Introduction to Numerical Python

Matthijs Douze

SED Grenoble

Inria

June 2014

What should I talk about?

- Python (and numpy) is very rich
  - threshold to make a new library is low → everyone makes libraries!
  - choose what is worthwhile to present
  - what is useful for image processing / high-performance computing / PDE / machine learning...
- Enable trainees to...
  - throw-away code, quick visualization and stats
  - research code – quick prototyping, produce tables and plots
  - build “systems” (Python is pretty good at this)
- Design choices
  - as little “magic” as possible
  - use the *most basic tool that fits the task*
  - bottom-up explanation: low-level point-of-view
  - focus on performance

About me

- At INRIA since 2005,
  - LEAR then SED engineer since 2010.
  - Mainly image processing
- Python programmer since 2001
- About 60 % of development in Python
  - rest is C, Matlab, bash, C++, OpenCL, etc.
- Technological late adopter
  - new technology only if *really, really* needed
- Never attended a training session on a technical topic
  - rather reference documentation, tutorials, ...
- not a geek (any more). Not a numpy “power user”

- attendents' introductions

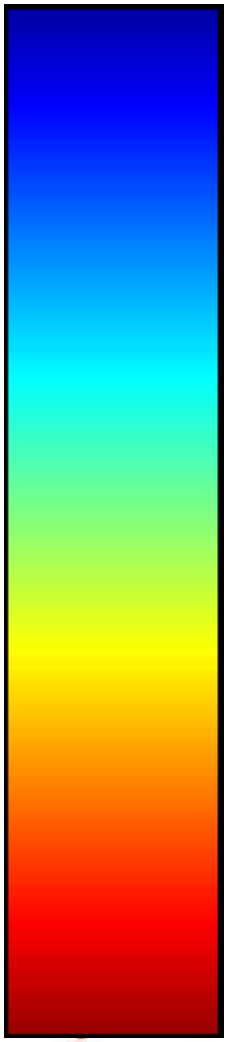
Outline

- Python/Numpy wrt. Matlab, C++
- Matrix operations
- Input/output
- Graphical output with Matplotlib
- Interfacing with C
- Writing parallel applications

Language levels

- High-level: developing time > execution time
  - **objectives**: compact, interactive “shell”, backtrace
  - **drawbacks**: slow, dependent on what it available in libraries, limited static code analysis
- Low level: execution time > developing time
  - **objectives**: fast, precise control
  - **drawbacks**: verbose, requires compilation, crashes are hard to debug
- Threshold is moving towards high level
  - Moore's law does not apply to programmers...
  - 1980: C = high level...

Operations



Command-line parsing

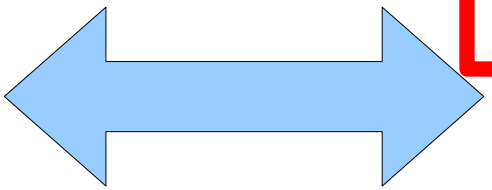
GUI

read XML/text

read binary data

compilation

matrix multiply



language levels

Langage	Speed wrt. C
Human brain	/10^9
Shell, make	/10000
TCL, perl, Python, Matlab	/100
Java, caml	/5
C, Fortran	1
SSE intrinsics, assembler	x4
CUDA, OpenCL	x50

Inria

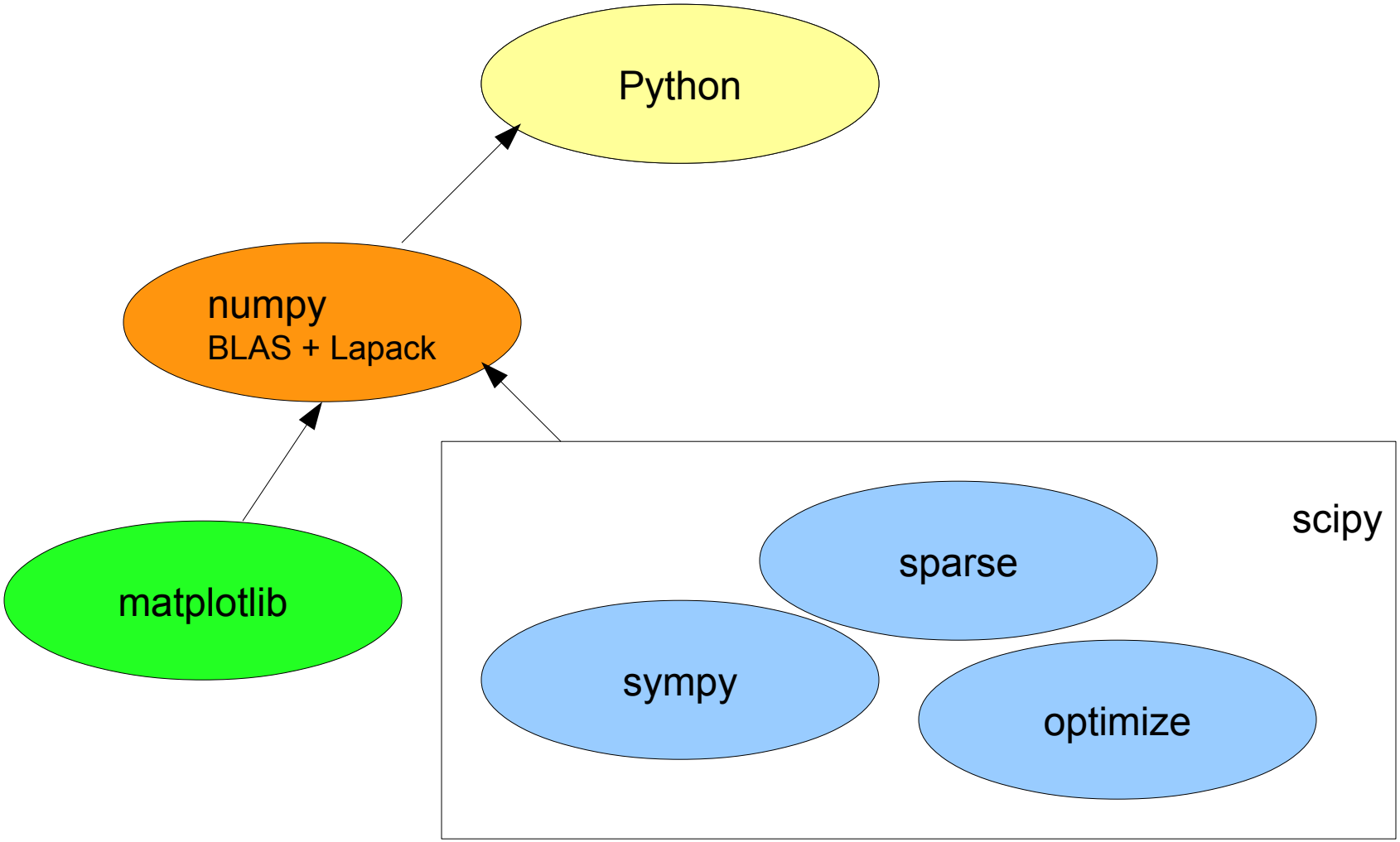
The Python language compared to Matlab

- Same “level” as Matlab
  - interpreted
  - automatic memory management
  - types are heavy, and always allocated dynamically
    - sizeof(PyIntObject) = 24 bytes
  - to optimize: use library functions to avoid loops
- more datastructures
  - hash-table
  - object-oriented
- general-purpose language
  - scripting
  - very rich standard library
    - web server, I/O
- differences
  - pass-by-reference all over the place
  - 0-based indexing
- Matlab = academic pricing 800 E + 200 E / year
  - management cost

The Python language compared to C++

- different language levels
  - difficult to compare
- rapid application development
  - less verbose
  - no compilation
  - readable error messages
    - “IndexError” + stacktrace rather than “segmentation fault”....
- optimization: think in terms of hotspots
- has a standard matrix library: numpy
  - instead of 50 matrix libraries

Numerical libraries in Python



Reminder on Python lists

- Python list
  - sequence of heterogeneous Python objects

    [2, “toto”, False, [4, 5]]

  - mutable (tuple and string are not mutable)
- Implemented as an array of pointers to python objects

Indexing: ranges and slices

- *range generates a list
- slice notation gets a subset of a list
- negative indices count from end of list

sequence	range/xrange/arange	slice within an array of size 100
0, 1, …, 99	range(100)	a[:]
10, 11, …, 99	range(10, 100)	a[10:]
0, 2, …, 98	range(0, 100, 2)	a[::2]
99, 98, …, 0	range(99, -1, -1)	a[::-1]

0-based indexing

- bounds are always
  begin <= i < end
- length of range
  end – begin
- in-memory address
  mem[i – begin]

EWD831-0

Why numbering should start at zero

To denote the subsequence of natural numbers 2, 3, ..., 12 without the pernicious three dots, four conventions are open to us:

a) $2 \leq i < 13$
b) $1 < i \leq 12$
c) $2 \leq i \leq 12$
d) $1 < i < 13$

Are there reasons to prefer one convention to the other? Yes, there are. The observation that conventions a) and b) have the advantage that the difference between the bounds as mentioned equals the length of the subsequence is valid.

Coding principles

- For performance,
  - use arrays rather than Python's lists
  - avoid loops
  - functions are implemented in C → no performance penalty
  - amortize interpretation cost on large units
- Vector/matrix programming
  - like Matlab
  - like SIMD

1D numpy arrays

- construction
  - arange
  - zeros, ones
  - random show
- homogeneous – dtype show
- can be indexed & sliced like list
  - indexing by array of indices or booleans show
  - slice assignment
- vector operations
  - unlike list
  - vector-vector operations
  - element-vector operations show
- tests
  - comparisons produce binary masks
  - reduction with all() and any()
- exercises

Outline

- Python/Numpy wrt. Matlab, C++
- Matrix operations
- Input/output
- Graphical output with Matplotlib
- Interfacing with C
- Writing parallel applications

Central numpy object : the ndarray

- 1D array is an ndarray
- Unlike Matlab: no preference for matrices
- constructors show
  - explicit
  - empty, ones, zeros
  - random
  - eye, diag
- Block manipulations
  - tile (= Matlab's repmat)
  - hstack/vstack
- indexing show
  - mixture of 1D array slices → Cartesian product
  - if one dimension is not filled in → assumed to be ":"
    - scalar index ≠ direct access in the array (unlike Matlab)
  - ndim arrays → array of values picked in the array

Data organization

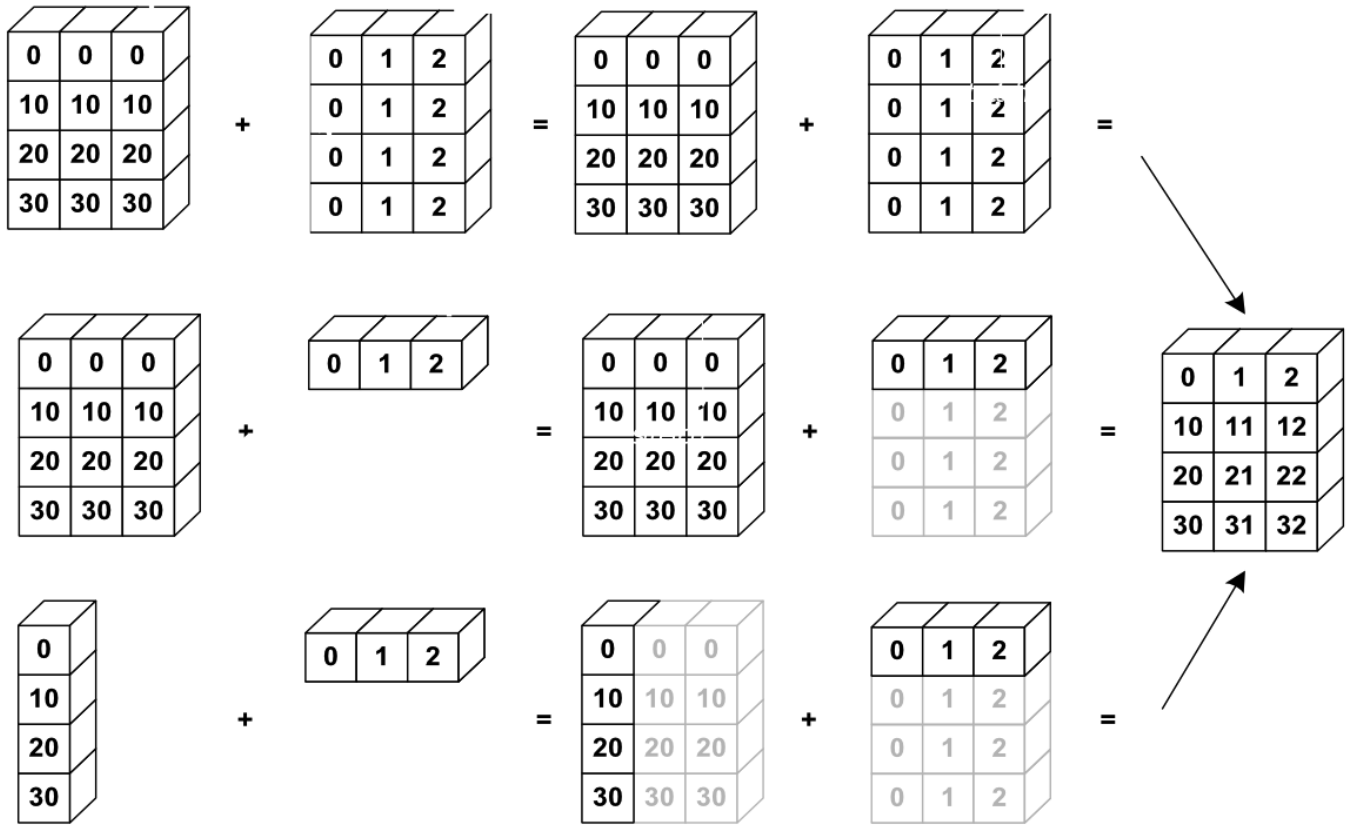
- data organization in memory (generalization of Matlab) whiteboard
  - dtype
  - shape
  - strides
  - ndim, size, itemsize, flags
- manipulate data interpretation show
  - ravel
  - slicing (+ newaxis)
  - reshape (+ implicit dimension)
  - transpose
  - view
- views / copies
  - implicit and not consistent!
  - owndata flag
  - copy

Axis operations

- operation performed on one dimension (“axis”) of the array
- Reductions
  - sum, cumsum, prod
  - max, argmax, min, argmin
  - all, any
  - mean, std
- Sorting
  - sort, argsort

Broadcasting

- extends element-wise operations to (some) arrays of different sizes
  - compare Matlab: often avoids meshgrid / bsxfun / repmat



Linear algebra

- "*" **does not** do matrix multiplication
  - use np.dot
- classical BLAS + Lapack linear algebra operators in np.linalg
  - svd, qr, cholesky, eig
  - linear least-squares, matrix (pseudo-)inverse, etc.

- exercises

Outline

- Python/Numpy wrt. Matlab, C++
- Matrix operations
- Input/output
- Graphical output with Matplotlib
- Interfacing with C
- Writing parallel applications

Input/output

- I/O, what for?
  - files
  - network sockets
  - pipes to subprocesses
- input/output = serialization
  - transform to byte array / string
  - write it to file object
- families
  - metadata?
  - binary/text?

Native formats

- Python pickle
  - serializes all python objects
  - works with numpy (but not efficient)
  - text format by default
- Numpy format
  - stores a matrix
  - binary
  - can be memory-mapped
- np.save / np.load show

Read/write text formats

- Easiest: read/write from Python
  - Python has excellent I/O
  - regexp, XML, generators
  - → Python list → numpy array
- Fastest: read with specialized functions for common formats
  - np.fromfile / np.fromstring (with sep=" ")
    - reads whitespace separated scalars
    - output = 1D array, should reshape show
  - np.loadtxt / np.savetxt
    - for matrices
    - line-oriented format, 1 line per matrix row, whitespace separated
    - comments with "#" show
- Can be used on an open file object
  - parse beginning of file with some method, end with another
  - supports on-the-fly zipping / unzipping

Read/write binary

- Only a fast option :-)
- a.tofile / a.tostring
  - binary representation of the array
  - no metadata show
- np.fromfile / np.fromstring
  - metadata must be provided (dtype)
  - reshape to relevant size
- np.memmap
  - memory mapping of a file

- Matlab .mat files
  - use scipy.io.loadmat()

- exercises

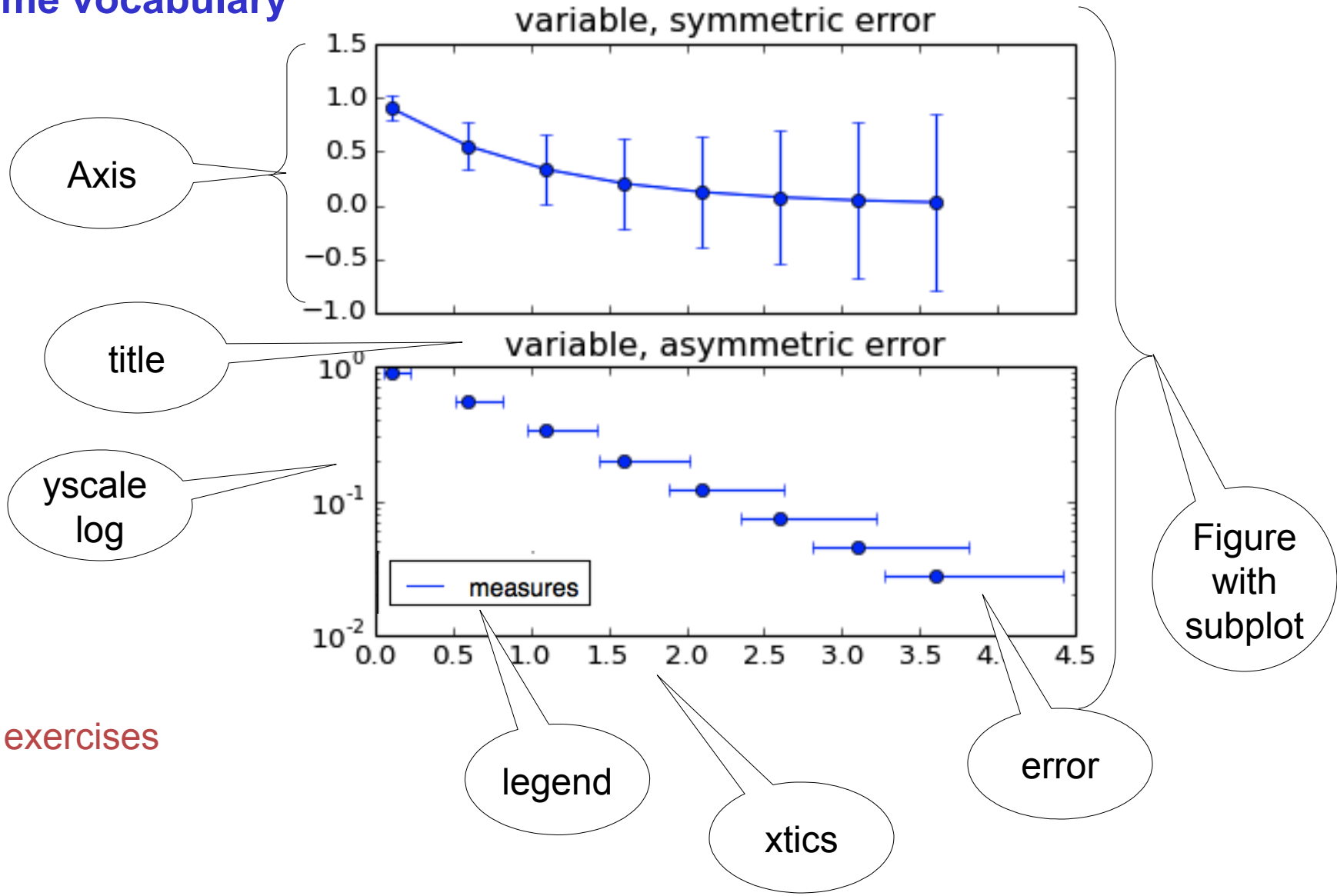
Outline

- Python/Numpy wrt. Matlab, C++
- Matrix operations
- Input/output
- Graphical output with Matplotlib
- Interfacing with C
- Writing parallel applications

Matplotlib

- Similar to Matlab show
  - plot, imshow, ...
- Two operating modes: interactive / scripts
  - pyplot.ion() / ioff()
  - pyplot.show()
- Output on screen by default show
  - to generate PDF: pyplot.savefig(“xx.pdf”)
- Images:
  - loading: matplotlib.image.imread

Some vocabulary



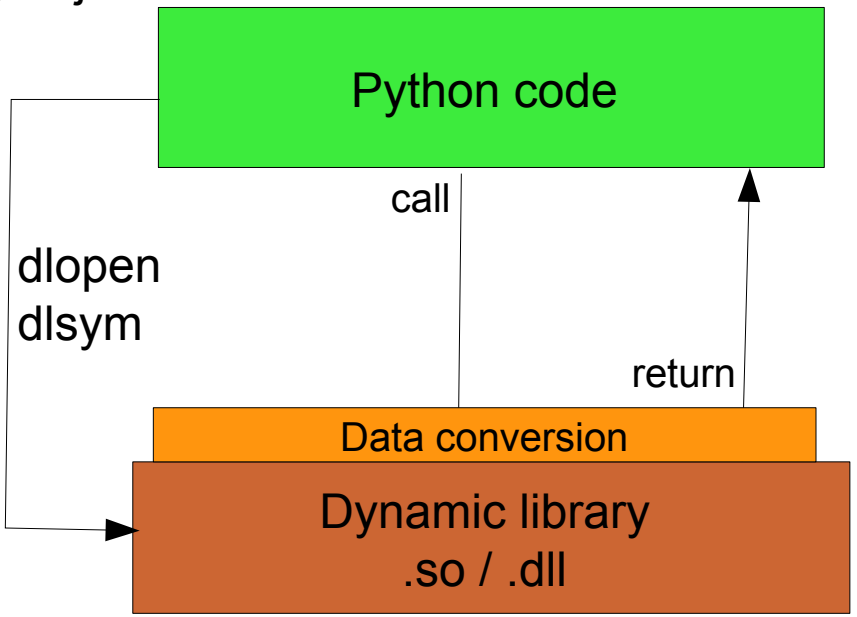
- exercises

Outline

- Python/Numpy wrt. Matlab, C++
- Matrix operations
- Input/output
- Graphical output with Matplotlib
- Interfacing with C
- Writing parallel applications

Calling C code from Python: basics

- All data (and code...) in Python is PyObject
  - typing info
  - reference count
  - object-specific info
- modules can be implemented
  - in Python
  - in C → compiled to shared lib
- very similar to Matlab/mex
- C modules
  - loaded by interpreter
  - format constraints show example



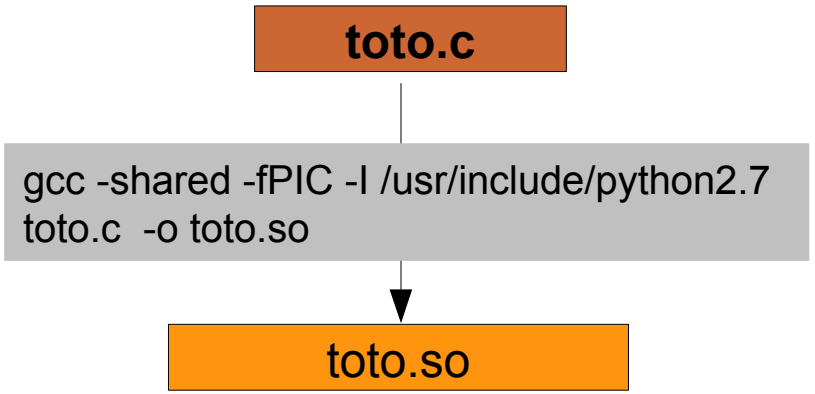
Data conversion

- Always check inputs
  - PyXXX_Check
  - convenience function: PyArg_ParseTuple show
- PyArrayObject contains same fields as the Python object + a pointer to the array data.
  - requires cast
  - be careful with strides
- PyArray results can be constructed show example
- To construct tuple results
  - symmetric of PyArg_ParseTuple: Py_BuildValue

Compiling

- Command line to produce a shared lib depends on platform
- requires access to includes (and .lib files on Windows)
  - distutils.sysconfig.get_python_inc()
  - numpy.get_include()

- setup.py can compile the module show
  - standard way of distributing and installing modules
  - heavyweight for simple projects



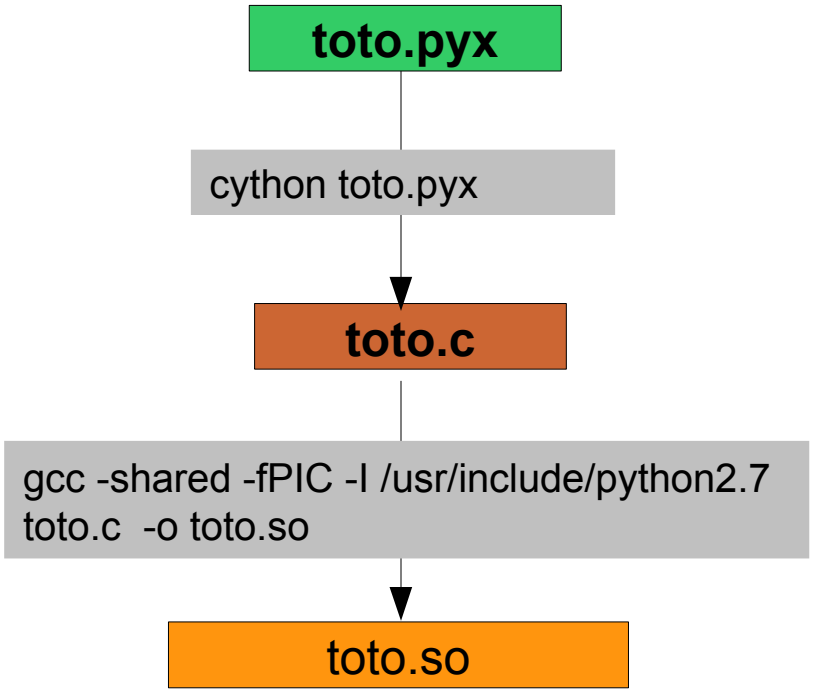
Generation of C code made easy (?)

- Tons of translators
- ctypes: call arbitrary dynamic library
  - Python does the data conversion
- scipy.weave: inline code
  - can compile on-the-fly
- pythran, shedskin:
  - converts Python to C++
- pypy:
  - a just-in-time compiler
- f2py
  - make Python interfaces for Fortran code

- cython:
  - convert annotated Python to C ←
- SWIG:
  - automatically wrap C/C++ library ←

Cython

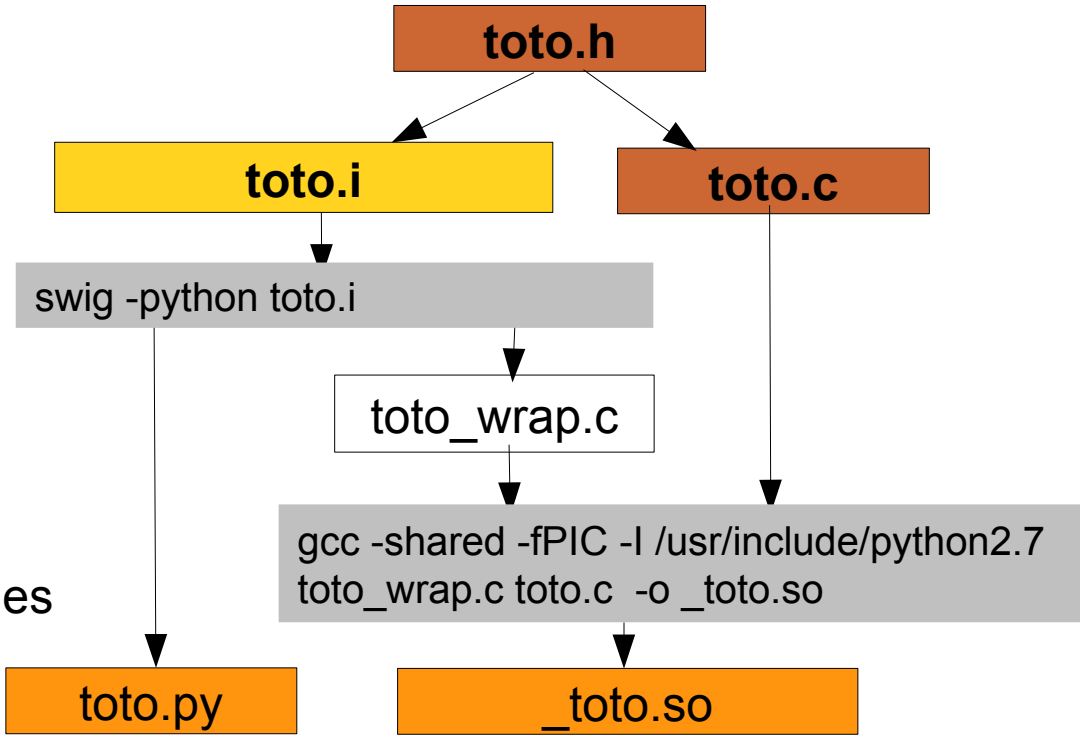
- Translates Python to C
  - support for a subset of the language
  - interesting to see how it translates... show
- Add typing information to Python code
  - then cython can optimize operations involving only typed variables
  - progressive path to optimization
- Support for numpy arrays
  - cinclude numpy



- Appropriate when starting from Python code...
  - hard to focus on low-level optimization

Simple Wrapper and Interface Generator

- Parses C headers to generate wrapper code
- Generates wrappers for all functions found in the header
  - discovers undefined functions!
- data conversions...
- reasonable defaults for standard types show
  - scalar types
  - const char *
  - struct / class
  - good support for C++ STL
- customizable for everything else
  - typemaps
- useful for existing C/C++ libraries
  - complex for small projects
- exercices



Outline

- Python/Numpy wrt. Matlab, C++
- Matrix operations
- Input/output
- Graphical output with Matplotlib
- Interfacing with C
- Writing parallel applications

Parallelization

- Use several cores/processors from the same Python code
- Useful for I/O: 1 processor per input or output
  - web server
  - user interface show
- Parallelizing heavy computations
  - multiprocessor/core machines
  - 1 processor per task
  - visible in top
- Classical usage patterns
  - fixed processor layout
  - parallel map ←
  - producer-consumer
- 3 parallelization levels (lightest → heaviest)
  - C-level, code parallelized show
  - threads
  - sub-processes

Threading

- Python can create threads
  - map to system threads
- but global interpreter lock whiteboard
  - all Python instructions are atomic
  - simplifies synchronization all over the place
  - makes multithreading pure Python code useless :-(
- in C: safe to release GIL when there is no call to the Python API
  - most numpy functions release the GIL
  - C API

    Py_BEGIN_ALLOW_THREADS / Py_END_ALLOW_THREADS
  - in cython
    - with nogil:
  - in SWIG

```
%exception {
  Py_BEGIN_ALLOW_THREADS
  $action
  Py_END_ALLOW_THREADS
}
```

- multiprocessing.dummy.Pool
  - implements a map function

Multiprocessing

- multiprocessing avoids the GIL
  - duplicates the whole Python process
  - not same address space → requires serialization/deserialization or shared mem
- multiprocessing.Pool has a map function
- quick-and-dirty tests
  - bad behavior with exceptions
  - performance penalty due to serialization and context switches

- exercise

End

- Longer exercises