bsortcpp_array( int n, int *keys, double *values )  
{  
  ...  
}
```

整数や実数の配列のみの引数

```
int bsortcpp_struct( int n, key_value_t *array )  
{  
  ...  
}
```

構造体配列の引数


# コンパイルとライブラリ作成, 関数名調査

各プログラムをコンパイル後, 共有ライブラリを作成する

```
$ make
gcc -Wall -fPIC -g -I. -c sort_c.c
gfortran -Wall -fPIC -g -c sort_f.f
gfortran -Wall -fPIC -g -c sort_f95.f95
g++ -Wall -fPIC -g -I. -c sort_cpp.cpp
gcc -Wall -fPIC -g -shared -Wl,-soname,libsort.so -lgfortran -lstdc++ sort_c.o
sort_f.o sort_f95.o sort_cpp.o -o libsort.so (mac なら、*.dylib)
```

共有ライブラリ中の各関数名を取得する

```
$ nm libsort.so | grep sort
00000000000001954 T _Z14bsortcpp_arrayiPiPd
00000000000001b52 T _Z15bsortcpp_structiP11key_value_t
00000000000001523 T __mysort_MOD_bsortf95_array
00000000000001138 T __mysort_MOD_bsortf95_struct
000000000000009cc T bsortc_array
00000000000000bc5 T bsortc_struct
00000000000000d88 T bsortf_
```



C++	array	関数
C++	struct	関数
F95	array	関数
F95	struct	関数
C	array	関数
C	struct	関数
F	array	関数

一般に, C 言語以外の関数は各コンパイラにより決められた規則により名前が変更されライブラリに保存される. Python から利用する場合には, ライブラリ中の名前が必要となる

# 各言語のプログラムのテスト

各プログラムをコンパイルし、動作を確認する

```
$ make exe
gfortran -Wall -fPIC -g -o test_f test_f.f -L. -lsort
gcc -Wall -fPIC -g -I. -o test_c test_c.c -L. -lsort
gfortran -Wall -fPIC -g -o test_f95 test_f95.f95 -L. -lsort
g++ -Wall -fPIC -g -o test_cpp test_cpp.cpp -L. -lsort

$ ./test_f
$ ...
```

# Fortran 副プログラムの C からの利用

- Python から他のコンパイル型言語のライブラリを利用する際には、それぞれの関数について、C 言語からのインターフェースを基準にしている
- Fortran のインターフェースはそのままでは利用出来ない

```
subroutine bsortf( n, keys, values )  
integer n  
integer keys(n)  
double precision values(n)
```



C言語から呼び出す場合の関数インターフェース

```
void bsortf_( int *n, int *keys, double *values ) ;
```


## 注意

- Fortran の 関数の引数はいつでも参照型と考える
- subroutine は C からは void とみなす
- Fortran の 1, 2 次元配列はいつでも C からは 1 次元配列として利用する (と良い)
- 関数名も変化するため, nm コマンド等により調べる (と納得できる)

# ctypes による外部プログラムの利用

ctypes は、C と互換性のあるデータ型を提供し  
共有ライブラリ等の内部に定義された関数の呼び出しを可能とする方法  
(モジュール) の 1 つ

libaaa.so (.dylib (mac)) 内の, "戻り値 func( 引数1, 引数2, ... )" なる  
関数を利用する場合

 基本はこれだけ

ctypes による func 呼び出し記述ファイル `wrap_sort.py`

```
import ctypes
module_aaa = ctypes.CDLL( "path_to_library/libaaa.so" )
module_aaa.func.restype = 戻り値の型
module_aaa.func.argtypes = [ 引数1の型, 引数2の型, ... ]
```

func 関数を利用するプログラム

```
import ctypes, wrap_sort
...
arg1 = ...
arg2 = ...
retval = wrap_sort.module_aaa.func( arg1, arg2, ... )
```

- `wrap_sort.py`, `module_aaa` の  
名称は自由に定める事が可能
- 戻り値型や引数の型は ctypes により  
定められた記述方法をとる

# wrap\_sort.py の詳細 (1/3)

wrap\_sort.py の全体

```
#1
# ctypes をインポート

#2
# libsort.so をロードし, モジュール名を設定

#3
# ctypes のエイリアス型名と構造体型を作成する
# (エイリアス形名の作成は必須ではなく, コードを見やすくするため)

#4
# それぞれの利用する関数に対し, 関数の型と引数を設定する
```

# wrap\_sort.py の詳細 (2/3)

wrap\_sort.py の前半

```
#1
# ctypes をインポート
import ctypes

#2
# libsort.dylib (.so) をロードし、モジュール名を設定
libsort = ctypes.CDLL( "../libsort.dylib" )
```

```
#3
# ctypes のエイリアス型名と構造体型を作成する
```

```
T_PTR_VOID    = ctypes.c_void_p
T_INT          = ctypes.c_int32
T_PTR_INT      = ctypes.POINTER( ctypes.c_int32 )
T_PTR_DOUBLE   = ctypes.POINTER( ctypes.c_double )
```

```
class STRUCT_ELEM( ctypes.Structure ):
    _fields_ = [ ( "key", ctypes.c_int ), ( "value", ctypes.c_double ) ]
    def __init__( self, key, value ):
        self.key = key
        self.value = value
```

```
T_PTR_STRUCT = ctypes.POINTER( STRUCT_ELEM )
```

- ctypes.c\_... の様な記述方法で型指定が可能
- POINTER 関数で ctype の型のポインタ指定する
- 構造体型は下の様に記述

```
typedef struct {
    int    key ;
    double value ;
} key_value_t ;
```

構造体型作成

構造体へのポインタ型のエイリアス形名作成

# wrap\_sort.py の詳細 (3/3)

wrap\_sort.py の後半

#4

# それぞれの利用する関数に対し、関数の型と引数を設定

```
void bsortf_( int *n, int *keys, double *values ) ;
```

元の関数定義 (C言語からリンク可能な形)

# Fortran77 bsortf subroutine

```
libsort.bsortf_.restype = T_PTR_VOID
```

```
libsort.bsortf_.argtypes = [ T_PTR_INT, T_PTR_INT, T_PTR_DOUBLE ]
```

戻り値が void の場合は  
void ポインタを設定

```
int bsortc_struct( int n, keyvalue_t *array ) ;
```

# C bsortc\_struct function

```
libsort.bsortc_struct.restype = T_INT
```

```
libsort.bsortc_struct.argtypes = [ T_INT, T_PTR_STRUCT ]
```

```
subroutine bfortf95_struct( n, array )
```

```
void .__mysort_MOD_bsortf95_struct( int *n, keyvalue_t *array ) ;
```

# Fortran95 bfortf95\_struct subroutine

```
libsort.__mysort_MOD_bsortf95_struct.restype = T_PTR_VOID
```

```
libsort.__mysort_MOD_bsortf95_struct.argtypes = [ T_PTR_INT, T_PTR_STRUCT ]
```

...



# bssort 関数の呼び出し (1/3)

構造体を利用しない場合の sort 関数呼び出しの全体 (main\_sort\_array.py)

#インポート (ctypes, wrap\_sort)

# 各種データの初期化( 整数 nelem, 整数リスト keys\_data, 実数リスト values\_data)

# nelem 変数を ctypes での int32 と, int32 の参照(ポインタ) に変換

nelem\_c = ctypes.c\_int32( nelem )

**ptr\_nelem** = ctypes.byref( nelem\_c )

# Python のリストデータを ctypes の配列データへ変換

**keys\_ctypesarray** = (ctypes.c\_int32 \* nelem)( \*keys\_data )

values\_ctypesarray = (ctypes.c\_double \* nelem)( \*values\_data )

型

要素数

リストデータの前に \* を付ける

# ソート前出力

...

# 関数呼び出し

ctypes 型に変換された**変数の参照**を渡す

wrap\_sort.libsort.bsorthf\_( **ptr\_nelem**, **keys\_ctypesarray**, **values\_ctypesarray** )

#wrap\_sort.libsort.bsorthc\_array( nelem\_c, keys\_ctypesarray, values\_ctypesarray )

#wrap\_sort.libsort.\_\_mysort\_MOD\_bsorthf95\_array( ptr\_nelem, keys\_ctypesarray,  
# values\_ctypesarray )

#wrap\_sort.libsort.\_Z14bsorthcpp\_arrayiPiPd( nelem\_c, keys\_ctypesarray,  
# values\_ctypesarray )

# ソート後出力

ctypes 型に変換された変数を渡す

...

# bssort 関数の呼び出し (2/3)

構造体を利用する場合の sort 関数呼び出しの全体 (main\_sort\_struct.py)

```
#インポート (ctypes, wrap_sort)
```

```
...
```

```
# 各種データの初期化( 整数 nelem, タプルのリスト array_data)
```

```
nelem = 10
```

```
array_data = [ ( 4, -3.38e+00 ), ( 7, -9.29e+00 ), ( 20, 1.38e+00 ), ( -9, 8.02e+00 ),  
               ( 77, 1.33e+00 ), ( -3, 3.30e+00 ), ( 13, -4.30e+00 ), ( -5, 5.00e+00 ),  
               ( 10, 7.90e+00 ), ( 1, 3.30e+00 ) ]
```

```
# nelem 変数を ctypes での int32 と, int32 の参照(ポインタ) に変換
```

```
...
```

```
# Python のタプルのリストデータを ctypes の構造体配列データへ変換
```

```
structarray = (wrap_sort.STRUCT_ELEM * nelem)( *array_data )
```

型

要素数

リストデータの前に \* を付ける

```
# ソート前出力
```

```
...
```

```
# 関数呼び出し
```

```
wrap_sort.libsort.bsorc_struct( nelem_c, structarray )
```

```
#wrap_sort.libsort.__mysort_MOD_bsorc_f95_struct( ptr_nelem, structarray )
```

```
#wrap_sort.libsort._Z15bsorc_cpp_structiP11key_value_t( nelem_c, structarray )
```

ctypes 型に変換された  
変数の参照を渡す

```
# ソート後出力
```

```
...
```

ctypes 型に変換された変数を渡す

# bssort 関数の呼び出し (3/3)

**NumPy を利用する場合**の sort 関数呼び出しの全体 (main\_sort\_numpy\_array.py)

```
#インポート (ctypes, wrap_sort, numpy)
```

```
# 各種データの初期化( 整数 nelem, 整数リスト keys_data, 実数リスト values_data)
```

```
# nelem 変数を ctypes での int32 に変換 ( nelem -> nelem_c = ctypes.int32( nelem ) )
```

```
# Python の初期化リストデータを NumPy の配列データへ変換
```

```
n      = numpy.array( [ nelem ],      dtype = numpy.int32 )
```

```
keys    = numpy.array( keys_data,     dtype = numpy.int32 )
```

```
values  = numpy.array( values_data,    dtype = numpy.float64 )
```

```
# NumPy の配列データの内容データの参照を取得
```

```
ptr_nelem = n.ctypes.data_as( wrap_sort.T_PTR_INT )
```

```
ptr_keys  = keys.ctypes.data_as( wrap_sort.T_PTR_INT )
```

```
ptr_values = values.ctypes.data_as( wrap_sort.T_PTR_DOUBLE )
```

```
# ソート前出力
```

numpy.ctypes.data\_as( ctypes のポインタ型 )  
により, NumPy 配列の参照を取得

```
# 関数呼び出し
```

```
wrap_sort.libsort.bsorthf_( ptr_nelem, ptr_keys, ptr_values )
```

```
#wrap_sort.libsort.bsorthc_array( nelemc, ptr_keys, ptr_values )
```

```
#wrap_sort.libsort.__mysort_MOD_bsorthf95_array( ptr_nelem, ptr_keys, ptr_values )
```

```
#wrap_sort.libsort._Z14bsorthcpp_arrayiPiPd( nelemc, ptr_keys, ptr_values )
```

```
# ソート後出力
```

# 課題5.2: LAPACK ライブラリの直接利用

## 課題

LAPACK ライブラリに有る実対称行列一般固有値問題 (dspgv 関数) を利用し、以下の行列 A, B による固有値問題 ( $Ax = \lambda Bx$ ) を計算する。

入力 input

```
4
0 0  0.24  4.16
0 1  0.39 -3.12
1 1 -0.11  5.03
0 2  0.42  0.56
1 2  0.79 -0.83
2 2 -0.25  0.76
0 3 -0.16 -0.10
1 3  0.63  1.09
2 3  0.48  0.34
3 3 -0.03  1.18
```

対象となる行列

```
A:
0.24  0.39  0.42 -0.16
      -0.11  0.79  0.63
              -0.25  0.48
                  -0.03

B:
4.16 -3.12  0.56 -0.10
      5.03 -0.83  1.09
              0.76  0.34
                  1.18
```

対称行列を1次元で記述

出力

	0	1	2	3
E	-2.2254	-0.4548	0.1001	1.1270
0	-0.0690	-0.5740	-1.5428	1.4004
1	0.3080	0.5329	-0.3496	-0.6211
2	-0.4469	-0.0371	0.0505	0.4743
3	-0.5528	-0.6766	-0.9276	0.2510

# dspgv 関数のインターフェース調査

LAPACK ライブラリ liblapack.so (liblapack.dylib (mac)) の場所を調べておく

本講習会では、以下のファイルとして用意  
/usr/lib64/liblapack.so

```
$ find /usr/lib64 -name '*liblapack*' -print    #で調べて下さい !!!  
        (ひょっとすると、/usr/lib, /usr/local,, or /opt/local の下かもしれません)
```

Fortran による dspgv 関数

```
SUBROUTINE dspgv(ITYPE, JOBZ, UPLO, N, AP, BP, W, Z, LDZ, WORK, INFO)  
  INTEGER          ITYPE, N, LDZ, INFO  
  double precision AP(LDA,*), BP(LDB,*), W(*), WORK(*)  
  CHARACTER*1      JOBZ, UPLO
```



C から利用する際の dspgv 関数

```
void dspgv_( int *itype, char *jobz, char *uplo, int *n, double *A, double *B,  
            double *w, int *ldz, double *work, int *info) ;
```

C から利用する際の dspgv 関数の名前は、nm コマンドにより確認しておく。  
LAPACK ライブラリの位置はシステムにより異なるので注意。

```
$ nm /home/lib/liblapack.dylib | grep dspgv  
00000000001e30d0 T dspgv_          (アンダースコアが有ることに注意)
```

# wrap\_dspgv.py の詳細

wrap\_dspgv.py の全体

# ctypes モジュールのインポート

```
import ctypes
```

# 共有ライブラリをロードして、モジュール名 liblapack として扱えるようにする

```
liblapack = ctypes.CDLL( "/home/lib/liblapack.dylib" )
```

# 型宣言のエイリアスの作成

```
T_PTR_VOID = ctypes.c_void_p
```

```
T_PTR_CHAR = ctypes.POINTER( ctypes.c_char )
```

```
T_PTR_INT = ctypes.POINTER( ctypes.c_int32 )
```

```
T_PTR_DOUBLE = ctypes.POINTER( ctypes.c_double )
```

前ページ find で調べた  
ファイルパスを指定して下さい

C 言語からの dspgv\_ 関数のインターフェース

```
void dspgv_( int *itype, char *jobz, char *uplo, int *n, double *AP, double *BP,  
             double *w, int *ldz, double *work, int *info) ;
```

# 関数 dspgv\_ の型を liblapack の中の dspgv\_ モジュールに伝える

# 返り値設定

```
liblapack.dspgv_.restype = T_PTR_VOID
```

返り値の型 void  
=> void のポインタ型

# 引数値設定

```
liblapack.dspgv_.argtypes = [  
    T_PTR_INT, T_PTR_CHAR, T_PTR_CHAR, T_PTR_INT, T_PTR_DOUBLE, T_PTR_DOUBLE,  
    T_PTR_DOUBLE, T_PTR_DOUBLE, T_PTR_INT, T_PTR_DOUBLE, T_PTR_INT ]
```

# dspgv\_ 関数の呼び出し (1/3)

全体プログラム (NumPy 利用版)

```
# sys, numpy, ctypes, wrap_dspgv モジュールのインポート  
  
# ndim を読み込む  
  
# dspgv_ に渡す NumPy 配列を生成  
  
# 上記で生成したそれぞれの配列へのポインタを取得する  
  
# A, B の 1 次元配列に入力を読み込む  
  
# wrap_dspgv.liblapack.dspgv_() へ、上記のポインタを渡して計算させる  
  
# 計算結果の固有ベクトルと固有値を配列から読み込み、出力
```

# dspgv\_ 関数の呼び出し (2/3)

全体プログラムの前半 (NumPy 利用版)

```
# sys, numpy, ctypes, wrap_dspgv モジュールのインポート
import sys, numpy
import wrap_dspgv
from ctypes import *

# ndim を読み込む
...

# dspgv_ に渡す NumPy 配列を生成
itype = numpy.array( [ 1 ], dtype = numpy.int32 )
n      = numpy.array( [ ndim ], dtype = numpy.int32 )
ap     = numpy.empty( [ nsize ], dtype = numpy.float64 )
bp     = numpy.empty( [ nsize ], dtype = numpy.float64 )
...

# 上記で生成したそれぞれの配列へのポインタを取得する
ptr_jobz = c_char_p( b'V' )
Ptr_itype = itype.ctypes.data_as( wrap_dspgv.T_PTR_INT )
ptr_n     = n.ctypes.data_as( wrap_dspgv.T_PTR_INT )
ptr_ap    = ap.ctypes.data_as( wrap_dspgv.T_PTR_DOUBLE )
ptr_bp    = bp.ctypes.data_as( wrap_dspgv.T_PTR_DOUBLE )
...

# A, B の 1 次元配列に入力を読み込む
...
```



# dspgv\_ 関数の呼び出し (3/3)

全体プログラムの後半

# wrap\_dspgv.liblapack.dspgv\_() へ、上記のポインタを渡して計算させる

```
wrap_dspgv.liblapack.dspgv_(  
    ptr_itype, ptr_jobz, ptr_uplo, ptr_n, ptr_ap, ptr_bp,  
    ptr_w, ptr_z, ptr_ldz, ptr_wk, ptr_info )
```

- この計算により,  
ptr\_w, ptr\_z のポインタが指す領域, w, z 配列に,  
固有値と固有ベクトルが保存される
- 計算の返り値は ptr\_info が指す領域 info の info[ 0 ] に保存されている

# 計算結果の固有ベクトルと固有値を利用する (出力する)

```
...  
sys.stdout.write( "%4s:" % 'E' )  
for i in range( ndim ):  
    sys.stdout.write( "%16.4e" % w[ i ] )  
sys.stdout.write( "¥n" )  
for j in range( ndim ):  
    sys.stdout.write( "%4d:" % j )  
    for i in range( ndim ):  
        sys.stdout.write( "%16.4e" % z[ j * ndim + i ] )  
sys.stdout.write( "¥n" )
```

# 付録

## 1. 配布ソフト

## 2. 各ソフトウェアのインストール方法

- Debian8 にて確認（センターの計算機でも可能なはず）
- 自身でソフトウェアをソースから \$HOME へインストール場合
  - \$HOME/Python
  - \$HOME/local/openmpi-1.10.3
  - \$HOME/local/lib/libblas.so, liblapack.so

root 権限を持つ場合には、  
ソフトウェアを yum や apt-get によりインストールすることができます。  
各自で調べてみてください。

# A1.1: 配布ソフト

## セットアップ

```
$ cd ./tutorial_riit  
$ . .profile  
$ ls  
doc ffi hello mpi seq
```

**PATH等の設定。  
Python, MPI, LAPACK の  
ライブラリ位置を設定してい  
ます。ファイル内を確認しま  
しょう**

## 資料との対応

hello	基本2.1, 2.2, 2.3, 2.4
seq/awk	課題3.1
seq/sbst	課題3.2
seq/mat	課題3.3.1, 3.3.2
mpi/python	課題4.1, 4.2
ffi/sort	課題5.1
ffi/dspgv	課題5.2

## 本講習内容の確認：

- 2種類の Linux 環境
  - 基盤センター CX400 (Redhat6.1)
    - Python2.7, Python3.5
    - OpenMPI-1.10.3
  - 通常の Linux 環境 (Debian8)
    - Python2.7, Python3.6
    - OpenMPI-1.10.3

**同一ソースで、両 python  
バージョンでの動作を確認。  
(CX400 は九大基盤セン  
ターの旧スパコン)**

# A1.2: .profile の設定

## .profile の設定説明

### #Root dirs

```
DIR_PYTHON=$HOME/Python
DIR_MPI=$HOME/local/openmpi-1.10.3
DIR_LIBLAPACK=$HOME/local/lib
```

それぞれのソフトウェアの  
インストール先に従い、変更

### #For python, python3, mpicc, and mpirun command

```
PATH=${DIR_PYTHON}/bin:${DIR_MPI}/bin:$PATH
```

### #For libblas.so liblapack.so, user-defined and mpi libraries

```
LD_LIBRARY_PATH=.:${DIR_LIBLAPACK}:${DIR_MPI}/lib:${LD_LIBRARY_PATH}
```

#I don't know whether these are necessary.

```
BLAS=${DIR_LIBLAPACK}/libblas.so
LAPACK=${DIR_LIBLAPACK}/liblapack.so
```

PYTHONPATH に  
自身の作成した \*.py を置いておくと、  
import が可能になる

### #For python user-defined modules

```
PYTHONPATH=.:${PYTHONPATH}
```

### #For python numpy. Configurations of threads

```
OPENBLAS_NUM_THREADS=1
```

```
OPENBLAS_USE_THREADS=0
```

- センターコンピュータでの実行の際に、numpy を import すると python から segmentation fault が発生する場合に有効.
- 現状で、Python から Fortran を利用する場合に、シリアル実行をする。特に、過去の CX400 上での実行では、Segmentation fault が発生した。

### #Don't forget to export.

```
export PATH LD_LIBRARY_PATH BLAS LAPACK PYTHONPATH OPENBLAS_NUM_THREADS
OPENBLAS_USE_THREADS
```

# A1.3: 配布ソフトの実行方法

## 課題 2

```
$ cd ./hello
$ python hello.0.py
$ python hello.1.py
$ python hello.2.py
$ python hello.3.py
```

## 課題 3. 1

```
$ cd ./seq/sbst
$ ./gen_xyz.sh
$ python sbst.py __ZZ__ 8.8 in.template
$ python gen_xyz.py
```

## 課題 3. 2

```
$ cd ./seq/awk
$ python awk.py < input
```

## 課題 3. 3

```
$ cd ./seq/mat
$ python eig.py < input_eig
$ python lu.py < input_lu
```

## 課題 4

```
$ cd ./mpi/python
$ ./run-local.sh
$ pjsub run-tatara.sh
----
run-local.sh と run-tatara.sh 内の
EXE=${DIR}/test_hello.py を
#EXE=${DIR}/test_hello.py にコメントアウト
----
$ ./run-local.sh
$ pjsub run-tatara.sh
```

### 課題 4. 1

CX400  
バックエンド実行

### 課題 4. 2

## 課題 5. 1

```
$ cd ./ffi/sort
$ make ; make exe
$ python main_sort_array.py
$ python main_sort_struct.py
$ python main_sort_numpy_array.py
```

native 版  
プログラムの作成  
別の実行を確認

## 課題 5. 2

```
$ cd ./ffi/dspgv
$ make
$ python main_dspgv.py < input
```

付属の native 版  
プログラムの作成  
別の実行を確認

以下、ソースから  
全てインストールする場合

# A2: Python のインストール

## Python 2 のインストール

```
$ wget https://www.python.org/ftp/python/2.7.12/Python-2.7.12.tgz
$ tar -xzf Python-2.7.12.tgz
$ cd Python-2.7.12
$ ./configure --prefix=$HOME/python
$ make ; make install
$ export PATH=$HOME/python/bin:$PATH
```

現在は 2.7.14 です

## Python 3 のインストール

```
$ wget https://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz
$ tar -xzf Python-3.5.2.tgz
$ cd Python-3.5.2
$ ./configure --prefix=$HOME/python
$ make ; make install
$ export PATH=$HOME/python/bin:$PATH
```

現在は 3.6.5 です

シェルのスタートアップファイルに記入する

Python2 と Python3 は同一のディレクトリにインストール可能 !!

# A3: MPI のインストール

## OpenMPI のインストール

```
$ wget https://www.open-mpi.org/software/ompi/v1.10/downloads/openmpi-1.10.3.tar.gz
$ tar -xzf openmpi-1.10.3.tar.gz
$ cd openmpi-1.10.3
$ ./configure --prefix=$HOME/local/openmpi-1.10.3 --enable-shared
$ make ; make install
$ export PATH=$HOME/local/openmpi-1.10.3/bin:$PATH
$ export LD_LIBRARY_PATH=$HOME/local/openmpi-1.10.3/lib:$LD_LIBRARY_PATH
```

シェルのスタートアップファイルに記入する

MPICH2 ソフト (<https://www.mpich.org>) でも良い



# A3: BLAS のインストール

## BLAS のインストール

```
$ wget http://www.netlib.org/blas/blas.tgz
$ tar xzf blas.tgz
$ cd BLAS-3.5.0
$ gfortran -O3 -m64 -fPIC -c *.f
$ ar r libblas.a *.o
$ ranlib libblas.a
$ mkdir -p $HOME/local/lib
$ cp libblas.a $HOME/local/lib
$ gfortran -O3 -m64 -fPIC -shared -o libblas.so *.o
$ cp libblas.a libblas.so $HOME/local/lib
$ export BLAS=$HOME/local/lib/libblas.so
```

シェルのスタートアップファイルに記入する

# A4: Lapack のインストール

## Lapack のインストール

```
$ wget http://www.netlib.org/lapack/lapack.tgz
$ tar xzf lapack.tgz
$ cd lapack-3.6.1
$ cp INSTALL/make.inc.gfortran make.inc
$ vi make.inc
----
OPTS = -O2 -m64 -fPIC
NOOPT = -O0 -m64 -fPIC
----
$ make lapacklib
$ cp liblapack.a $HOME/local/lib
$ cd SRC
$ vi Makefile
----
liblapack.so: $(ALLOBJ)
    gfortran -shared -Wl,-soname,$@ -o $@ $(ALLOBJ) -L$(HOME)/local/lib -lblas
    cp liblapack.so ../
----
$ make liblapack.so ; cd ..
$ cp liblapack.so $HOME/local/lib
$ export LAPACK=$HOME/local/lib/liblapack.so
$ export LD_LIBRARY_PATH=$HOME/local/lib:$LD_LIBRARY_PATH
```

make.inc を  
2 行変更

Makefile の  
一番最後に追加。  
先頭の空白は TAB 文字

シェルのスタートアップファイルに記入する

Lapack のインストールは BLAS インストール後に

# A5: NumPy, mpi4py, SciPy のインストール

Python パッケージソフト管理ツール (pip) のインストール

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ python2.7 get-pip.py
$ python3.6 get-pip.py
```

pip を利用した各パッケージのインストール

```
$ pip2.7 install numpy
$ pip3.6 install numpy
$ pip2.7 install nose
$ pip3.6 install nose
$ pip2.7 install mpi4py
$ pip3.6 install mpi4py
$ pip2.7 install scipy
$ pip3.6 install scipy
$ pip2.7 install setuptools
$ pip3.6 install setuptools
```

} NumPy, SciPy のテストを実施する際には必要

mpi4py のインストールはMPI インストール後に

} pip 以外のパッケージソフト管理ツール easy\_install のインストール (おまけ)

以下のエラーメッセージが出力された場合には、すでにそのパッケージがインストール済み。  
(ひょっとして、バージョンが古い等の問題もある)

```
Requirement already satisfied (use --upgrade to upgrade): setuptools in ...
```

python プログラムにて numpy をインポートするとそれだけで segmentation fault が発生する際には以下を試みる

```
export OPENBLAS_NUM_THREADS=1
export OPENBLAS_USE_THREADS=0
```