ETH zürich

Introduction to Scientific Computation
Lecture 9
Fall 2023

Symbolic computations, graph computations, sparse matrices

Mikhail Usvyatsov **December 6, 2023**

SymPy

1. Evaluate expressions with arbitrary precision.
2. Perform algebraic manipulations on symbolic expressions.
3. Perform basic calculus tasks (limits, differentiation and integration) with symbolic expressions.

4. Solve polynomial and transcendental equations.
5. Solve some differential equations.

What is SymPy? SymPy is a Python library for symbolic mathematics. It aims to be an alternative to systems such as Mathematica or Maple while keeping the code as simple as possible and easily extensible. SymPy is written entirely in Python and does not require any external libraries.

3 basic numerical types

- Real
- Rational
- Integer

```
>>> import sympy as sym
>>> a = sym.Rational(1, 2)

>>> a
1/2

>>> a*2
1
```

Arbitrary precision of math constants

```
>>> sym.pi**2
pi**2

>>> sym.pi.evalf()
3.14159265358979

>>> (sym.pi + sym.exp(1)).evalf()
5.85987448204884
```

Algebraic manipulations

Expand

```
>>> sym.expand((x + y) ** 3)
 3       2         2    3
x  + 3*x *y + 3*x*y  + y
>>> 3 * x * y ** 2 + 3 * y * x ** 2 + x ** 3 + y ** 3
 3       2         2    3
x  + 3*x *y + 3*x*y  + y
```

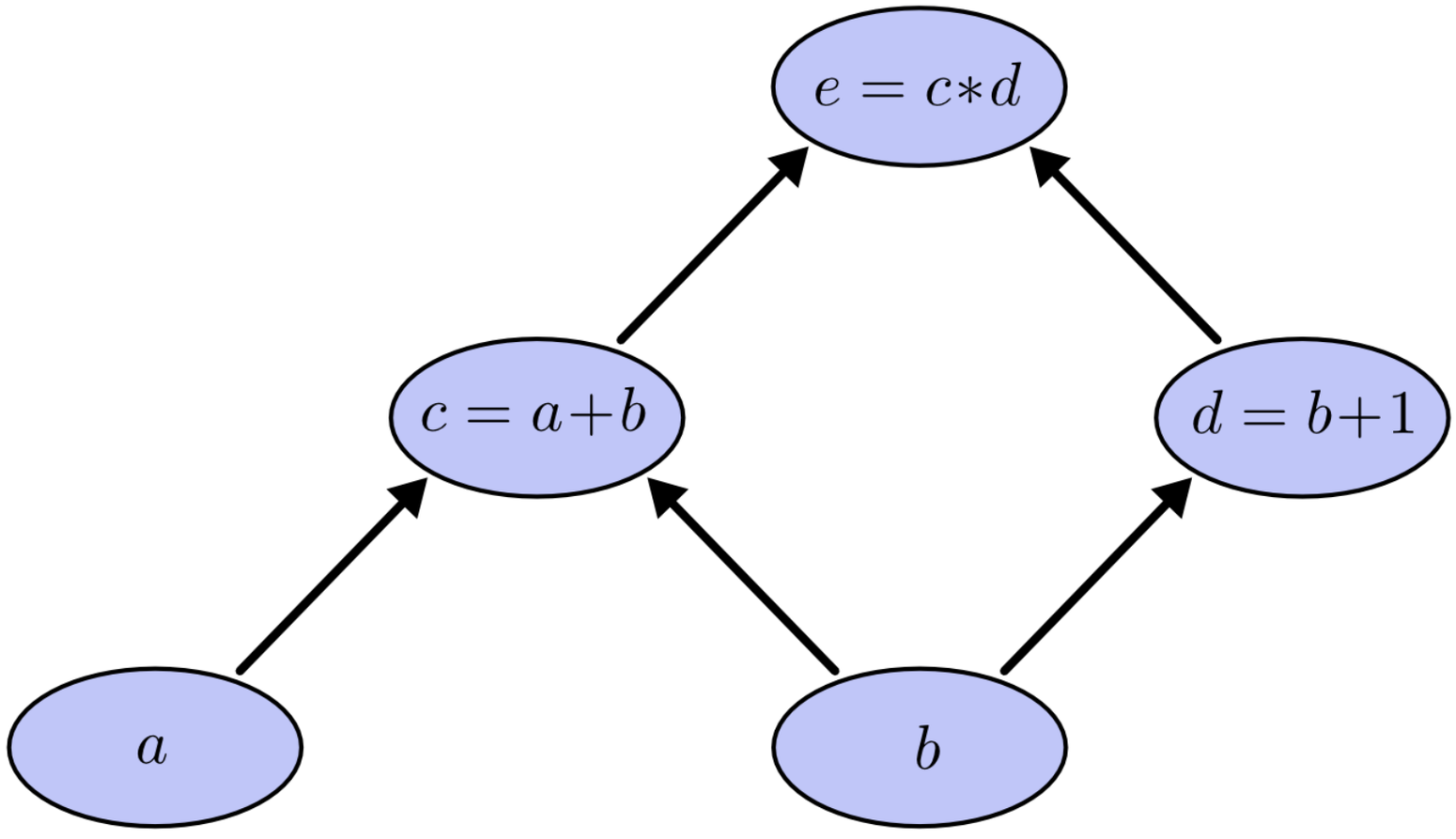
Simplify

```
>>> sym.simplify((x + x * y) / x)
y + 1
```

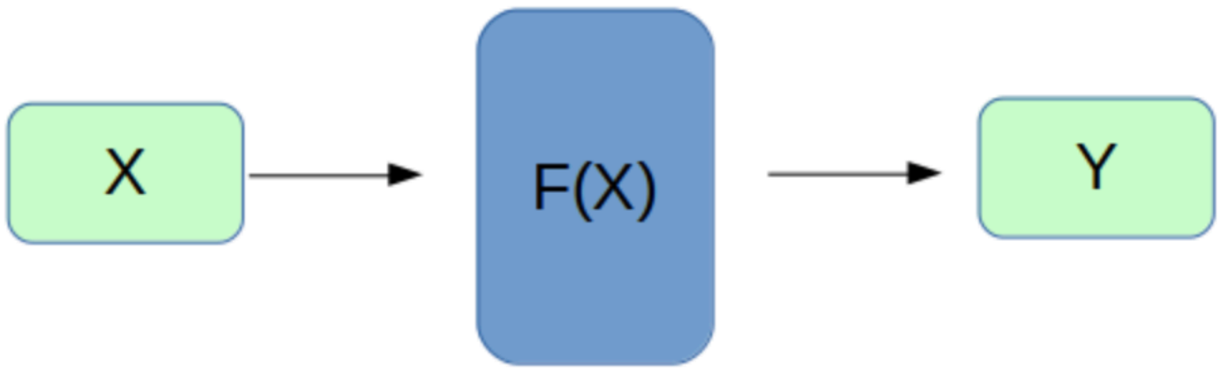
Calculus

- lim
- diff
- series
- Int
- eqs + system of eqs

Graph Computations



Graph Computations



1. Define the graph, that solves the problem you need
2. You provide the graph with the input

e.g.

```
x = 5
```

or

```
x = 5 * np.ones(100)
```

Jax

Is a library that allows to perform symbolic computations.
Some important features:

- Use of GPU / TPU for computations
- Huge community
- Supported by Google

It allows

- Symbolically define mathematical functions
- Automatically derive gradient expressions
- Execute expression
- Fast scaling

Graph Computations

Pros:

- Optimised computation possible
- Faster execution
- Can reuse graphs

Cons:

- Need to maintain graph
- Can be hard to debug
- Different way of thinking

Graph Computations

Pros:

- Optimised computation possible
- Faster execution
- Can reuse graphs

Cons:

- Need to maintain graph
- Can be hard to debug
- Different way of thinking

Tensorflow 2.0, jax and pytorch fixed all the cons

Dense matrices

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

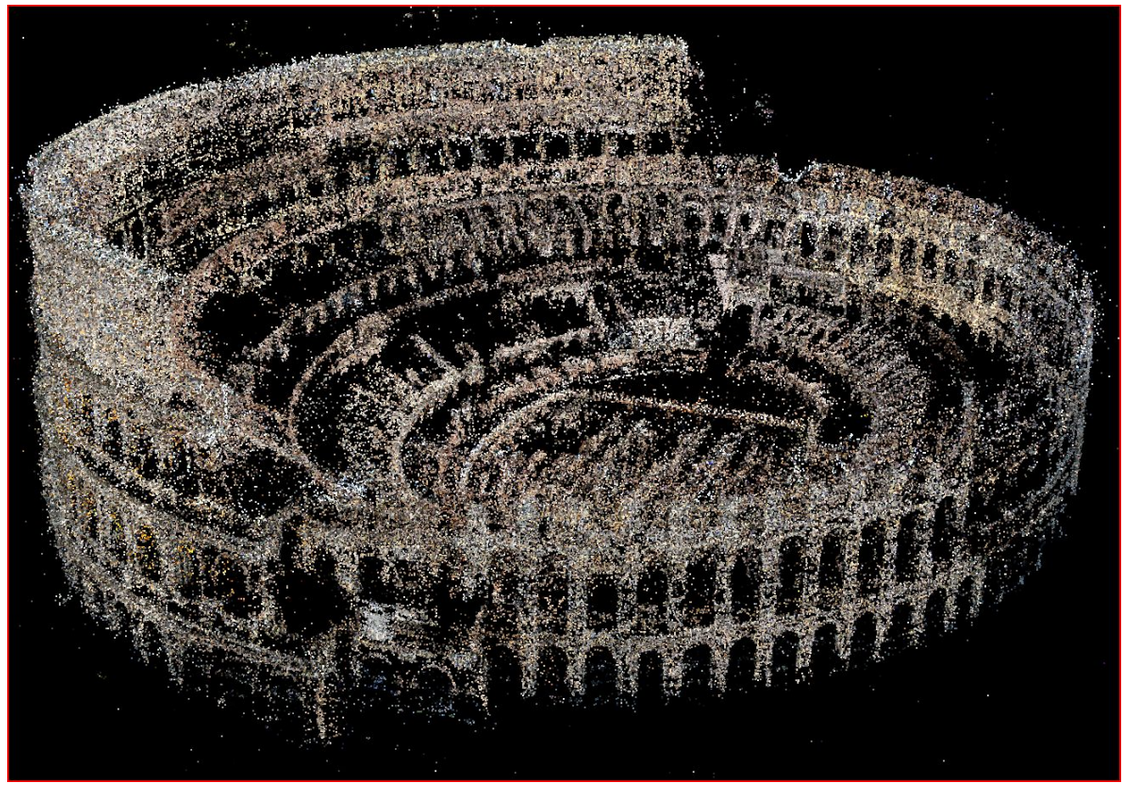
It means:

- 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 (CSC)

- 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)