Matrix Operations (in Chemistry)

Guido Falk von Rudorff

Motivation

Eigenvectors $\mathbf{v}_i$

- Normal modes, vibrational spectra
- Principal Components Analysis (PCA)

Eigenvalues λ_i

- Energy levels of an Hamiltonian
- Molecular graph comparison

Matrix inverse $\mathbf{A}^{-1}$

- Solving systems of linear equations
- Machine learning methods

Matrix multiplication

- Symmetry operations

Eigenvectors and eigenvalues

Eigenvectors

- Multiplication with **A** leaves direction unchanged
- Invariants of a (symmetry) transformation
- Must be non-zero

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$$

$$(\mathbf{A} - \lambda_i \mathbf{I})\mathbf{v}_i = 0$$

Eigenvalues

- Linked to eigenvectors
- Set of all is called *spectrum*

Normal modes

Hessian

- Local curvature of the potential energy
- If in local minimum: vibrational modes

Normal modes

- Mass-weighted Hessian $F_{ij}=H_{ij}(M_iM_j)^{-1/2}$
- Eigenvalues: Frequencies
- Eigenvectors: Modes

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 E}{\partial x_1^2} & \dfrac{\partial^2 E}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 E}{\partial x_1\,\partial x_n} \\ \dfrac{\partial^2 E}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 E}{\partial x_2^2} & \cdots & \dfrac{\partial^2 E}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 E}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 E}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 E}{\partial x_n^2} \end{bmatrix},$$

Normal modes in Python


```python
from pyscf import gto
import numpy as np
from pyscf.data import nist

# Build molecule
mol = gto.M(atom=[['O', 0., 0., 0.106817],
                  ['H', 0., -0.785198 , -0.427268],
                  ['H', 0., 0.785198 , -0.427268]],
            basis='6-31G',
            verbose=0)

# Do Hartree-Fock calculation
mf = mol.RHF().run()

# Calculate Hessian
hessian = mf.Hessian().kernel()
# Change the array layout
hessian = hessian.transpose(0, 2, 1, 3).reshape(mol.natm * 3, mol.natm * 3)

# Mass-weight the Hessian
atom_masses = mol.atom_mass_list(isotope_avg=True)
atom_masses = np.repeat(atom_masses, 3)
Mhalf = 1/np.sqrt(np.outer(atom_masses, atom_masses))
weighted_hessian = hessian * Mhalf

# Calculate eigenvalues and eigenvectors
force_constants, modes = np.linalg.eigh(weighted_hessian)

# Change units for wavenumbers
frequencies = np.sqrt(np.abs(force_constants))
to_wavenumbers = (nist.HARTREE2J / (nist.ATOMIC_MASS * nist.BOHR_SI**2))**.5
to_wavenumbers *= 1/(2 * np.pi) / nist.LIGHT_SPEED_SI * 1e-2
to_wavenumbers * np.sort(frequencies)[-3:]
```

```
array([1736.71755055, 3988.23212533, 4145.21010618])
```

Imports

- PySCF: Quantum chemistry calculations
- Numpy: Mathematical operations

Normal modes in Python


```
from pyscf import gto
import numpy as np
from pyscf.data import nist

# Build molecule
mol = gto.M(atom=[['O', 0., 0., 0.106817],
                  ['H', 0., -0.785198 , -0.427268],
                  ['H', 0., 0.785198 , -0.427268]],
            basis='6-31G',
            verbose=0)

# Do Hartree-Fock calculation
mf = mol.RHF().run()

# Calculate Hessian
hessian = mf.Hessian().kernel()
# Change the array layout
hessian = hessian.transpose(0, 2, 1, 3).reshape(mol.natm * 3, mol.natm * 3)

# Mass-weight the Hessian
atom_masses = mol.atom_mass_list(isotope_avg=True)
atom_masses = np.repeat(atom_masses, 3)
Mhalf = 1/np.sqrt(np.outer(atom_masses, atom_masses))
weighted_hessian = hessian * Mhalf

# Calculate eigenvalues and eigenvectors
force_constants, modes = np.linalg.eigh(weighted_hessian)

# Change units for wavenumbers
frequencies = np.sqrt(np.abs(force_constants))
to_wavenumbers = (nist.HARTREE2J / (nist.ATOMIC_MASS * nist.BOHR_SI**2))**.5
to_wavenumbers *= 1/(2 * np.pi) / nist.LIGHT_SPEED_SI * 1e-2
to_wavenumbers * np.sort(frequencies)[-3:]
```

```
array([1736.71755055, 3988.23212533, 4145.21010618])
```

Define a molecule: water

- Coordinates (need to be a minimum!)
- Basis set: measure of accuracy

Normal modes in Python


```python
from pyscf import gto
import numpy as np
from pyscf.data import nist

# Build molecule
mol = gto.M(atom=[['O', 0., 0., 0.106817],
                  ['H', 0., -0.785198 , -0.427268],
                  ['H', 0., 0.785198 , -0.427268]],
            basis='6-31G',
            verbose=0)

# Do Hartree-Fock calculation
mf = mol.RHF().run()

# Calculate Hessian
hessian = mf.Hessian().kernel()
# Change the array layout
hessian = hessian.transpose(0, 2, 1, 3).reshape(mol.natm * 3, mol.natm * 3)

# Mass-weight the Hessian
atom_masses = mol.atom_mass_list(isotope_avg=True)
atom_masses = np.repeat(atom_masses, 3)
Mhalf = 1/np.sqrt(np.outer(atom_masses, atom_masses))
weighted_hessian = hessian * Mhalf

# Calculate eigenvalues and eigenvectors
force_constants, modes = np.linalg.eigh(weighted_hessian)

# Change units for wavenumbers
frequencies = np.sqrt(np.abs(force_constants))
to_wavenumbers = (nist.HARTREE2J / (nist.ATOMIC_MASS * nist.BOHR_SI**2))**.5
to_wavenumbers *= 1/(2 * np.pi) / nist.LIGHT_SPEED_SI * 1e-2
to_wavenumbers * np.sort(frequencies)[-3:]
```

```
array([1736.71755055, 3988.23212533, 4145.21010618])
```

Do the actual calculation

- Restricted Hartree-Fock as a method
- For large molecules: expensive

Normal modes in Python


```python
from pyscf import gto
import numpy as np
from pyscf.data import nist

# Build molecule
mol = gto.M(atom=[['O', 0., 0., 0.106817],
                  ['H', 0., -0.785198 , -0.427268],
                  ['H', 0., 0.785198 , -0.427268]],
            basis='6-31G',
            verbose=0)

# Do Hartree-Fock calculation
mf = mol.RHF().run()

# Calculate Hessian
hessian = mf.Hessian().kernel()
# Change the array layout
hessian = hessian.transpose(0, 2, 1, 3).reshape(mol.natm * 3, mol.natm * 3)

# Mass-weight the Hessian
atom_masses = mol.atom_mass_list(isotope_avg=True)
atom_masses = np.repeat(atom_masses, 3)
Mhalf = 1/np.sqrt(np.outer(atom_masses, atom_masses))
weighted_hessian = hessian * Mhalf

# Calculate eigenvalues and eigenvectors
force_constants, modes = np.linalg.eigh(weighted_hessian)

# Change units for wavenumbers
frequencies = np.sqrt(np.abs(force_constants))
to_wavenumbers = (nist.HARTREE2J / (nist.ATOMIC_MASS * nist.BOHR_SI**2))**.5
to_wavenumbers *= 1/(2 * np.pi) / nist.LIGHT_SPEED_SI * 1e-2
to_wavenumbers * np.sort(frequencies)[-3:]
```

```
array([1736.71755055, 3988.23212533, 4145.21010618])
```

Calculate the analytical Hessian

- For large molecules: even more expensive

Normal modes in Python


```python
from pyscf import gto
import numpy as np
from pyscf.data import nist

# Build molecule
mol = gto.M(atom=[['O', 0., 0., 0.106817],
                  ['H', 0., -0.785198 , -0.427268],
                  ['H', 0., 0.785198 , -0.427268]],
            basis='6-31G',
            verbose=0)

# Do Hartree-Fock calculation
mf = mol.RHF().run()

# Calculate Hessian
hessian = mf.Hessian().kernel()
# Change the array layout
hessian = hessian.transpose(0, 2, 1, 3).reshape(mol.natm * 3, mol.natm * 3)

# Mass-weight the Hessian
atom_masses = mol.atom_mass_list(isotope_avg=True)
atom_masses = np.repeat(atom_masses, 3)
Mhalf = 1/np.sqrt(np.outer(atom_masses, atom_masses))
weighted_hessian = hessian * Mhalf

# Calculate eigenvalues and eigenvectors
force_constants, modes = np.linalg.eigh(weighted_hessian)

# Change units for wavenumbers
frequencies = np.sqrt(np.abs(force_constants))
to_wavenumbers = (nist.HARTREE2J / (nist.ATOMIC_MASS * nist.BOHR_SI**2))**.5
to_wavenumbers *= 1/(2 * np.pi) / nist.LIGHT_SPEED_SI * 1e-2
to_wavenumbers * np.sort(frequencies)[-3:]
```

```
array([1736.71755055, 3988.23212533, 4145.21010618])
```

Change the memory layout

- PySCF returns the Hessian as four-dimensional array (atom1, atom2, dimension1, dimension2)
- We need it as square symmetric matrix

np.transpose()

- Transposes a matrix = changes axes order
- By default: reverse axes
- Here: Sort into (atom1, dimension1, atom2, dimension2)

np.reshape()

- Keeps data, looks at it differently
- Here: Makes matrix square

Normal modes in Python


```
from pyscf import gto
import numpy as np
from pyscf.data import nist

# Build molecule
mol = gto.M(atom=[['O', 0., 0., 0.106817],
                  ['H', 0., -0.785198 , -0.427268],
                  ['H', 0., 0.785198 , -0.427268]],
            basis='6-31G',
            verbose=0)

# Do Hartree-Fock calculation
mf = mol.RHF().run()

# Calculate Hessian
hessian = mf.Hessian().kernel()
# Change the array layout
hessian = hessian.transpose(0, 2, 1, 3).reshape(mol.natm * 3, mol.natm * 3)

# Mass-weight the Hessian
atom_masses = mol.atom_mass_list(isotope_avg=True)
atom_masses = np.repeat(atom_masses, 3)
Mhalf = 1/np.sqrt(np.outer(atom_masses, atom_masses))
weighted_hessian = hessian * Mhalf

# Calculate eigenvalues and eigenvectors
force_constants, modes = np.linalg.eigh(weighted_hessian)

# Change units for wavenumbers
frequencies = np.sqrt(np.abs(force_constants))
to_wavenumbers = (nist.HARTREE2J / (nist.ATOMIC_MASS * nist.BOHR_SI**2))**.5
to_wavenumbers *= 1/(2 * np.pi) / nist.LIGHT_SPEED_SI * 1e-2
to_wavenumbers * np.sort(frequencies)[-3:]
```

```
array([1736.71755055, 3988.23212533, 4145.21010618])
```

Build list of atomic masses

- PySCF has built-in data sets, no copying required

np.repeat()

- Repeats each element
- Once for each dimension

Normal modes in Python


```python
from pyscf import gto
import numpy as np
from pyscf.data import nist

# Build molecule
mol = gto.M(atom=[['O', 0., 0., 0.106817],
                  ['H', 0., -0.785198 , -0.427268],
                  ['H', 0., 0.785198 , -0.427268]],
            basis='6-31G',
            verbose=0)

# Do Hartree-Fock calculation
mf = mol.RHF().run()

# Calculate Hessian
hessian = mf.Hessian().kernel()
# Change the array layout
hessian = hessian.transpose(0, 2, 1, 3).reshape(mol.natm * 3, mol.natm * 3)

# Mass-weight the Hessian
atom_masses = mol.atom_mass_list(isotope_avg=True)
atom_masses = np.repeat(atom_masses, 3)
Mhalf = 1/np.sqrt(np.outer(atom_masses, atom_masses))
weighted_hessian = hessian * Mhalf

# Calculate eigenvalues and eigenvectors
force_constants, modes = np.linalg.eigh(weighted_hessian)

# Change units for wavenumbers
frequencies = np.sqrt(np.abs(force_constants))
to_wavenumbers = (nist.HARTREE2J / (nist.ATOMIC_MASS * nist.BOHR_SI**2))**.5
to_wavenumbers *= 1/(2 * np.pi) / nist.LIGHT_SPEED_SI * 1e-2
to_wavenumbers * np.sort(frequencies)[-3:]
```

```
array([1736.71755055, 3988.23212533, 4145.21010618])
```

Build mass-weighting matrix

- Note that *np.sqrt()* is operating elementwise
- *matrix* * *matrix* is elementwise (*np.matmul()* would be matrix multiplication)

np.outer()

- Outer product
- Pairwise multiplication: $M_{ij} = m_i m_j$

Normal modes in Python


```
from pyscf import gto
import numpy as np
from pyscf.data import nist

# Build molecule
mol = gto.M(atom=[['O', 0., 0., 0.106817],
                  ['H', 0., -0.785198 , -0.427268],
                  ['H', 0., 0.785198 , -0.427268]],
            basis='6-31G',
            verbose=0)

# Do Hartree-Fock calculation
mf = mol.RHF().run()

# Calculate Hessian
hessian = mf.Hessian().kernel()
# Change the array layout
hessian = hessian.transpose(0, 2, 1, 3).reshape(mol.natm * 3, mol.natm * 3)

# Mass-weight the Hessian
atom_masses = mol.atom_mass_list(isotope_avg=True)
atom_masses = np.repeat(atom_masses, 3)
Mhalf = 1/np.sqrt(np.outer(atom_masses, atom_masses))
weighted_hessian = hessian * Mhalf

# Calculate eigenvalues and eigenvectors
force_constants, modes = np.linalg.eigh(weighted_hessian)

# Change units for wavenumbers
frequencies = np.sqrt(np.abs(force_constants))
to_wavenumbers = (nist.HARTREE2J / (nist.ATOMIC_MASS * nist.BOHR_SI**2))**.5
to_wavenumbers *= 1/(2 * np.pi) / nist.LIGHT_SPEED_SI * 1e-2
to_wavenumbers * np.sort(frequencies)[-3:]
```

```
array([1736.71755055, 3988.23212533, 4145.21010618])
```

Get eigenvalues

- From weighted Hessian(!)

np.eigh()

- For symmetric matrices
- Otherwise: *np.eig()*

Normal modes in Python


```python
from pyscf import gto
import numpy as np
from pyscf.data import nist

# Build molecule
mol = gto.M(atom=[['O', 0., 0., 0.106817],
                  ['H', 0., -0.785198 , -0.427268],
                  ['H', 0., 0.785198 , -0.427268]],
            basis='6-31G',
            verbose=0)

# Do Hartree-Fock calculation
mf = mol.RHF().run()

# Calculate Hessian
hessian = mf.Hessian().kernel()
# Change the array layout
hessian = hessian.transpose(0, 2, 1, 3).reshape(mol.natm * 3, mol.natm * 3)

# Mass-weight the Hessian
atom_masses = mol.atom_mass_list(isotope_avg=True)
atom_masses = np.repeat(atom_masses, 3)
Mhalf = 1/np.sqrt(np.outer(atom_masses, atom_masses))
weighted_hessian = hessian * Mhalf

# Calculate eigenvalues and eigenvectors
force_constants, modes = np.linalg.eigh(weighted_hessian)

# Change units for wavenumbers
frequencies = np.sqrt(np.abs(force_constants))
to_wavenumbers = (nist.HARTREE2J / (nist.ATOMIC_MASS * nist.BOHR_SI**2))**.5
to_wavenumbers *= 1/(2 * np.pi) / nist.LIGHT_SPEED_SI * 1e-2
to_wavenumbers * np.sort(frequencies)[-3:]
```

```
array([1736.71755055, 3988.23212533, 4145.21010618])
```

Get frequencies from force constants

- Harmonic approximation
- Negative and small entries:
  3 translational and 3 rotational degrees of freedom

np.abs()

- Absolute value
- Here: shortcut (better: remove translational and rotational degrees first)

Normal modes in Python


```python
from pyscf import gto
import numpy as np
from pyscf.data import nist

# Build molecule
mol = gto.M(atom=[['O', 0., 0., 0.106817],
                  ['H', 0., -0.785198 , -0.427268],
                  ['H', 0., 0.785198 , -0.427268]],
            basis='6-31G',
            verbose=0)

# Do Hartree-Fock calculation
mf = mol.RHF().run()

# Calculate Hessian
hessian = mf.Hessian().kernel()
# Change the array layout
hessian = hessian.transpose(0, 2, 1, 3).reshape(mol.natm * 3, mol.natm * 3)

# Mass-weight the Hessian
atom_masses = mol.atom_mass_list(isotope_avg=True)
atom_masses = np.repeat(atom_masses, 3)
Mhalf = 1/np.sqrt(np.outer(atom_masses, atom_masses))
weighted_hessian = hessian * Mhalf

# Calculate eigenvalues and eigenvectors
force_constants, modes = np.linalg.eigh(weighted_hessian)

# Change units for wavenumbers
frequencies = np.sqrt(np.abs(force_constants))
to_wavenumbers = (nist.HARTREE2J / (nist.ATOMIC_MASS * nist.BOHR_SI**2))**.5
to_wavenumbers *= 1/(2 * np.pi) / nist.LIGHT_SPEED_SI * 1e-2
to_wavenumbers * np.sort(frequencies)[-3:]
```

```
array([1736.71755055, 3988.23212533, 4145.21010618])
```

Convert to wavenumbers

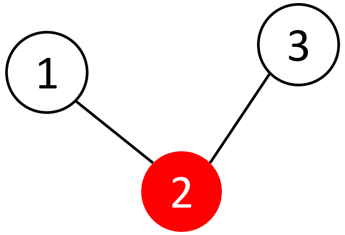
- PySCF has predefined constants

PySCF can do it

```python
from pyscf import gto
from pyscf.hessian.thermo import harmonic_analysis

# Build molecule
mol = gto.M(atom=[['O', 0., 0., 0.106817],
                  ['H', 0., -0.785198 , -0.427268],
                  ['H', 0., 0.785198 , -0.427268]],
            basis='6-31G',
            verbose=0)

# Do Hartree-Fock calculation
mf = mol.RHF().run()

# Calculate Hessian
hessian = mf.Hessian().kernel()

harmonic_analysis(mol, hessian)['freq_wavenumber']
```

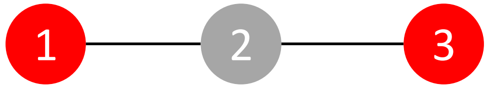
```
array([1736.71755056, 3988.23212533, 4145.21010587])
```

Eigenvalue spectrum



Definition

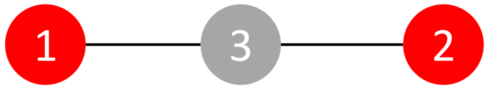
- Unordered set of eigenvalues
- Invariant under permutations



Adjacency matrices

- 1 if there is a bond between atom *i* and *j*
- 0 otherwise
- Graph property, many useful algorithms

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$



Eigenvalues

- $\pm\, 2^{1/2}$, 0
- *np.linalg.eigh()[0]*

Solving systems of linear equations

Core idea
- Collect terms in matrix
- Invert matrix
- Matrix vector product

np.linalg.inv()
- Inverts the matrix

np.dot()
- Matrix vector product

Overdetermined case
- *np.linalg.pinv()*
- Pseudoinverse
- Alternative for least-squares fit

$$\mathbf{A}\mathbf{x} = \mathbf{b} \Rightarrow \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

```
A
```

```
array([[85, 85, 66],
       [42, 27, 24],
       [85, 58, 62]])
```

```
b
```

```
array([7742, 3201, 6871])
```

```
np.dot(np.linalg.inv(A), b)
```

```
array([45., 29., 22.])
```

Determinant

Application

- System unsolvable, i.e. A not invertible: det(A) = 0
- *np.linalg.det()*

$$\mathbf{A}\mathbf{x} = \mathbf{b} \Rightarrow \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

Pitfalls

Numerics

- Matrix inversion unstable
- Machine epsilon introduces inaccuracies
- Hint: do not simplify algorithms you implement

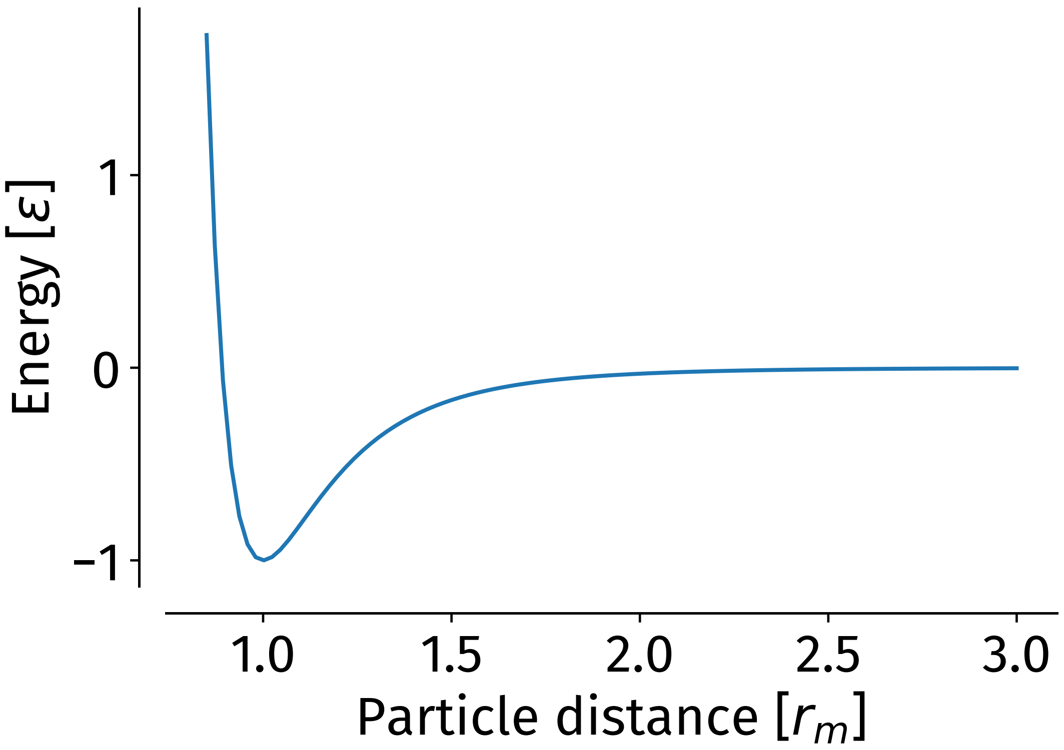
Resources

- Matrices become large quickly
  1GB: N=12k
  10GB: N=36k
- Matrix operations scale worse
  - Matrix multiplication scales as $N^{\sim 2.8}$

Matrix exponentiation

Numerics

- Matrix multiplication scales as $N^{\sim 2.8}$
- Group matrix operations $A^4 = (A^2)^2$
- Works for scalars as well
  - Reason why Lennard-Jones potential is commonly used



Unstable matrix inversion

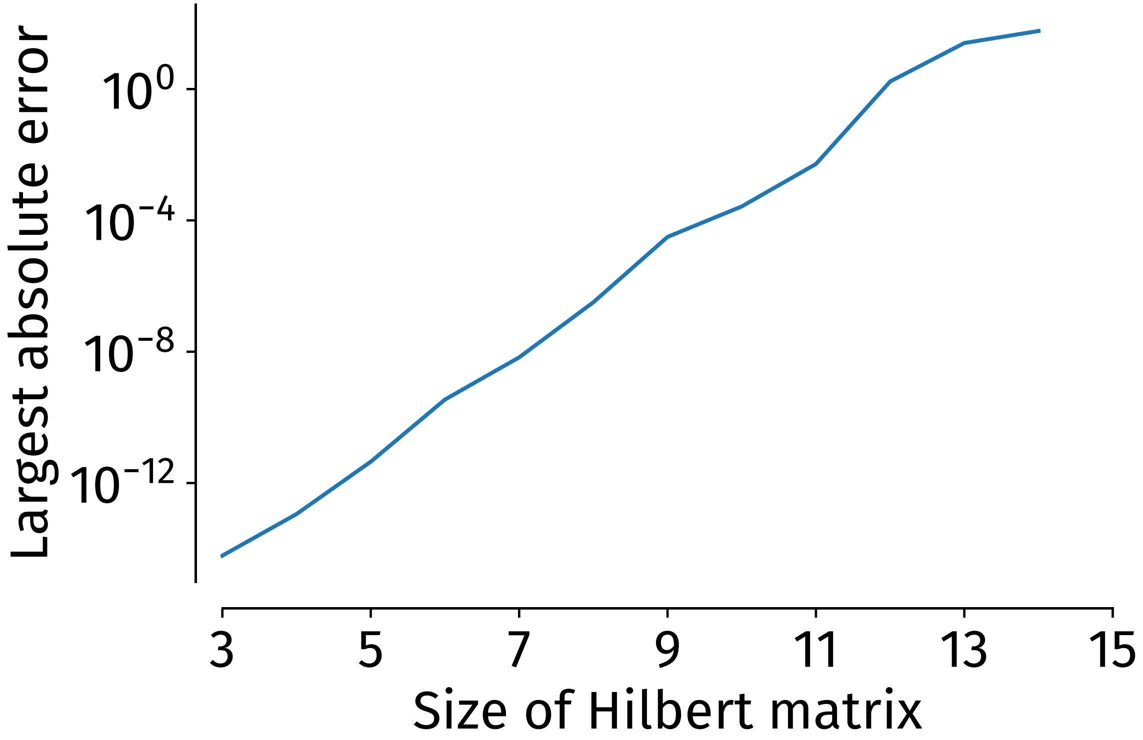
Simple case

- Hilbert matrix
- $H_{ij} = 1/(i+j-1)$

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}$$

```python
def hilbert_matrix(N):
    idx = np.arange(0, N)
    hilbert = 1 / (np.repeat(idx, N) + np.tile(idx, N) + 1)
    return hilbert.reshape(N, N)
def check_inverse(matrix):
    inverse = np.linalg.inv(matrix)
    product = np.matmul(inverse, matrix)
    deviations = product - np.identity(matrix.shape[0])
    return np.max(np.abs(deviations))
check_inverse(hilbert_matrix(3)), check_inverse(hilbert_matrix(15))
```

```
(6.143234069592521e-15, 351.1986284929532)
```



Symmetry operations

Matrices are common tools

- Entries encode geometrical change
- *pymatgen* implements these

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

Identity

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{x} = \mathbf{I}\mathbf{x}$$

Reflection in the xy-plane

$$\mathbf{R}_{xy} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \qquad \mathbf{x}' = \mathbf{R}_{xy}\mathbf{x}$$

Symmetry operations

Inversion

$$\mathbf{T} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \qquad \mathbf{T}^2 = \mathbf{I}$$

Rotation around x

$$\mathbf{R}_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \qquad C_n : \alpha = \frac{2\pi}{n}$$

General tip

Try to make any problem you face either

- A matrix
- A graph
- An optimization problem

Advantages

- Huge literature on either topic
- Efficient algorithms
- Reliable libraries with interfaces to Python

Libraries in Python

- Matrix: *numpy* / *scipy*
- Graph: *NetworkX* (easy but slow, good visualisation)
- Graph: *igraph* (fast, but quite technical interface)
- Optimisation: *scipy* (easy interface)
- Optimisation: *DEAP* (global optimization, quite technical interface)

Summary

Operations
- Eigenvalues / Eigenvectors
- Matrix multiplications
- Matrix inversions
- Solving systems of linear equations

Caveats
- Numerical stability
- Memory requirements

Python
- *pymatgen* for symmetry operations

ferchault @ferchault guido.vonrudorff.de