ETH zürich

# Introduction to Scientific Computation
# Lecture 9
# Fall 2018

## Sparse matrices, packaging

Mikhail Usvyatsov December 5, 2018

## Intro

Dense matrix:

- Mathematical object
- Good DS for storing 2D arrays

Properties:

- Memory allocated once for all items
- Fast access to entries

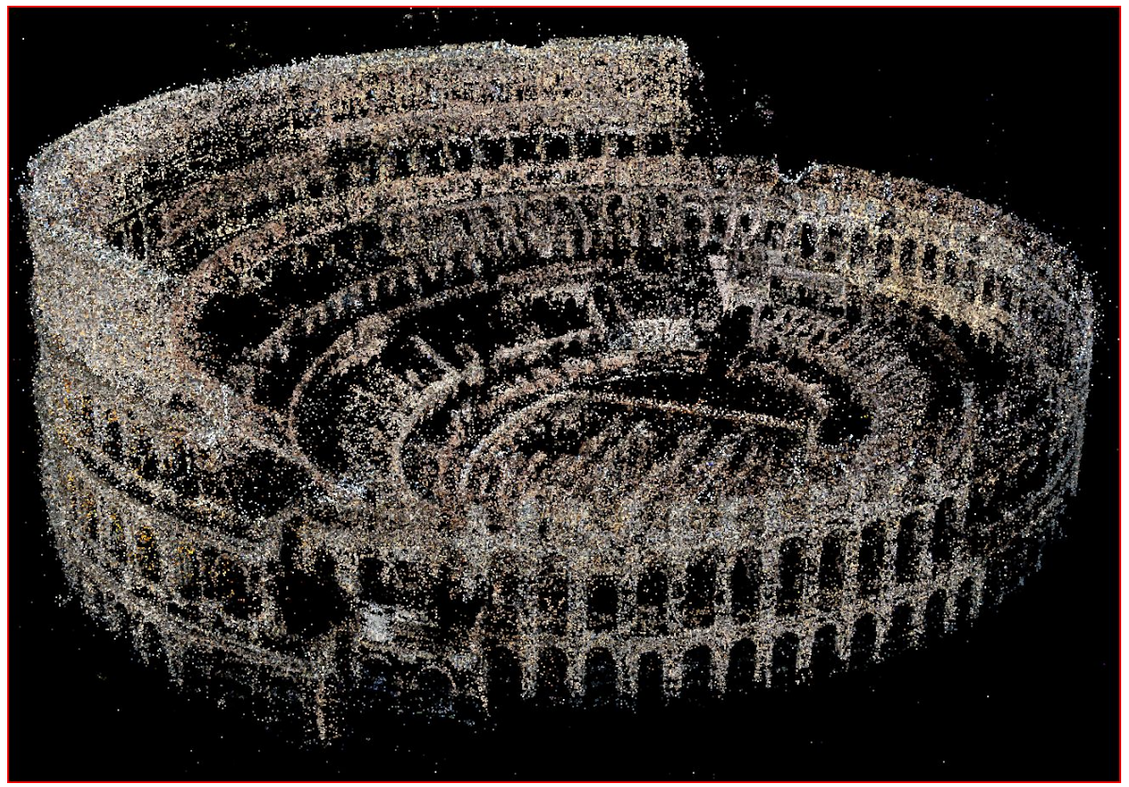
$$A_{dense} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25} \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \\ A_{41} & A_{42} & A_{43} & A_{44} & A_{45} \\ A_{51} & A_{52} & A_{53} & A_{54} & A_{55} \end{bmatrix}$$

## Sparse Matrices

Often the data around is sparse

It means:

- Data is almost empty
- Storing all the zeros is wasteful
- Think **compression**



Cons:

- Usually slow individual elements access,

But depends on storage algorithm

## Storage schemes

- csc_matrix: Compressed Sparse Column format
- csr_matrix: Compressed Sparse Row format
- bsr_matrix: Block Sparse Row format
- lil_matrix: List of Lists format
- dok_matrix: Dictionary of Keys format
- coo_matrix: COOrdinate format (aka IJV, triplet format)
- dia_matrix: DIAgonal format

Each suitable for some tasks

```
import scipy.sparse as sps
```

**warning** for NumPy users:

- the multiplication with '*' is the *matrix multiplication* (dot product)
- not part of NumPy!
  - passing a sparse matrix object to NumPy functions expecting ndarray/matrix does not work

## Implementation details

Subclasses of **spmatrix**

- default implementation of arithmetic operations
  - always converts to CSR
  - subclasses override for efficiency

- shape, data type set/get
  - nonzero indices
  - format conversion, interaction with NumPy (*toarray()*, *todense()*)

- attributes:
  - *mtx.A* - same as mtx.toarray()
  - *mtx.T* - transpose (same as mtx.transpose())
  - *mtx.H* - Hermitian (conjugate) transpose
  - *mtx.real* - real part of complex matrix
  - *mtx.imag* - imaginary part of complex matrix
  - *mtx.size* - the number of nonzeros (same as self.getnnz())
  - *mtx.shape* - the number of rows and columns (tuple)
- data usually stored in NumPy arrays

## DIAgonal format

- Simple
- Diagonals in dense numpy format, shape (**n_diags, length**)
- Fixed length -> waste space a bit when far from main diagonal
- Offset for each diagonal
  - 0 is the main diagonal
  - negative offset = below
  - positive offset = above
- Fast matrix * vector (sparsetools)
- Fast and easy item-wise operations
- No slicing, no individual item access

```
offset: row

  2:  9
  1:  --10------
  0:  1  . 11  .
 -1:  5  2  . 12
 -2:  .  6  3  .
 -3:  .  .  7  4
      ---------8
```

## List of Lists format (LIL)

- Row-based linked list
- Each row is a Python list (sorted) of column indices of non-zero elements
- Rows stored in a NumPy array (*dtype=np.object*)
- Non-zero values data stored analogously
- Slow arithmetics, slow column slicing due to being row-based
- Used when sparsity pattern is not known apriori or changes

```
>>> print(mtx)
  (0, 1)  1.0
  (0, 2)  1.0
  (0, 3)  1.0
  (1, 1)  1.0
  (1, 3)  1.0
>>> mtx.todense()
matrix([[ 0.,  1.,  1.,  1.,  0.],
        [ 0.,  1.,  0.,  1.,  0.],
        [ 0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.]])
```

## Dictionary of Keys format

- Subclass of Python dict
  - keys are *(row, column)* index tuples (no duplicate entries allowed)
  - values are corresponding non-zero values

- Efficient for constructing sparse matrices incrementally

- Efficient O(1) access to individual elements
- Flexible slicing, changing sparsity structure is efficient
- Can be efficiently converted to a coo_matrix once constructed
- Slow arithmetics (*for* loops with *dict.iteritems()*)
- Used when sparsity pattern is not known apriori or changes

## Coordinate format (COO)

- also known as the ‘ijv’ or ‘triplet’ format
  - three NumPy arrays: *row*, *col*, *data*
  - *data[i]* is value at *(row[i], col[i])* position
  - permits duplicate entries

- fast format for constructing sparse matrices

- very fast conversion to and from CSR/CSC formats
- fast matrix * vector (sparsetools)
- fast and easy item-wise operations
  - manipulate data array directly (fast NumPy machinery)

- no slicing, no arithmetics (directly)
- use:
  - facilitates fast conversion among sparse formats
  - when converting to other format (usually CSR or CSC), duplicate entries are summed together

## Coordinate format (COO)

```
>>> row = np.array([0, 3, 1, 0])
>>> col = np.array([0, 3, 1, 2])
>>> data = np.array([4, 5, 7, 9])
>>> mtx = sparse.coo_matrix((data, (row, col)), shape=(4, 4))
>>> mtx
<4x4 sparse matrix of type '<... 'numpy.int64'>'
        with 4 stored elements in COOrdinate format>
>>> mtx.todense()
matrix([[4, 0, 9, 0],
        [0, 7, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 5]])
```

## Compressed Sparse Row format (CSR)

- row oriented
  - three NumPy arrays: *indices*, *indptr*, *data*
    - *indices* is array of column indices
    - *data* is array of corresponding nonzero values
    - *indptr* points to row starts in *indices* and *data*
    - length is *n_row + 1*, last item = number of values = length of both *indices* and *data*
    - nonzero values of the *i*-th row are *data[indptr[i]:indptr[i+1]]* with column indices *indices[indptr[i]:indptr[i+1]]*
    - item *(i, j)* can be accessed as *data[indptr[i]+k]*, where *k* is position of *j* in *indices[indptr[i]:indptr[i+1]]*
- fast matrix vector products and other arithmetics (sparsetools)
- efficient row slicing, row-oriented operations
- slow column slicing, expensive changes to the sparsity structure
- use:
  - actual computations (most linear solvers support this format)

## Compressed Sparse Column format (CSR)

- row oriented
  - three NumPy arrays: *indices*, *indptr*, *data*
    - *indices* is array of row indices
    - *data* is array of corresponding nonzero values
    - *indptr* points to column starts in *indices* and *data*
    - length is *n_col + 1*, last item = number of values = length of both *indices* and *data*
    - nonzero values of the *i*-th column are *data[indptr[i]:indptr[i+1]]* with row indices *indices[indptr[i]:indptr[i+1]]*
    - item *(i, j)* can be accessed as *data[indptr[i]+k]*, where *k* is position of *j* in *indices[indptr[i]:indptr[i+1]]*
- fast matrix vector products and other arithmetics (sparsetools)
- efficient column slicing, column-oriented operations
- slow row slicing, expensive changes to the sparsity structure
- use:
  - actual computations (most linear solvers support this format)

# Packaging

## What is a “package”?

- In a broad sense, anything you install using your package manager
- Some kinds of packages have implied behaviour and requirements
- Unfortunate overloading: python “package”: a folder that python imports

## Package Managers and Repos

- Many package managers: some OS specific:
  - apt, yum, dnf, chocolatey, homebrew, etc.
- Some language specific:
  - NPM, pip, RubyGems
- And there are many online repositories of packages:
  - PyPI, anaconda.org, CRAN, CPAN

But they all contain:

- Some form of dependency management
- Artifact and/or source repository

The idea is that you install something, and have it *just work*.

# Package types:

A package can be essentially in two forms:

- source
- binary

Focusing now on the Python world:
As Python is a dynamic language, this distinction can get a bit blurred:
There is little difference between a source and binary package *for a pure python package*
But if there is any compiled code in there, building from source can be a challenge:

- Binary packages are very helpful

# Source Packages

A source package is all the source code required to build the package.
Package managers (like pip) can automatically build your package from source.
**But:**

- Your system needs the correct tools installed, compilers, build tools, etc
- You need to have the dependencies available
- Sometimes it takes time, sometimes a LONG time

# Binary Packages

A collection of code all ready to run.

- Everything is already compiled and ready to go – makes it easy.

**But:**

- It's likely to be platform dependent
- May require dependencies to be installed

# Python Packaging

There are two package managers widely used for Python.

**pip:** The "official" solution.

- Pulls packages from PyPI
- Handles both source and binary packages (wheels)
- Python only

**conda:** Widely used in the scipy community.

- Pulls packages from anaconda.org
- Binary only (does not compile code when installing)
- Supports other languages / libraries: C, Fortran, R, Perl, Java (anything, really)
- Manages Python itself!

# WHY ARE WE DOING THIS?

One of the most powerful things about coding for the sciences is that it costs nothing to re-use code we've written in the past, allowing us to build on past work rather than starting over every project or paper.

However, one practice we see again and again is copying and pasting code from one project into another. Sometimes it will just be a function, other times (coughMATLABcough) it's files.

Assembling code in packages makes it really easy to re-use old code: all the scripts and functions end up in a central location and can be called and imported from anywhere on the computer - just like the famous packages `numpy` or `matplotlib`.

ETH zürich

# SETTING UP

## 1. THE BASICS

The most basic directory structure for a Python package looks like this:

```
project
|
|__ setup.py
|
|__ myPackage
     |
     |_  somePython.py
     |_  __init__.py
```

But at the moment, we've just got some flat files.

```
project
|
|__ norms.py
|__ metrics.py
```

So, the first step is to move files around. First comes the hardest part: choosing a package name. I'll call mine `measure`. Create a directory with that name, and move the python files in there.

```
project
|
|__ measure
     |__ norms.py
     |__ metrics.py
```

There is one more crucial file: `__init__.py` lets the Python interpreter know that there are importable modules in this directory. This is the script that gets run when you execute `import measure`. For more about what you can do with modules, you can see the (Python docs)[https://docs.python.org/3/tutorial/modules.html]. After adding `__init__.py`, the project directory should be

```
project
|
|__ measure
     |__ __init__.py
     |__ norms.py
     |__ metrics.py
```

# 2. SETUP.PY

At this point, the library can be imported if we're in the same directory, but it isn't a package. To let setuptools and `pip` know how to handle it, we need to add the `setup.py` file.

A very basic version of `setup.py` is

```
from setuptools import setup

setup(
    # Whatever arguments you need/want
)
```

If you were to run that file, you'd get a whole bunch of warnings, and nothing would actually get packaged. As a bare minimum, here is our `setup.py` for it to work.

```
from setuptools import setup

setup(
    # Needed to silence warnings (and to be a worthwhile package)
    name='Measurements',
    url='https://github.com/jladan/package_demo',
    author='John Ladan',
    author_email='jladan@uwaterloo.ca',
    # Needed to actually package something
    packages=['measure'],
    # Needed for dependencies
    install_requires=['numpy'],
    # *strongly* suggested for sharing
    version='0.1',
    # The license can be anything you like
    license='MIT',
    description='An example of a python package from pre-existing code',
    # We will also need a readme eventually (there will be a warning)
    # long_description=open('README.txt').read(),
)
```

```
project
|
|__ setup.py
|__ MANIFEST.in
|
|__ measure
|    |__ __init__.py
|    |__ norms.py
|    |__ metrics.py
|
|__ tests
|    |__ test_suite.py
|
|__ doc
|    |__ docs.rst
|
|__ scripts
|    |__ hello.py
|
|__ README.txt
|__ CHANGES.txt
|__ LICENSE.txt
```