Introduction to High Performance Scientific Computing

Autumn, 2016

Lecture 9

Imperial College
London

Prasun Ray
7 November 2016

Avoiding Fortran

- Libraries
- Introduction to F2Py

Imperial College
London

Using libraries with Fortran

- Avoid writing own code whenever possible!
- Many well-established libraries for scientific computing are freely available.
  - Save time: don't have to write/test own code
  - Standard libraries have been extensively tested
  - Libraries often optimized to run fast, difficult to do better

Imperial College
London

Using libraries with Fortran

- Avoid writing own code whenever possible!
- Many well-established libraries for scientific computing are freely available.
    - Save time: don't have to write/test own code
    - Standard libraries have been extensively tested
    - Libraries often optimized to run fast, difficult to do better

Examples:

- Netlib: minpack, odepack, blas, …
- FFTW – fastest fourier transform in west
- Lapack (we will focus on this)

Usually possible to call libraries written in *c* from *fortran* and vice versa

Imperial College
London

Examples

minpack: ***…****software for solving nonlinear equations and nonlinear least squares problems, netlib.org/minpack (fortran 77)*

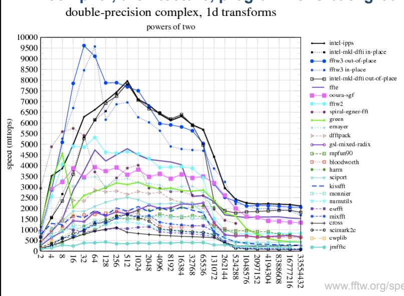
blas: *The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. netlib.org/blas (fortran 77, see also atlas, http://math-atlas.sourceforge.net/ for optimized blas)*

fftw: fastest fourier transform in the west, *FFTW is a free collection of fast C routines for computing the Discrete Fourier Transform in one or more dimensions.* fftw.org

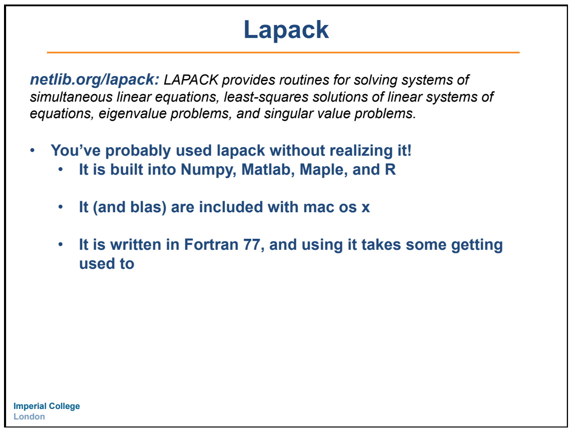
Imperial College
London

Choosing the right library

- Sometimes there are many libraries for the same task
- Example: "Right" fast fourier transform package depends on: compiler, architecture, programmer's background



www.fftw.org/speed/CoreDuo-3.0GHz-icc64/

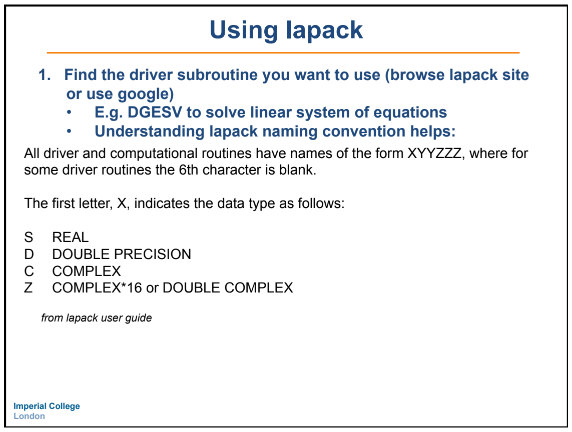


Lapack

netlib.org/lapack: *LAPACK provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems.*

- **You've probably used lapack without realizing it!**
  - **It is built into Numpy, Matlab, Maple, and R**
  - **It (and blas) are included with mac os x**
  - **It is written in Fortran 77, and using it takes some getting used to**

Imperial College
London



Using lapack

1. **Find the driver subroutine you want to use (browse lapack site or use google)**
   - **E.g. DGESV to solve linear system of equations**
   - **Understanding lapack naming convention helps:**

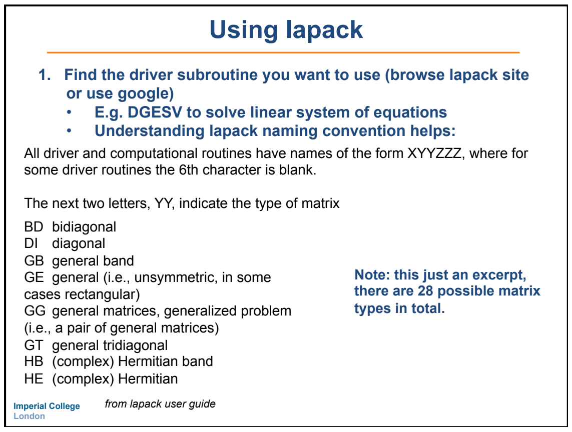
All driver and computational routines have names of the form XYYZZZ, where for some driver routines the 6th character is blank.

The first letter, X, indicates the data type as follows:

S	REAL
D	DOUBLE PRECISION
C	COMPLEX
Z	COMPLEX*16 or DOUBLE COMPLEX

from lapack user guide

Imperial College
London



Using lapack

1. **Find the driver subroutine you want to use (browse lapack site or use google)**
   - **E.g. DGESV to solve linear system of equations**
   - **Understanding lapack naming convention helps:**

All driver and computational routines have names of the form XYYZZZ, where for some driver routines the 6th character is blank.

The next two letters, YY, indicate the type of matrix

BD	bidiagonal
DI	diagonal
GB	general band
GE	general (i.e., unsymmetric, in some cases rectangular)
GG	general matrices, generalized problem (i.e., a pair of general matrices)
GT	general tridiagonal
HB	(complex) Hermitian band
HE	(complex) Hermitian

Note: this just an excerpt, there are 28 possible matrix types in total.

Imperial College
London

from lapack user guide

Using lapack

1. **Find the driver subroutine you want to use (browse lapack site or use google)**
   - **E.g. DGESV to solve linear system of equations**
   - **Understanding lapack naming convention helps:**

All driver and computational routines have names of the form XYYZZZ, where for some driver routines the 6th character is blank.

The next two letters, YY, indicate the type of matrix

BD	bidiagonal
DI	diagonal
GB	general band
GE	general (i.e., unsymmetric, in some cases rectangular)
GG	general matrices, generalized problem (i.e., a pair of general matrices)
GT	general tridiagonal
HB	(complex) Hermitian band
HE	(complex) Hermitian

from lapack user guide

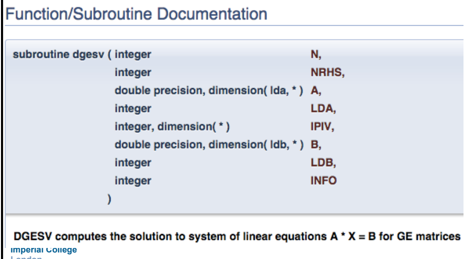
Note: this just an excerpt, there are 28 possible matrix types in total.

So, DGESV is a double precision routine for SolVing (SV) systems with general matrices

Imperial College London

Using lapack

2. **Look at documentation for subroutine to see what input is needed and how output is returned**
   - **googling "lapack dgesv" takes me to: http://www.netlib.org/lapack/explore-html/d8/d72/dgesv_8f.html**
   - **Eight variables in subroutine header (are they input and/or output?)**



Function/Subroutine Documentation

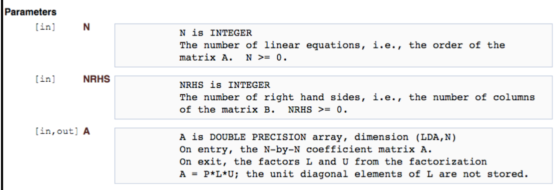
```
subroutine dgesv ( integer                                N,
                   integer                                NRHS,
                   double precision, dimension( lda, * )  A,
                   integer                                LDA,
                   integer, dimension( * )                IPIV,
                   double precision, dimension( ldb, * )  B,
                   integer                                LDB,
                   integer                                INFO
                 )
```

DGESV computes the solution to system of linear equations A * X = B for GE matrices

Imperial College London

Using lapack

2. **Look at documentation for subroutine to see what input is needed and how output is returned**
   - **googling "lapack dgesv" takes me to: http://www.netlib.org/lapack/explore-html/d8/d72/dgesv_8f.html**
   - **Eight variables in subroutine header (are they input and/or output?)**



Parameters

[in] **N**
```
N is INTEGER
The number of linear equations, i.e., the order of the
matrix A.  N >= 0.
```

[in] **NRHS**
```
NRHS is INTEGER
The number of right hand sides, i.e., the number of columns
of the matrix B.  NRHS >= 0.
```

[in,out] **A**
```
A is DOUBLE PRECISION array, dimension (LDA,N)
On entry, the N-by-N coefficient matrix A.
On exit, the factors L and U from the factorization
A = P*L*U; the unit diagonal elements of L are not stored.
```

- **N and NRHS are input, A is input which is modified so the LU decomposition is returned.**

Imperial College London

Using lapack

2. Look at documentation for subroutine to see what input is needed and how output is returned
   - googling "lapack dgesv" takes me to: http://www.netlib.org/lapack/explore-html/d8/d72/dgesv_8f.html
   - Eight variables in subroutine header (are they input and/or output?)
   - Solving A X = B:
     - A must be a LDA x N matrix (though only the 1st N rows will be used)
     - B is LDB x NRHS (again only 1st N rows are used)
     - X will be N x NRHS and must be extracted from B
     - IPIV contains information about the LU decomposition

Imperial College London

Using lapack

3. Create test example to check routine is used correctly:

dgesv_test.f90: `!example illustrating use of lapack routine, dgesv, to solve AX=B`

1. Create (nonsingular) matrix A and RHS, B
2. Send copies of A and B to *dgesv* (why use copies?)
3. Extract the solution x
4. Verify that Ax = B

Imperial College London

Using lapack

3. Create test example to check routine is used correctly:

dgesv_test.f90: `!example illustrating use of lapack routine, dgesv, to solve AX=B`

1. Create (nonsingular) matrix A and RHS, B

```
!allocate arrays, create matrices A and B
   allocate(A(LDA,N),B(LDB,NRHS),IPIV(N))
   allocate(Atemp(size(A,1),size(A,2)),Btemp(LDB,NRHS))
   A(1,:) = (/1.0,2.0,3.0,4.0/)
   A(2,:) = (/4.0,3.0,2.0,1.0/)
   A(3,:) = A(1,:)**2
   A(4,:) = sqrt(A(2,:))

   B(:,1) = (/-2.0,2.0,-1.0,1.0/)
```

Imperial College London

Using lapack

3. **Create test example to check routine is used correctly:**

dgesv_test.f90: `!example illustrating use of lapack routine, dgesv, to solve AX=B`

2. **Send copies of A and B to *dgesv* (why use copies?)**

```
!solve Ax = B
   Atemp = A
   Btemp = B
   call dgesv(N, NRHS, Atemp, LDA, IPIV, Btemp, LDB, INFO)
   print *, 'INFO=',INFO
```

Imperial College
London

Using lapack

3. **Create test example to check routine is used correctly:**

dgesv_test.f90: `!example illustrating use of lapack routine, dgesv, to solve AX=B`

3. **Extract the solution x**

```
!extract soln from Btemp
   allocate(x(N,NRHS))
   x = Btemp(1:N,:)
```

Imperial College
London

Using lapack

3. **Create test example to check routine is used correctly:**

dgesv_test.f90: `!example illustrating use of lapack routine, dgesv, to solve AX=B`

4. **Verify that Ax = B**

```
print *, 'test:',matmul(A,x)-B
```

Imperial College
London

Compiling and linking to libraries

Generally, if *program.f90* uses library *libname:*

```
$ gfortran –c program.f90
$ gfortran –o program.exe program.o –llibname
```

or:

```
$ gfortran –o program.exe program.f90 –llibname
```

Note: compiler will look for libname (often a file, liblibname.a) in "standard" locations (e.g. /usr/lib)

If you have built the library elsewhere, specify the location with "-L *libpath*" option when linking. Here *libpath* specifies location of library (e.g. /users/prasun/somewhere)

Imperial College London

Compiling and linking to libraries

So, for dgesv:

```
$ gfortran –o dgesv_test.exe dgesv_test.f90 –llapack
$ ./dgesv_test.exe
```

```
INFO=           0
test:   2.1316282072803006E-014  1.7763568394002505E-015  2.8421709430404007E-014
-5.3290705182007514E-015
```

What does INFO=0 indicate?

Imperial College London

Intro to F2Py

- Major projects:
  - Variety of tasks, some more computationally intensive than others
  - F2Py:
    - Place expensive parts in Fortran routines (why?)
    - Everything else in Python
    - Use F2Py to convert Fortran routines into Python modules which are called from main Python code

Imperial College London

F2Py example

- **Simple example: call fortran function *sumxy* from python**

```
!----------------------------------------------------------------------
!function to sum two numbers provided as input
!note that the function name is a variable whose
!type should be declared in the header
function sumxy(x,y)
    implicit none
    double precision, intent(in) :: x,y
    double precision :: sumxy

    sumxy = x + y
end function sumxy

!----------------------------------------------------------------------
```

Imperial College
London

F2Py example

Simple example: call fortran function *sumxy* from python (see f2pyfunction1.f90)

```
!----------------------------------------------------------------------
!function to sum two numbers provided as input
!note that the function name is a variable whose
!type should be declared in the header
function sumxy(x,y)
    implicit none
    double precision, intent(in) :: x,y
    double precision :: sumxy

    sumxy = x + y
end function sumxy
!----------------------------------------------------------------------
```

Use f2py to “compile” this function:

```
$ f2py –c f2pyfunction1.f90 –m f1
```

```
$ ls f1.so
f1.so
```

f1.so **is a *shared object file* which can be imported in python**

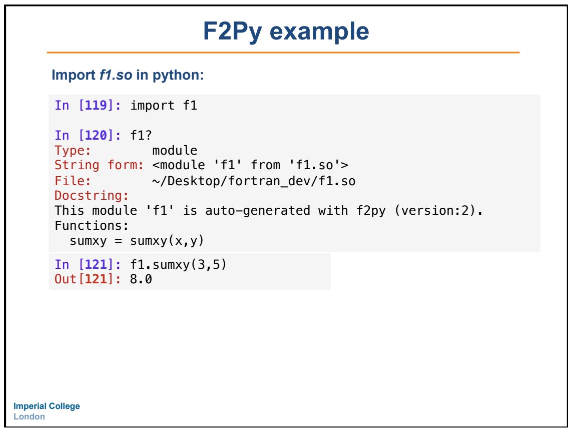
Imperial College
London

F2Py example

Import *f1.so* in python:

```
In [119]: import f1

In [120]: f1?
Type:        module
String form: <module 'f1' from 'f1.so'>
File:        ~/Desktop/fortran_dev/f1.so
Docstring:
This module 'f1' is auto-generated with f2py (version:2).
Functions:
  sumxy = sumxy(x,y)
```

Imperial College
London



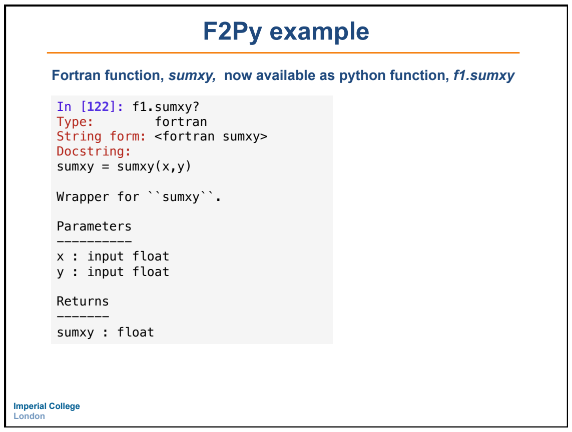
F2Py example

Import *f1.so* in python:

```
In [119]: import f1

In [120]: f1?
Type:        module
String form: <module 'f1' from 'f1.so'>
File:        ~/Desktop/fortran_dev/f1.so
Docstring:
This module 'f1' is auto-generated with f2py (version:2).
Functions:
  sumxy = sumxy(x,y)
```

```
In [121]: f1.sumxy(3,5)
Out[121]: 8.0
```

Imperial College
London

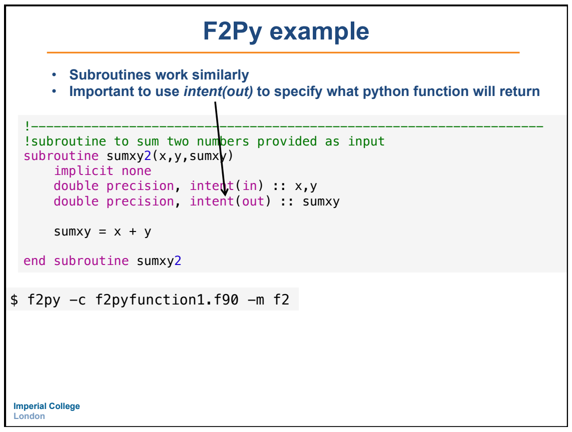


F2Py example

Fortran function, *sumxy*, now available as python function, *f1.sumxy*

```
In [122]: f1.sumxy?
Type:        fortran
String form: <fortran sumxy>
Docstring:
sumxy = sumxy(x,y)

Wrapper for ``sumxy``.

Parameters
----------
x : input float
y : input float

Returns
-------
sumxy : float
```

Imperial College
London



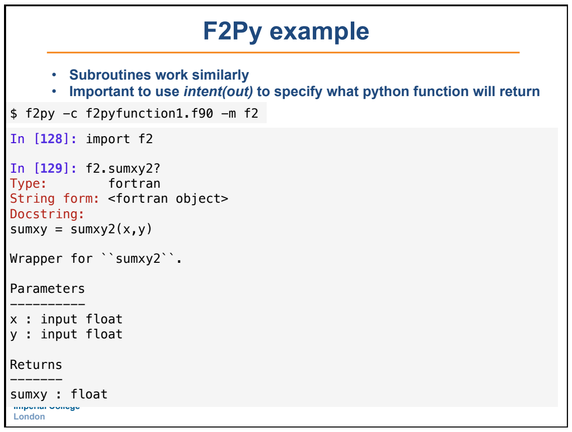
F2Py example

- **Subroutines work similarly**
- **Important to use *intent(out)* to specify what python function will return**

```
!---------------------------------------------------------------------
!subroutine to sum two numbers provided as input
subroutine sumxy2(x,y,sumxy)
    implicit none
    double precision, intent(in) :: x,y
    double precision, intent(out) :: sumxy

    sumxy = x + y

end subroutine sumxy2
```

```
$ f2py -c f2pyfunction1.f90 -m f2
```

Imperial College
London



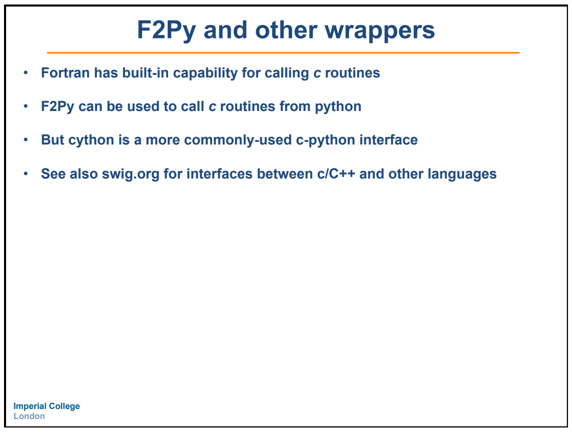
F2Py example

- Subroutines work similarly
- Important to use *intent(out)* to specify what python function will return

```
$ f2py –c f2pyfunction1.f90 –m f2
```

```
In [128]: import f2

In [129]: f2.sumxy2?
Type:        fortran
String form: <fortran object>
Docstring:
sumxy = sumxy2(x,y)

Wrapper for ``sumxy2``.

Parameters
----------
x : input float
y : input float

Returns
-------
sumxy : float
```

Imperial College
London



F2Py and other wrappers

- Fortran has built-in capability for calling *c* routines
- F2Py can be used to call *c* routines from python
- But cython is a more commonly-used c-python interface
- See also swig.org for interfaces between c/C++ and other languages

Imperial College
London