# Procrustes Documentation

***Release 0.0.1-alpha***

**The QC-Devs Community**

**Feb 26, 2022**

# USER DOCUMENTATION

Procrustes is a free, open-source, and cross-platform Python library for (generalized) Procrustes problems with the goal of finding the optimal transformation(s) that makes two matrices as close as possible to each other. Please use the following citation in any publication using Procrustes library:

> **"Procrustes: A Python Library to Find Transformations that Maximize the Similarity Between Matrices"**, F. Meng, M. Richer, A. Tehrani, J. La, T. D. Kim, P. W. Ayers, F. Heidar-Zadeh, JOURNAL 2021; ISSUE PAGE NUMBER.

The Procrustes source code is hosted on GitHub and is released under the GNU General Public License v3.0. We welcome any contributions to the Procrustes library in accordance with our Code of Conduct; please see our Contributing Guidelines. Please report any issues you encounter while using Procrustes library on GitHub Issues. For further information and inquiries please contact us at qcdevs@gmail.com.

CHAPTER

# ONE

# DESCRIPTION OF PROCRUSTES METHODS

Procrustes problems arise when one wishes to find one or two transformations, $\mathbf{T} \in \mathbb{R}^{n \times n}$ and $\mathbf{S} \in \mathbb{R}^{m \times m}$, that make matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ (input matrix) resemble matrix $\mathbf{B} \in \mathbb{R}^{m \times n}$ (target or reference matrix) as closely as possible:

$$\underbrace{\min}_{\mathbf{S},\mathbf{T}} \|\mathbf{SAT} - \mathbf{B}\|_F^2$$

where, the $\|\cdot\|_F$ denotes the Frobenius norm defined as,

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2} = \sqrt{\mathrm{Tr}(\mathbf{A}^\dagger\mathbf{A})}$$

Here $a_{ij}$ and $\mathrm{Tr}(\mathbf{A})$ denote the $ij$-th element and trace of matrix $\mathbf{A}$, respectively. When $\mathbf{S}$ is an identity matrix, this is called a **one-sided Procrustes problem**, and when it is equal to $\mathbf{T}$, this becomes **two-sided Procrustes problem with one transformation**, otherwise, it is called **two-sided Procrustes problem**. Different Procrustes problems use different choices for the transformation matrices $\mathbf{S}$ and $\mathbf{T}$ which are commonly taken to be orthogonal/unitary matrices, rotation matrices, symmetric matrices, or permutation matrices. The table below summarizes various Procrustes methods supported:

| **Procrustes Type** | $\mathbf{S}$ | $\mathbf{T}$ | Constraints |
|---|---|---|---|
| *Generic* [1] | $\mathbf{I}$ | $\mathbf{T}$ | None |
| *Orthogonal* [2][3][4] | $\mathbf{I}$ | $\mathbf{Q}$ | $\mathbf{Q}^{-1} = \mathbf{Q}^\dagger$ |
| *Rotational* [4][5][6] | $\mathbf{I}$ | $\mathbf{R}$ | $\begin{cases}\mathbf{R}^{-1} = \mathbf{R}^\dagger \\ \lvert\mathbf{R}\rvert = 1\end{cases}$ |
| *Symmetric* [7][8][9] | $\mathbf{I}$ | $\mathbf{X}$ | $\mathbf{X} = \mathbf{X}^\dagger$ |
| *Permutation* [10] | $\mathbf{I}$ | $\mathbf{P}$ | $\begin{cases}[\mathbf{P}]_{ij} \in \{0,1\} \\ \sum_{i=1}^{n}[\mathbf{P}]_{ij} = \sum_{j=1}^{n}[\mathbf{P}]_{ij} = 1\end{cases}$ |
| *Two-sided Orthogonal* [11] | $\mathbf{Q}_1^\dagger$ | $\mathbf{Q}_2$ | $\begin{cases}\mathbf{Q}_1^{-1} = \mathbf{Q}_1^\dagger \\ \mathbf{Q}_2^{-1} = \mathbf{Q}_2^\dagger\end{cases}$ |
| *Two-sided Orthogonal with One Transformation* [12] | $\mathbf{Q}^\dagger$ | $\mathbf{Q}$ | $\mathbf{Q}^{-1} = \mathbf{Q}^\dagger$ |
| *Two-sided Permutation* [13] | $\mathbf{P}_1^\dagger$ | $\mathbf{P}_2$ | $\begin{cases}[\mathbf{P}_1]_{ij} \in \{0,1\} \\ [\mathbf{P}_2]_{ij} \in \{0,1\} \\ \sum_{i=1}^{n}[\mathbf{P}_1]_{ij} = \sum_{j=1}^{n}[\mathbf{P}_1]_{ij} = 1 \\ \sum_{i=1}^{n}[\mathbf{P}_2]_{ij} = \sum_{j=1}^{n}[\mathbf{P}_2]_{ij} = 1\end{cases}$ |
| *Two-sided Permutation with One Transformation* [2][14] | $\mathbf{P}^\dagger$ | $\mathbf{P}$ | $\begin{cases}[\mathbf{P}]_{ij} \in \{0,1\} \\ \sum_{i=1}^{n}[\mathbf{P}]_{ij} = \sum_{j=1}^{n}[\mathbf{P}]_{ij} = 1\end{cases}$ |

In addition to these Procrustes methods, summarized in the table above, the *generalized Procrustes analysis (GPA)* [15][1][16][17][18] and softassign algorithm [19][20][21] are also implemented in our package. The GPA algorithm

seeks the optimal transformation matrices $\mathbf{T}$ to superpose the given objects (usually more than 2) with minimum distance,

$$\min \sum_{i<j}^{j} \|\mathbf{A}_i\mathbf{T}_i - \mathbf{A}_j\mathbf{T}_j\|_F{}^2 (1.1)$$

where $\mathbf{A}_i$ and $\mathbf{A}_j$ are the configurations and $\mathbf{T}_i$ and $\mathbf{T}_j$ denotes the transformation matrices for $\mathbf{A}_i$ and $\mathbf{A}_j$ respectively. When only two objects are given, the problem shrinks to generic Procrustes.

The *softassign* algorithm was first proposed to deal with quadratic assignment problem [19] inspired by statistical physics algorithms and has subsequently been developed theoretically [20][21] and extended to many other applications [22][20][23][24][25]. Because the two-sided permutation Procrustes problem is a special case of the quadratic assignment problem, it can be used here. The objective function is to minimize $E_{qap}(\mathbf{M}, \mu, \nu)$, [20][26], which is defined as follows,

$$\begin{aligned} E_{qap}(\mathbf{M}, \mu, \nu) = &-\frac{1}{2}\sum_{aibj} \mathbf{C}_{ai;bj}\mathbf{M}_{ai}\mathbf{M}_{bj} \\ &+ \sum_a \mu_a \left(\sum_i \mathbf{M}_{ai} - 1\right) + \sum_i \nu_i \left(\sum_a \mathbf{M}_{ai} - 1\right) \\ &- \frac{\gamma}{2}\sum_{ai} \mathbf{M}_{ai}^2 + \frac{1}{\beta}\sum_{ai} \mathbf{M}_{ai}\log \mathbf{M}_{ai} \end{aligned}$$

Procrustes problems arise when aligning molecules and other objects, when evaluating optimal basis transformations, when determining optimal mappings between sets, and in many other contexts. This package includes the options to translate, scale, and zero-pad matrices, so that matrices with different centers/scaling/sizes can be considered.

## 1.1 Installation

### 1.1.1 Downloading Code

The latest code can be obtained through theochem (https://github.com/theochem/procrustes) in Github,

```
git clone git@github.com:theochem/procrustes.git
```

### 1.1.2 Dependencies

The following dependencies will be necessary for Procrustes to build properly,

- Python >= 3.6: https://www.python.org/
- SciPy >= 1.5.0: https://www.scipy.org/
- NumPy >= 1.18.5: https://www.numpy.org/
- Pip >= 19.0: https://pip.pypa.io/
- PyTest >= 5.4.3: https://docs.pytest.org/
- PyTest-Cov >= 2.8.0: https://pypi.org/project/pytest-cov/
- Sphinx >= 2.3.0, if one wishes to build the documentation locally: https://www.sphinx-doc.org/

### 1.1.3 Installation

The stable release of the package can be easily installed through the *pip* and *conda* package management systems, which install the dependencies automatically, if not available. To use *pip*, simply run the following command:

```
pip install qc-procrustes
```

To use *conda*, one can either install the package through Anaconda Navigator or run the following command in a desired *conda* environment:

```
conda install -c theochem procrustes
```

Alternatively, the *Procrustes* source code can be download from GitHub (either the stable version or the development version) and then installed from source. For example, one can download the latest source code using *git* by:

```
# download source code
git clone git@github.com:theochem/procrustes.git
cd procrustes
```

From the parent directory, the dependencies can either be installed using *pip* by:

```
# install dependencies using pip
pip install -r requirements.txt
```

or, through *conda* by:

```
# create and activate myenv environment
# Procruste works with Python 3.6, 3.7, and 3.8
conda create -n myenv python=3.6
conda activate myenv
# install dependencies using conda
conda install --yes --file requirements.txt
```

Finally, the *Procrustes* package can be installed (from source) by:

```
# install Procrustes from source
pip install .
```

### 1.1.4 Testing

To make sure that the package is installed properly, the *Procrustes* tests should be executed using *pytest* from the parent directory:

```
# testing without coverage report
pytest -v .
```

In addition, to generate a coverage report alongside testing, one can use:

```
# testing with coverage report
pytest --cov-config=.coveragerc --cov=procrustes procrustes/test
```

## 1.2 Quick Start

**Orthogonal Procrustes**

Given matrix $\mathbf{A}_{m\times n}$ and a reference matrix $\mathbf{B}_{m\times n}$, find the orthogonal transformation matrix $\mathbf{Q}_{n\times n}$ that makes $\mathbf{AQ}$ as close as possible to $\mathbf{B}$, i.e.,

$$\underbrace{\min}_{\{\mathbf{Q}|\mathbf{Q}^{-1}=\mathbf{Q}^{\dagger}\}} \|\mathbf{AQ}-\mathbf{B}\|_F^2 \tag{1.2}$$

The code block below gives an example of the orthogonal Procrustes problem for random matrices $\mathbf{A}$ and $\mathbf{B}$. Here, matrix $\mathbf{B}$ is constructed by shifting an orthogonal transformation of matrix $\mathbf{A}$, so the matrices can be perfectly matched. As is the case with all Procrustes flavours, the user can specify whether the matrices should be translated (so that both are centered at origin) and/or scaled (so that both are normalized to unity with respect to the Frobenius norm). In addition, the other optional arguments (not appearing in the code-block below) specify whether the zero columns (on the right-hand side) and rows (at the bottom) should be removed prior to transformation.

[1]:
```
import numpy as np
from scipy.stats import ortho_group
from procrustes import orthogonal

# random input 10x7 matrix A
a = np.random.rand(10, 7)

# random orthogonal 7x7 matrix T (acting as Q)
t = ortho_group.rvs(7)

# target matrix B (which is a shifted AT)
b = np.dot(a, t) + np.random.rand(1, 7)

# orthogonal Procrustes analysis with translation
result = orthogonal(a, b, scale=True, translate=True)

# compute transformed matrix A (i.e., A x Q)
aq = np.dot(result.new_a, result.t)

# display Procrustes results
print("Procrustes Error = ", result.error)
print("\nDoes the obtained transformation match variable t? ", np.allclose(t, result.t))
print("Does AQ and B matrices match?", np.allclose(aq, result.new_b))

print("Transformation Matrix T = ")
print(result.t)
print("")
print("Matrix A (after translation and scaling) = ")
print(result.new_a)
print("")
print("Matrix AQ = ")
print(aq)
print("")
print("Matrix B (after translation and scaling) = ")
print(result.new_b)
```

```
Procrustes Error =  2.4289236009459004e-30

Does the obtained transformation match variable t?  True
Does AQ and B matrices match? True
Transformation Matrix T =
[[-0.05485823 -0.21987681  0.71054591 -0.25910115  0.54815253 -0.11057903
  -0.25285756]
 [-0.52504088  0.57359009  0.08125893 -0.38628239 -0.16691215  0.36377965
  -0.28162755]
 [-0.13427477 -0.35050672  0.5114134  -0.07667139 -0.71683955  0.04718091
   0.27496942]
 [ 0.19650197  0.65967521  0.42739622  0.33295066  0.08143714 -0.10148903
   0.46449961]
 [ 0.45237142 -0.06057057 -0.04420777 -0.48171449  0.15466827  0.61863151
   0.38866554]
 [-0.0206257   0.12956287 -0.16552502 -0.64649468 -0.03687109 -0.66740601
   0.30107121]
 [ 0.67794881  0.21015927  0.12230059 -0.13005245 -0.35481889 -0.12155183
  -0.56892541]]

Matrix A (after translation and scaling) =
[[-0.00055723 -0.01377505 -0.08991782 -0.16477186  0.14750515  0.00704046
  -0.20861997]
 [ 0.13709756 -0.02863268 -0.04751636  0.07709446 -0.14385904  0.10799371
  -0.0032753 ]
 [-0.16853924 -0.17772675  0.06648402 -0.16374724  0.04464136  0.09435058
   0.03478612]
 [ 0.11811112 -0.01507476 -0.06673255  0.139217   -0.0976698  -0.11659321
  -0.03462358]
 [-0.15903118 -0.01484409 -0.16695146  0.11677978 -0.19205803  0.01023694
   0.10550636]
 [ 0.02382673 -0.01263136  0.05182648 -0.10599329 -0.08805356 -0.07338563
   0.00990156]
 [-0.16720248 -0.04508176  0.03784015  0.10676988  0.04590812  0.00066062
  -0.03174471]
 [-0.06525965  0.14948796  0.07651695  0.02690549 -0.13166814 -0.24761947
  -0.19722096]
 [ 0.12908648  0.16851992 -0.01048313 -0.21501857  0.21002468  0.11819247
   0.14823474]
 [ 0.15246788 -0.01024143  0.14893372  0.18276434  0.20522923  0.09912353
   0.17705575]]

Matrix AQ =
[[-0.08789303 -0.13682353 -0.15112392 -0.09097679  0.14960896  0.1194413
   0.08089827]
 [-0.04048379  0.04296137  0.09182028  0.00475755  0.09519934 -0.19631547
  -0.02539345]
 [ 0.10328743 -0.17937652 -0.1835174  -0.03432131 -0.13263092 -0.06584305
   0.06085581]
 [-0.02749881  0.06414488  0.12745368  0.15361799  0.12791064 -0.01422005
  -0.03266849]
 [ 0.04631795  0.19713971 -0.12997567  0.17079912 -0.02302644 -0.14601308
  -0.07885918]
```

(continues on next page)

(continued from previous page)

```
 [-0.05406852 -0.10266454  0.01435798  0.04801396 -0.04504059 -0.00072564
  -0.09940151]
 [ 0.04797432  0.05870918 -0.06350457  0.07497127 -0.08421826  0.02485656
   0.1510763 ]
 [-0.26805587  0.02546819  0.03909608  0.21141601 -0.05464003  0.17025779
  -0.00558307]
 [ 0.05666167 -0.03614484 -0.00256305 -0.36619058  0.00816506  0.1013846
  -0.14997974]
 [ 0.22375865  0.06658609  0.25795658 -0.17208723 -0.04132777  0.00717706
   0.09905505]]

Matrix B (after translation and scaling) =
[[-0.08789303 -0.13682353 -0.15112392 -0.09097679  0.14960896  0.1194413
   0.08089827]
 [-0.04048379  0.04296137  0.09182028  0.00475755  0.09519934 -0.19631547
  -0.02539345]
 [ 0.10328743 -0.17937652 -0.1835174  -0.03432131 -0.13263092 -0.06584305
   0.06085581]
 [-0.02749881  0.06414488  0.12745368  0.15361799  0.12791064 -0.01422005
  -0.03266849]
 [ 0.04631795  0.19713971 -0.12997567  0.17079912 -0.02302644 -0.14601308
  -0.07885918]
 [-0.05406852 -0.10266454  0.01435798  0.04801396 -0.04504059 -0.00072564
  -0.09940151]
 [ 0.04797432  0.05870918 -0.06350457  0.07497127 -0.08421826  0.02485656
   0.1510763 ]
 [-0.26805587  0.02546819  0.03909608  0.21141601 -0.05464003  0.17025779
  -0.00558307]
 [ 0.05666167 -0.03614484 -0.00256305 -0.36619058  0.00816506  0.1013846
  -0.14997974]
 [ 0.22375865  0.06658609  0.25795658 -0.17208723 -0.04132777  0.00717706
   0.09905505]]
```

---

The corresponding file can be obtained from:

- Jupyter Notebook: `Quick_Start.ipynb`
- Interactive Jupyter Notebook:

---

## 1.3 Tutorials

In order to show how to use Procrustes, we have implemented some practical examples.

### 1.3.1 Chemical Structure Alignment

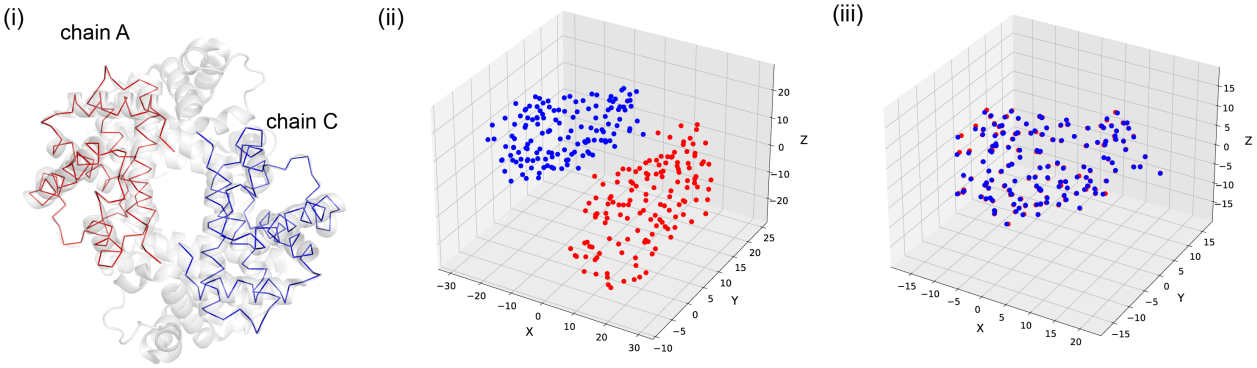
Molecular alignment is a fundamental problem in cheminformatics and can be used for structure determination, similarity based searching, and ligand-based drug design. This problem can be formulated as an orthogonal Procrustes problem where the two matrices represent three-dimensional Cartesian coordinates of molecules.

**Orthogonal Procrustes**

Given matrix $\mathbf{A}_{m\times n}$ and a reference $\mathbf{B}_{m\times n}$, find the find the orthogonal transformation matrix $\mathbf{Q}_{n\times n}$ that makes $\mathbf{A}$ as close as possible to $\mathbf{B}$, i.e.,

$$\underbrace{\min}_{\{\mathbf{Q}|\mathbf{Q}^{-1}=\mathbf{Q}^{\dagger}\}} \|\mathbf{AQ}-\mathbf{B}\|_F^2 \tag{1.3}$$

In the code block below, we use the `procrustes` library for protein structure alignment. We use the 2HHB, which has cyclic-$C_2$ global symmetry, and load its PDB file using IOData library to obtain the 3D-Cartesian coordinates atoms. In 2HHB, the A and C (or B and D) are hemoglobin deoxy-alpha (beta) chains shown in **Fig. (i)**, and their $C_\alpha$ atoms in **Fig. (ii)** are aligned in **Fig. (iii)** to show that they are homologous. The results in **Fig. (iii)** are obtained with orthogonal Procrustes which implements the *Kabsch* algorithm. The root-mean-square deviation (RMSD) is used to assess the discrepancy between structures before and after the translation-rotation transformation.




human deoxyhemoglobin
(PDB ID: 2HHB)

before alignment
RMSD = 39.47 Å

after alignment
RMSD = 0.23 Å

[1]:
```
# chemical structure alignment with orthogonal Procrustes

import numpy as np

from iodata import load_one
from iodata.utils import angstrom
from procrustes import rotational

# load PDB
pdb = load_one("notebook_data/chemical_strcuture_alignment/2hhb.pdb")

# get coordinates of C_alpha atoms in chains A & C (in angstrom)
chainid = pdb.extra['chainids']
attypes = pdb.atffparams['attypes']
```

(continues on next page)

(continued from previous page)

```
# alpha carbon atom coordinates in chain A
ca_a = pdb.atcoords[(chainid == 'A') & (attypes == 'CA')] / angstrom
# alpha carbon atom coordinates in chain A
ca_c = pdb.atcoords[(chainid == 'C') & (attypes == 'CA')] / angstrom

# compute root-mean-square deviation of original chains A and C
rmsd_before = np.sqrt(np.mean(np.sum((ca_a - ca_c)**2, axis=1)))
print("RMSD of initial coordinates:", rmsd_before)

# rotational Procrustes analysis
result = rotational(ca_a, ca_c, translate=True)

# compute transformed (translated & rotated) coordinates of chain A
ca_at = np.dot(result.new_a, result.t)

# compute root-mean-square deviation of trainsformed chains A and C
rmsd_after = np.sqrt(np.mean(np.sum((ca_at - result.new_b)**2, axis=1)))
print("RMSD of transformed coordinates:", rmsd_after)
```

```
RMSD of initial coordinates: 39.46851987559469
RMSD of transformed coordinates: 0.23003870483785005
```

### Plot Procrustes Results

[2]:
```
# Plot outputs of Procrustes

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(12, 10))

# =============
# First subplot
# =============
# set up the axes for the first plot
ax = fig.add_subplot(1, 2, 1, projection='3d')

# coordinates of chains A & C (before alignment)
coords1, coords2 = ca_a, ca_c
title = "Original Chains: RMSD={:0.2f} $\AA$".format(rmsd_before)

ax.scatter(xs=coords1[:, 0], ys=coords1[:, 1], zs=coords1[:, 2],
           marker="o", color="blue", s=55, label="Chain A")
ax.scatter(xs=coords2[:, 0], ys=coords2[:, 1], zs=coords2[:, 2],
           marker="o", color="red", s=55, label="Chain C")

ax.set_xlabel("X", fontsize=14)
ax.set_ylabel("Y", fontsize=14)
ax.set_zlabel("Z", fontsize=14)
ax.legend(fontsize=12, loc="best")
plt.title(title, fontsize=14)
```

(continues on next page)

(continued from previous page)

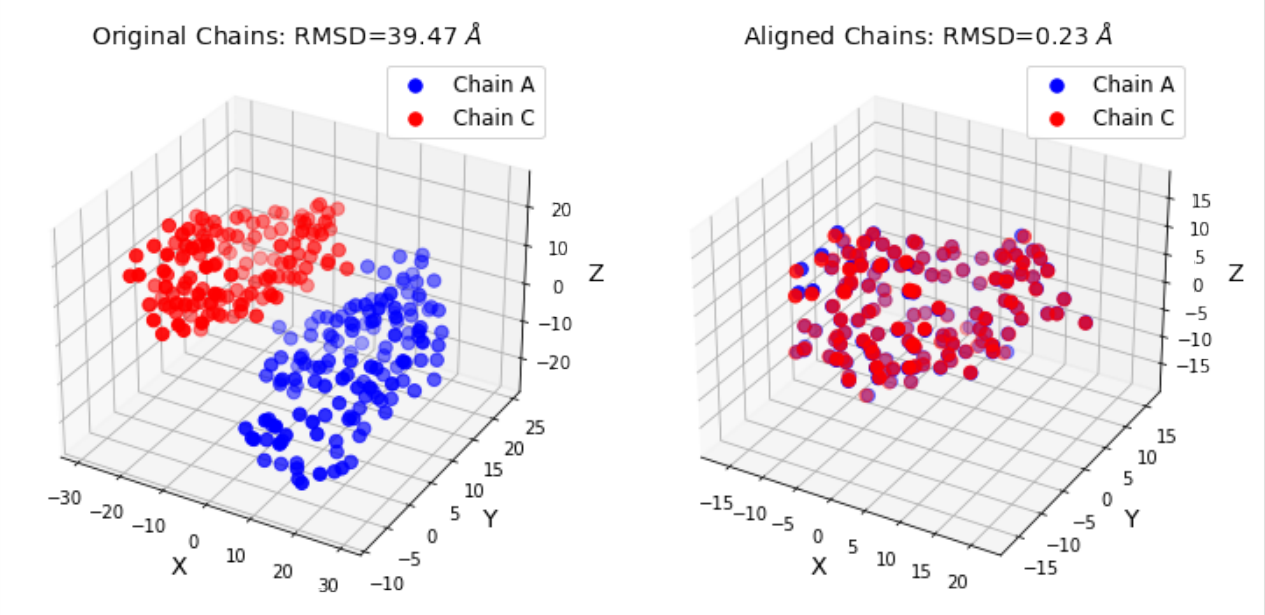
```
# ==============
# Second subplot
# ==============
# set up the axes for the second plot
ax = fig.add_subplot(1, 2, 2, projection='3d')

# coordinates of chains A & C after translation and rotation
coords1, coords2 = ca_at, result.new_b
title="Aligned Chains: RMSD={:0.2f} $\AA$".format(rmsd_after)

ax.scatter(xs=coords1[:, 0], ys=coords1[:, 1], zs=coords1[:, 2],
           marker="o", color="blue", s=55, label="Chain A")
ax.scatter(xs=coords2[:, 0], ys=coords2[:, 1], zs=coords2[:, 2],
           marker="o", color="red", s=55, label="Chain C")

ax.set_xlabel("X", fontsize=14)
ax.set_ylabel("Y", fontsize=14)
ax.set_zlabel("Z", fontsize=14)
ax.legend(fontsize=12, loc="best")
plt.title(title, fontsize=14)

plt.show()
```




The corresponding file can be obtained from:

- Jupyter Notebook: `Chemical_Structure_Alignment.ipynb`
- Interactive Jupyter Notebook:

### 1.3.2 Chirality Check

In chemistry, a molecule is chiral if it cannot be superimposed onto its mirror image by any combination of translation and rotation. These non-superposable mirror images are called enantiomers which share identical chemical and physical properties, but have distinct chemical reactivity and optical rotation properties. Checking whether two structures are enantiomers can be formulated as a Procrustes problem.

**Rotational Procrustes**

Given matrix $\mathbf{A}_{m\times n}$ and a reference $\mathbf{B}_{m\times n}$, find the rotational transformation matrix $\mathbf{R}_{n\times n}$ that makes $\mathbf{A}$ as close as possible to $\mathbf{B}$, i.e.,

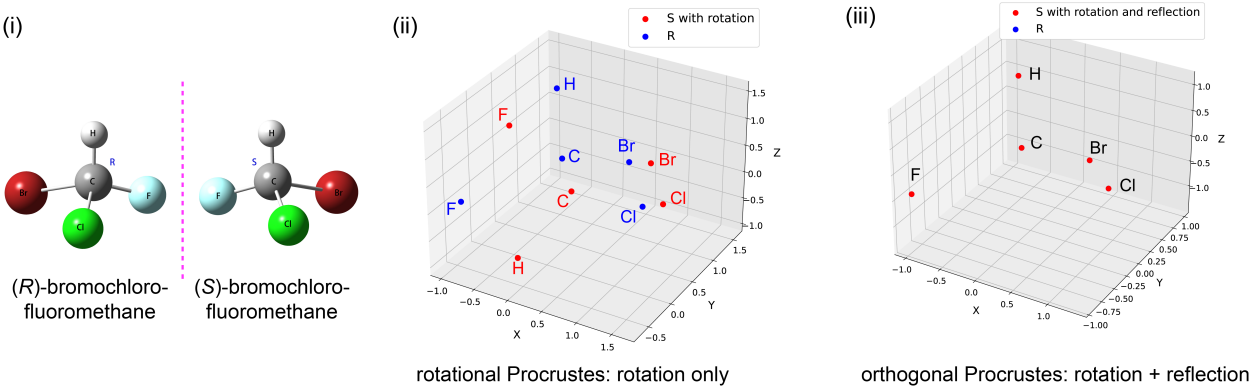
$$\underbrace{\min}_{\left\{\mathbf{R}\middle|\substack{\mathbf{R}^{-1}=\mathbf{R}^{\dagger}\\|\mathbf{R}|=1}\right\}} \|\mathbf{AR}-\mathbf{B}\|_F^2 \tag{1.4}$$

**Orthogonal Procrustes**

Given matrix $\mathbf{A}_{m\times n}$ and a reference $\mathbf{B}_{m\times n}$, find the find the orthogonal transformation matrix $\mathbf{Q}_{n\times n}$ that makes $\mathbf{A}$ as close as possible to $\mathbf{B}$, i.e.,

$$\underbrace{\min}_{\{\mathbf{Q}|\mathbf{Q}^{-1}=\mathbf{Q}^{\dagger}\}} \|\mathbf{AQ}-\mathbf{B}\|_F^2 \tag{1.5}$$

In the code block below, we use the `procrustes` library to check whether two geometries of the CHFClBr molecule are enantiomers; see **Fig. (i)**. The 3D-Cartesian coordinates of molecules are loaded from their XYZ files using IOData library. Testing whether the coordinates can be matched through translation and rotation (i.e., rotational Procrustes) reveals that these two structures are not identical; see **Fig (ii)**. However, the two coordinates are enantiomers because they can be matched through translation, rotation, and reflection (i.e., orthogonal Procrustes) as shown in **Fig (iii)**.




```
[1]: # chirality check with rotational and orthogonal Procrustes

     import numpy as np

     from iodata import load_one
     from procrustes import orthogonal, rotational

     # load CHClFBr enantiomers' coordinates from XYZ files
     a = load_one("notebook_data/chirality_checking/enantiomer1.xyz").atcoords
     b = load_one("notebook_data/chirality_checking/enantiomer2.xyz").atcoords

     # rotational Procrustes on a & b coordinates
     result_rot = rotational(a, b, translate=True, scale=False)
     print("Rotational Procrustes Error = ", result_rot.error)     # output: 26.085545
```

(continues on next page)

(continued from previous page)

```
# orthogonal Procrustes on a & b coordinates
result_ortho = orthogonal(a, b, translate=True, scale=False)
print("Orthogonal Procrustes Error = ", result_ortho.error)   # output: 4.432878e-08
```

```
Rotational Procrustes Error =  26.08554575402178
Orthogonal Procrustes Error =  4.432878638510348e-08
```

### Plot Procrustes Results

[2]:
```
# Plot outputs of Procrustes

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(12, 10))

# =============
# First subplot
# =============
# set up the axes for the first plot
ax = fig.add_subplot(1, 2, 1, projection='3d')

# coordinates of rotated molecule A and molecule B
a_rot = np.dot(result_rot.new_a, result_rot.t)
coords1, coords2 = a_rot, result_rot.new_b
title = "Rotational Procrustes Error={:0.2f} $\AA$".format(result_rot.error)

ax.scatter(xs=coords1[:, 0], ys=coords1[:, 1], zs=coords1[:, 2],
           marker="o", color="blue", s=55, label="enantiomer1 with rotation")
ax.scatter(xs=coords2[:, 0], ys=coords2[:, 1], zs=coords2[:, 2],
           marker="o", color="red", s=55, label="enantiomer2")

ax.set_xlabel("X", fontsize=14)
ax.set_ylabel("Y", fontsize=14)
ax.set_zlabel("Z", fontsize=14)
ax.legend(fontsize=12, loc="best")
plt.title(title, fontsize=14)


# ==============
# Second subplot
# ==============
# set up the axes for the second plot
ax = fig.add_subplot(1, 2, 2, projection='3d')

# coordinates of rotated-and-refelcted molecule A and molecule B
a_rot = np.dot(result_ortho.new_a, result_ortho.t)
coords1, coords2 = a_rot, result_rot.new_b
title="Orthogonal Procrustes Error={:0.2f} $\AA$".format(result_ortho.error)

ax.scatter(xs=coords1[:, 0], ys=coords1[:, 1], zs=coords1[:, 2],
```

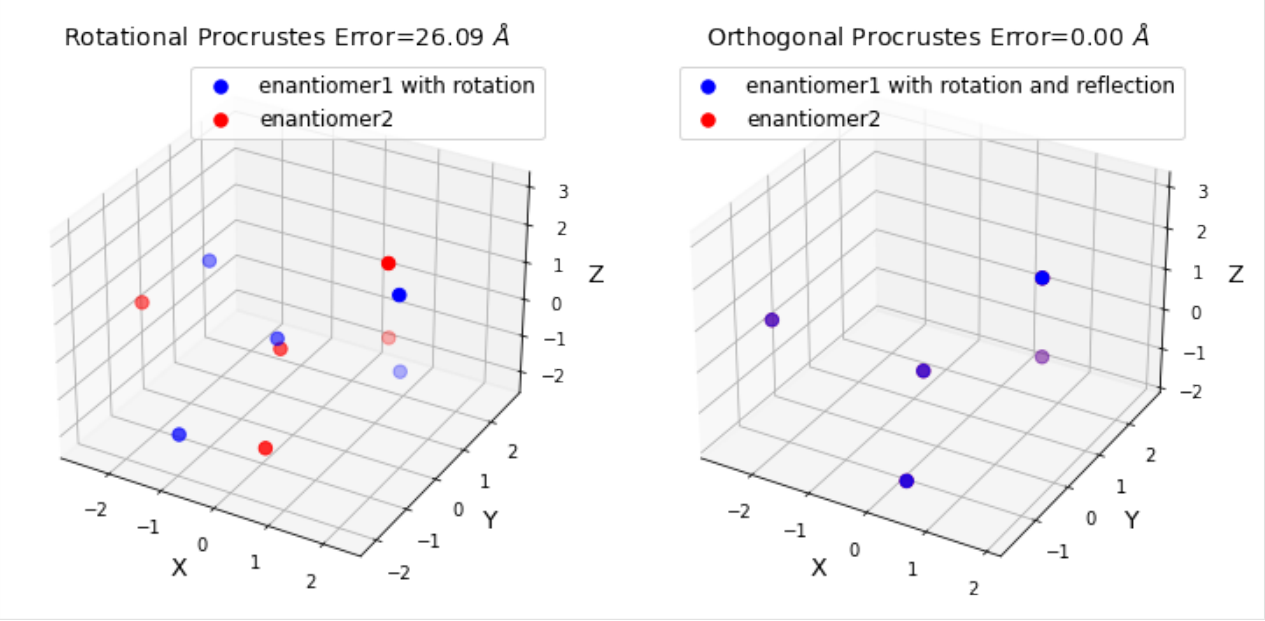
(continues on next page)

(continued from previous page)

```
           marker="o", color="blue", s=55, label="enantiomer1 with rotation and␣
↪reflection")
ax.scatter(xs=coords2[:, 0], ys=coords2[:, 1], zs=coords2[:, 2],
           marker="o", color="red", s=55, label="enantiomer2")

ax.set_xlabel("X", fontsize=14)
ax.set_ylabel("Y", fontsize=14)
ax.set_zlabel("Z", fontsize=14)
ax.legend(fontsize=12, loc="best")
plt.title(title, fontsize=14)

plt.show()
```




The corresponding file can be obtained from:

- Jupyter Notebook: `Chirality_Check.ipynb`
- Interactive Jupyter Notebook:

### 1.3.3 Atom-Atom Mapping

Given two molecular structures, it is important to identify atoms that are chemically similar. This a commonly used in 3D-QSAR pharmacore analysis, substructure searching, metabolic pathway identification, and chemical machine learning. This problem can be formulated as a 2-sided permutation Procrustes with single transformation.

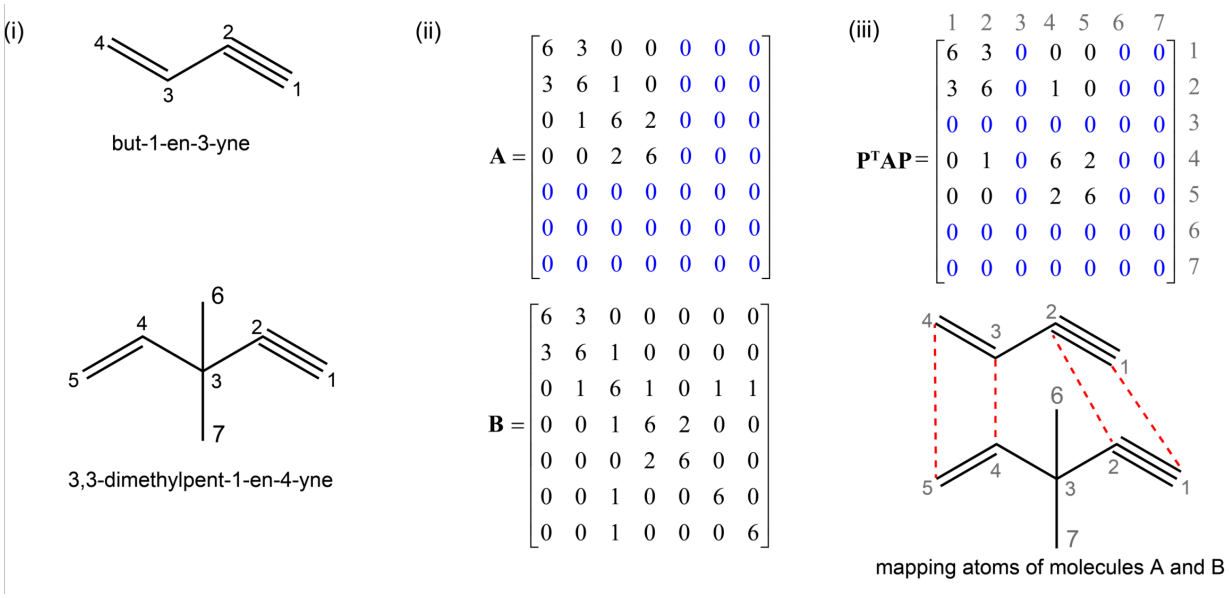
**Permutation Procrustes 2-Sided with Single-Transformation**

Given matrix $\mathbf{A}_{n\times n}$ and a reference $\mathbf{B}_{n\times n}$, find a permutation of rows/columns of $\mathbf{A}_{n\times n}$ that makes it as close as possible to $\mathbf{B}_{n\times n}$, i.e.,

$$\underbrace{\min}_{\left\{\mathbf{P}\,\middle|\,\substack{p_{ij}\in\{0,1\}\\ \sum_{i=1}^{n} p_{ij}=\sum_{j=1}^{n} p_{ij}=1}\right\}} \|\mathbf{P}^{\dagger}\mathbf{A}\mathbf{P}-\mathbf{B}\|_F^2 \tag{1.6}$$

In the code block below, we use the `procrustes` library to map atoms of *but-1-en-3-yne* (molecule **A**) and *3,3-dimethylpent-1-en-4-yne* (molecule **B**) in **Fig. (i)**. Based on our chemical intuition, we can tell that the triple and double bonds of the molecules "match"; however, simple (geometric) molecular alignment based on three-dimensional coordinates does not identify that. The key step is defining a representation that contains bonding information before applying permutation Procrustes to match atoms.

- **Fig. (ii):** Inspired by graph theory, we represent each molecule with an "adjacency" matrix where the diagonal elements are the atomic numbers and the off-diagonal elements are the bond orders. This results in matrices $\mathbf{A} \in \mathbb{R}^{4\times4}$ and $\mathbf{B} \in \mathbb{R}^{7\times7}$. Note that the permutation Procrustes requires the two matrices to be of the same size, so the smaller matrix $\mathbf{A}$ is padded with zero rows and columns to have same shape as matrix $\mathbf{B}$.
- **Fig. (iii):** The mapping between atoms can be also directly deduced from the optimal permutation matrix $\mathbf{P}$. Specifically, the transformed matrix $\mathbf{P}^\top\mathbf{AP}$ should be compared to matrix $\mathbf{B}$ to identify the matching atoms; the zero rows/columns in $\mathbf{A}$ (colored in blue) correspond to atoms in $\mathbf{B}$ for which there are no corresponding atoms.




[1]:
```
# atom-atom mapping with 2-sided permutation procrustes (with single transformation)

import numpy as np
from procrustes import permutation_2sided

# Define molecule A representing but-1-en-3-yne
A = np.array([[6, 3, 0, 0],
              [3, 6, 1, 0],
              [0, 1, 6, 2],
              [0, 0, 2, 6]])

# Define molecule B representing 3,3-dimethylpent-1-en-4-yne
B = np.array([[6, 3, 0, 0, 0, 0, 0],
              [3, 6, 1, 0, 0, 0, 0],
              [0, 1, 6, 1, 0, 1, 1],
              [0, 0, 1, 6, 2, 0, 0],
              [0, 0, 0, 2, 6, 0, 0],
              [0, 0, 1, 0, 0, 6, 0],
              [0, 0, 1, 0, 0, 0, 6]])
```

(continues on next page)

(continued from previous page)

```
# Two-sided permutation Procrustes (with single transformation)
result = permutation_2sided(A, B, method="approx-normal1", single=True, pad=True)

# Compute the transformed molecule A using transformation matrix P
P = result.t
new_A = np.dot(P.T, np.dot(result.new_a, P)).astype(int)

print("Permutation Matrix:\n", P)
print("\nTransformed A: \n", new_A)
print("\nCompare to Original (padded) B:\n", result.new_b)
print("\nProcrustes Error:", result.error)
```

```
Permutation Matrix:
 [[1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1.]]

Transformed A:
 [[6 3 0 0 0 0 0]
 [3 6 0 1 0 0 0]
 [0 0 0 0 0 0 0]
 [0 1 0 6 2 0 0]
 [0 0 0 2 6 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]

Compare to Original (padded) B:
 [[6 3 0 0 0 0 0]
 [3 6 1 0 0 0 0]
 [0 1 6 1 0 1 1]
 [0 0 1 6 2 0 0]
 [0 0 0 2 6 0 0]
 [0 0 1 0 0 6 0]
 [0 0 1 0 0 0 6]]

Procrustes Error: 118.0
```

---

The corresponding file can be obtained from:

- Jupyter Notebook: `Atom_Atom_Mapping.ipynb`
- Interactive Jupyter Notebook:

---

### 1.3.4 Ranking by Reordering

The problem of ranking a set of objects is ubiquitous not only in everyday life, but also for many scientific problems such as information retrieval, recommender systems, natural language processing, and drug discovery. This problem can be formulated as a 2-sided permutation Procrustes with single transformation.

**Permutation Procrustes 2-Sided with Single-Transformation**

Given matrix $\mathbf{A}_{n\times n}$ and a reference $\mathbf{B}_{n\times n}$, find a permutation of rows/columns of $\mathbf{A}_{n\times n}$ that makes it as close as possible to $\mathbf{B}_{n\times n}$, i.e.,

$$\underbrace{\min}_{\left\{\mathbf{P}\,\middle|\,\substack{p_{ij}\in\{0,1\}\\ \sum_{i=1}^{n} p_{ij}=\sum_{j=1}^{n} p_{ij}=1}\right\}} \|\mathbf{P}^{\dagger}\mathbf{A}\mathbf{P}-\mathbf{B}\|_F^2 \tag{1.7}$$

The code block below, we use the `procrustes` library to rank five American collegiate football teams, where each team plays one game against every other team, using their score-differentials as summarized below (data taken from A. N. Langville, C. D. Meyer, *Ranking by Reordering Methods*, Princeton University Press, 2012, Ch. 8, pp. 97–112). Here, each team is given a zero score for a game they lost (e.g., Duke lost to every other team) and the score difference is calculated for games won (e.g., Miami beat Duke by 45 points and UNC by 18 points). These results are also summarized in the square score-differential matrix $\mathbf{A}$ in **Fig (i)**.

| Team | Duke | Miami | UNC | UVA | VT |
|---|---|---|---|---|---|
| Duke | 0 | 0 | 0 | 0 | 0 |
| Miami | 45 | 0 | 18 | 8 | 20 |
| UNC | 3 | 0 | 0 | 2 | 0 |
| UVA | 31 | 0 | 0 | 0 | 0 |
| VT | 45 | 0 | 27 | 38 | 0 |

Before applying Procrustes, one needs to define a proper target matrix. Traditionally, the rank-differential matrix has been used for this purpose and is defined for $n$ teams as,

$$\mathbf{R}_{n\times n} = \begin{bmatrix} 0 & 1 & 2 & \cdots & n-1 \\ & 0 & 1 & \cdots & n-2 \\ & & \ddots & \ddots & \vdots \\ & & & \ddots & 1 \\ & & & & 0 \end{bmatrix} \tag{1.8}$$

The rank-differential matrix $\mathbf{R} \in \mathbb{R}^{n\times n}$ is an upper-triangular matrix and its $ij$-th element specifies the difference in ranking between team $i$ and team $j$. Considering the rank-differential matrix in **Fig. (ii)** as the target matrix $\mathbf{B}$, the two-sided permutation Procrustes finds the single permutation matrix that maximizes the similarity between the score-differential matrix $\mathbf{A}$ and the rank-differential matrix $\mathbf{B}$. This results to $[5, 2, 4, 3, 1]$ as the final rankings of the teams in **Fig. (iii)**.

(i)

$$\mathbf{A} = \begin{array}{c} \begin{matrix} \text{Duke} & \text{Miami} & \text{UNC} & \text{UVA} & \text{VT} \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 45 & 0 & 18 & 8 & 20 \\ 3 & 0 & 0 & 2 & 0 \\ 31 & 0 & 0 & 0 & 0 \\ 45 & 0 & 27 & 38 & 0 \end{bmatrix} \end{array} \begin{matrix} \text{Duke} \\ \text{Miami} \\ \text{UNC} \\ \text{UVA} \\ \text{VT} \end{matrix}$$

score-differential matrix

(ii)

$$\mathbf{B} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

rank-differential matrix

(iii)

| Team | Ranking |
|---|---|
| Duke | 5 |
| Miami | 2 |
| UNC | 4 |
| UVA | 3 |
| VT | 1 |

[1]:
```
# ranking by reordering with 2-sided permutation procrustes (with single transformation)

import numpy as np

from procrustes import permutation_2sided

# input score-differential matrix
A = np.array([[ 0, 0, 0 ,  0,  0 ],    # Duke
              [45, 0, 18,  8,  20],    # Miami
              [ 3, 0, 0 ,  2,  0 ],    # UNC
              [31, 0, 0 ,  0,  0 ],    # UVA
              [45, 0, 27, 38,  0 ]])   # VT

# make rank-differential matrix
n = A.shape[0]
B = np.zeros((n, n))
for index in range(n):
    B[index, index:] = range(0, n - index)

# rank teams using two-sided Procrustes
result = permutation_2sided(A, B, single=True, method="approx-normal1")

# compute teams' ranks (by adding 1 because Python's list index starts from 0)
_, ranks = np.where(result.t == 1)
ranks += 1
print("Ranks = ", ranks)     # displays [5, 2, 4, 3, 1]
```

```
Ranks =  [5 2 4 3 1]
```

---

The corresponding file can be obtained from:

- Jupyter Notebook: `Ranking_by_Reordering.ipynb`
- Interactive Jupyter Notebook:

---

The user can download the jupyter notebook from our github repo or just use the online version hosted by MyBinder.

## 1.4 References

## 1.5 procrustes.utils

Utility Module.

procrustes.utils.**compute_error**(*a: numpy.ndarray*, *b: numpy.ndarray*, *t: numpy.ndarray*, *s: Optional[numpy.ndarray] = None*) → float

Return the one- or two-sided Procrustes (squared Frobenius norm) error.

The double-sided Procrustes error is defined as

$$\|\mathbf{SAT} - \mathbf{B}\|_F^2 = \mathrm{Tr}\left[(\mathbf{SAT} - \mathbf{B})^\dagger (\mathbf{SAT} - \mathbf{B})\right]$$

when $\mathbf{S}$ is the identity matrix $\mathbf{I}$, this is called the one-sided Procrustes error.

**Parameters**

- **a** (*ndarray*) – The 2D-array $\mathbf{A}_{m\times n}$ which is going to be transformed.
- **b** (*ndarray*) – The 2D-array $\mathbf{B}_{m\times n}$ representing the reference matrix.
- **t** (*ndarray*) – The 2D-array $\mathbf{T}_{n\times n}$ representing the right-hand-side transformation matrix.
- **s** (*ndarray, optional*) – The 2D-array $\mathbf{S}_{m\times m}$ representing the left-hand-side transformation matrix. If set to *None*, the one-sided Procrustes error is computed.

**Returns** **error** – The squared Frobenius norm of difference between the transformed array, $\mathbf{SAT}$, and the reference array, $\mathbf{B}$.

**Return type** float

procrustes.utils.**setup_input_arrays**(*array_a: numpy.ndarray*, *array_b: numpy.ndarray*, *remove_zero_col: bool*, *remove_zero_row: bool*, *pad: bool*, *translate: bool*, *scale: bool*, *check_finite: bool*, *weight: Optional[numpy.ndarray] = None*) → Tuple[numpy.ndarray, numpy.ndarray]

Check and process array inputs for the Procrustes transformation routines.

Usually, the precursor step before all Procrustes methods.

**Parameters**

- **array_a** (*npdarray*) – The 2D array $A$ being transformed.
- **array_b** (*npdarray*) – The 2D reference array $B$.
- **remove_zero_col** (*bool*) – If True, zero columns (values less than 1e-8) on the right side will be removed.
- **remove_zero_row** (*bool*) – If True, zero rows (values less than 1e-8) on the bottom will be removed.
- **pad** (*bool*) – Add zero rows (at the bottom) and/or columns (to the right-hand side) of matrices $\mathbf{A}$ and $\mathbf{B}$ so that they have the same shape.
- **translate** (*bool*) – If true, then translate both arrays $A, B$ to the origin, ie columns of the arrays will have mean zero.
- **scale** – If True, both arrays are normalized to one with respect to the Frobenius norm, ie $Tr(A^T A) = 1$.

- **check_finite** (*bool*) – If true, then checks if both arrays $A$, $B$ are numpy arrays and two-dimensional.
- **weight** (*A list of ndarray or ndarray*) – A list of the weight arrays or one numpy array. When only on numpy array provided, it is assumed that the two arrays $A$ and $B$ share the same weight matrix.

**Returns** Returns the padded arrays, in that they have the same matrix dimensions.

**Return type** (ndarray, ndarray)

**class** procrustes.utils.**ProcrustesResult**

Represents the Procrustes analysis result.

**error**

The Procrustes (squared Frobenius norm) error.

**Type** float

**new_a**

The translated/scaled numpy ndarray $\mathbf{A}$.

**Type** ndarray

**new_b**

The translated/scaled numpy ndarray $\mathbf{B}$.

**Type** ndarray

**t**

The 2D-array $\mathbf{T}$ representing the right-hand-side transformation matrix.

**Type** ndarray

**s**

The 2D-array $\mathbf{S}$ representing the left-hand-side transformation matrix. If set to *None*, the one-sided Procrustes was performed.

**Type** ndarray

### Methods

| | |
|---|---|
| clear() | |
| copy() | |
| fromkeys(iterable[, value]) | Create a new dictionary with keys from iterable and values set to value. |
| get(key[, default]) | Return the value for key if key is in the dictionary, else default. |
| items() | |
| keys() | |
| pop(k[,d]) | If key is not found, d is returned if given, otherwise KeyError is raised |
| popitem() | 2-tuple; but raise KeyError if D is empty. |

continues on next page

Table 1.1 – continued from previous page

| | |
|---|---|
| setdefault(key[, default]) | Insert key with a value of default if key is not in the dictionary. |
| update([E, ]**F) | If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k] |
| values() | |

## 1.6 procrustes.kopt

K-opt (Greedy) Heuristic Module.

procrustes.kopt.**kopt_heuristic_single**(*fun: Callable*, *p0: numpy.ndarray*, *k: int = 3*, *tol: float = 1e-08*) → Tuple[numpy.ndarray, float]

Find a locally-optimal permutation matrix using the k-opt (greedy) heuristic.

$$\underbrace{\min}_{\left\{\mathbf{P}\,\middle|\,\substack{[\mathbf{P}]_{ij}\in\{0,1\}\\ \sum_{i=1}^{n}[\mathbf{P}]_{ij}=\sum_{j=1}^{n}[\mathbf{P}]_{ij}=1}\right\}} f(\mathbf{P})$$

All possible 2-, …, k-fold column-permutations of the initial permutation matrix are tried to identify one which gives a lower value of objective function $f$. Starting from this updated permutation matrix, the process is repeated until no further k-fold column-reordering of a given permutation matrix lower the objective function.

**Parameters**

- **fun** (*callable*) – The objective function $f$ to be minimized.
- **p0** (*ndarray*) – The 2D-array permutation matrix representing the initial guess for $\mathbf{P}$.
- **k** (*int, optional*) – The order of the permutation. For example, *k=3* swaps all possible 3-permutations.
- **tol** (*float, optional*) – When value of the objective function is less than given tolerance, the algorithm stops.

**Returns**

- **p_opt** (*ndarray*) – The locally-optimal permutation matrix $\mathbf{P}$ (i.e., solution).
- **f_opt** (*float*) – The locally-optimal value of objective function given by

procrustes.kopt.**kopt_heuristic_double**(*fun: Callable*, *p1: numpy.ndarray*, *p2: numpy.ndarray*, *k: int = 3*, *tol: float = 1e-08*) → Tuple[numpy.ndarray, numpy.ndarray, float]

Find locally-optimal permutation matrices using the k-opt (greedy) heuristic.

$$\underbrace{\arg\min}_{\left\{\mathbf{P}_1,\mathbf{P}_2\,\middle|\,\substack{[\mathbf{P}_1]_{ij}\in\{0,1\}\\ [\mathbf{P}_2]_{ij}\in\{0,1\}\\ \sum_{i=1}^{m}[\mathbf{P}_1]_{ij}=\sum_{j=1}^{m}[\mathbf{P}_1]_{ij}=1\\ \sum_{i=1}^{n}[\mathbf{P}_2]_{ij}=\sum_{j=1}^{n}[\mathbf{P}_2]_{ij}=1}\right\}} f(\mathbf{P}_1,\mathbf{P}_2)$$

All possible 2-, …, k-fold permutations of the initial permutation matrices are tried to identify ones which give a lower value of objective function $f$. This corresponds to row-swaps for $\mathbf{P}_1$ and column-swaps for $\mathbf{P}_2$. Starting from these updated permutation matrices, the process is repeated until no further k-fold reordering of either permutation matrix lower the objective function.

**Parameters**

- **fun** (*callable*) – The objective function $f$ to be minimized.
- **p1** (*ndarray*) – The 2D-array permutation matrix representing the initial guess for $\mathbf{P}_1$.
- **p2** (*ndarray*) – The 2D-array permutation matrix representing the initial guess for $\mathbf{P}_2$.
- **k** (*int, optional*) – The order of the permutation. For example, k=3 swaps all possible 3-permutations.
- **tol** (*float, optional*) – When value of the objective function is less than given tolerance, the algorithm stops.

**Returns**

- **p1_opt** (*ndarray*) – The locally-optimal permutation matrix $\mathbf{P}_1$.
- **p2_opt** (*ndarray*) – The locally-optimal permutation matrix $\mathbf{P}_2$.
- **f_opt** (*float*) – The locally-optimal value of objective function given by

## 1.7 procrustes.generic

Generic Procrustes Module.

procrustes.generic.**generic**(*a: numpy.ndarray*, *b: numpy.ndarray*, *pad: bool = True*, *translate: bool = False*, *scale: bool = False*, *unpad_col: bool = False*, *unpad_row: bool = False*, *check_finite: bool = True*, *weight: Optional[numpy.ndarray] = None*, *use_svd: bool = False*) → *procrustes.utils.ProcrustesResult*

Perform generic one-sided Procrustes.

Given matrix $\mathbf{A}_{m\times n}$ and a reference matrix $\mathbf{B}_{m\times n}$, find the transformation matrix $\mathbf{T}_{n\times n}$ that makes $\mathbf{AT}$ as close as possible to $\mathbf{B}$. In other words,

$$\underbrace{\min}_{\mathbf{T}} \quad \|\mathbf{AT} - \mathbf{B}\|_F^2$$

This Procrustes method requires the $\mathbf{A}$ and $\mathbf{B}$ matrices to have the same shape, which is gauranteed with the default pad argument for any given $\mathbf{A}$ and $\mathbf{B}$ matrices. In preparing the $\mathbf{A}$ and $\mathbf{B}$ matrices, the (optional) order of operations is: **1)** unpad zero rows/columns, **2)** translate the matrices to the origin, **3)** weight entries of $\mathbf{A}$, **4)** scale the matrices to have unit norm, **5)** pad matrices with zero rows/columns so they have the same shape.

**Parameters**

- **a** (*ndarray*) – The 2D-array $\mathbf{A}$ which is going to be transformed.
- **b** (*ndarray*) – The 2D-array $\mathbf{B}$ representing the reference matrix.
- **pad** (*bool, optional*) – Add zero rows (at the bottom) and/or columns (to the right-hand side) of matrices $\mathbf{A}$ and $\mathbf{B}$ so that they have the same shape.
- **translate** (*bool, optional*) – If True, both arrays are centered at origin (columns of the arrays will have mean zero).
- **scale** (*bool, optional*) – If True, both arrays are normalized with respect to the Frobenius norm, i.e., Tr $\left[\mathbf{A}^\dagger\mathbf{A}\right] = 1$ and Tr $\left[\mathbf{B}^\dagger\mathbf{B}\right] = 1$.
- **unpad_col** (*bool, optional*) – If True, zero columns (with values less than 1.0e-8) on the right-hand side of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.
- **unpad_row** (*bool, optional*) – If True, zero rows (with values less than 1.0e-8) at the bottom of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.

- **check_finite** (*bool, optional*) – If True, convert the input to an array, checking for NaNs or Infs.
- **weight** (*ndarray, optional*) – The 1D-array representing the weights of each row of $\mathbf{A}$. This defines the elements of the diagonal matrix $\mathbf{W}$ that is multiplied by $\mathbf{A}$ matrix, i.e., $\mathbf{A} \rightarrow \mathbf{WA}$.
- **use_svd** (*bool, optional*) – If True, the (Moore-Penrose) pseudo-inverse is computed by singular-value decomposition (SVD) including all 'large' singular values (using *scipy.linalg.pinv2*). If False, the the (Moore-Penrose) pseudo-inverse is computed by least-squares solver (using *scipy.linalg.pinv*). The least-squares implementation is less efficient, but more robust, than the SVD implementation.

**Returns** **res** – The Procrustes result represented as a class:*utils.ProcrustesResult* object.

**Return type** *ProcrustesResult*

### Notes

The optimal transformation matrix is obtained by solving the least-squares equations,

$$\mathbf{X}_{\text{opt}} = \left(\mathbf{A}^\top \mathbf{A}\right)^{-1} \mathbf{A}^\top \mathbf{B}$$

If $m < n$, the transformation matrix $\mathbf{T}_{\text{opt}}$ is not unique, because the system of equations is underdetermined (i.e., there are fewer equations than unknowns).

## 1.8 procrustes.generalized

Generalized Procrustes Module.

procrustes.generalized.**generalized**(*array_list: List[numpy.ndarray]*, *ref: Optional[numpy.ndarray] = None*, *tol: float = 1e-07*, *n_iter: int = 200*, *check_finite: bool = True*) → Tuple[List[numpy.ndarray], float]

Generalized Procrustes Analysis.

**Parameters**

- **array_list** (*List*) – The list of 2D-array which is going to be transformed.
- **ref** (*ndarray, optional*) – The reference array to initialize the first iteration. If None, the first array in *array_list* will be used.
- **tol** (*float, optional*) – Tolerance value to stop the iterations.
- **n_iter** (*int, optional*) – Number of total iterations.
- **check_finite** (*bool, optional*) – If true, convert the input to an array, checking for NaNs or Infs.

**Returns**

- **array_aligned** (*List*) – A list of transformed arrays with generalized Procrustes analysis.
- **new_distance_gpa** (*float*) – The distance for matching all the transformed arrays with generalized Procrustes analysis.

### Notes

Given a set of matrices, $\mathbf{A}_1, \mathbf{A}_2, \cdots, \mathbf{A}_k$ with $k > 2$, the objective is to minimize in order to superimpose pairs of matrices.

$$\min \quad = \sum_{i<j}^{j} \|\mathbf{A}_i\mathbf{T}_i - \mathbf{A}_j\mathbf{T}_j\|^2$$

This function implements the Equation (20) and the corresponding algorithm in Gower's paper.

## 1.9 procrustes.orthogonal

Orthogonal Procrustes Module.

procrustes.orthogonal.**orthogonal**(*a: numpy.ndarray*, *b: numpy.ndarray*, *pad: bool = True*, *translate: bool = False*, *scale: bool = False*, *unpad_col: bool = False*, *unpad_row: bool = False*, *check_finite: bool = True*, *weight: Optional[numpy.ndarray] = None*, *lapack_driver: str = 'gesvd'*) → *procrustes.utils.ProcrustesResult*

Perform orthogonal Procrustes.

Given a matrix $\mathbf{A}_{m\times n}$ and a reference matrix $\mathbf{B}_{m\times n}$, find the orthogonal transformation matrix $\mathbf{Q}_{n\times n}$ that makes $\mathbf{AQ}$ as close as possible to $\mathbf{B}$. In other words,

$$\underbrace{\min}_{\{\mathbf{Q}|\mathbf{Q}^{-1}=\mathbf{Q}^{\dagger}\}} \quad \|\mathbf{AQ} - \mathbf{B}\|_F^2$$

This Procrustes method requires the $\mathbf{A}$ and $\mathbf{B}$ matrices to have the same shape, which is gauranteed with the default `pad` argument for any given $\mathbf{A}$ and $\mathbf{B}$ matrices. In preparing the $\mathbf{A}$ and $\mathbf{B}$ matrices, the (optional) order of operations is: **1)** unpad zero rows/columns, **2)** translate the matrices to the origin, **3)** weight entries of $\mathbf{A}$, **4)** scale the matrices to have unit norm, **5)** pad matrices with zero rows/columns so they have the same shape.

**Parameters**

- **a** (*ndarray*) – The 2D-array $\mathbf{A}$ which is going to be transformed.
- **b** (*ndarray*) – The 2D-array $\mathbf{B}$ representing the reference matrix.
- **pad** (*bool, optional*) – Add zero rows (at the bottom) and/or columns (to the right-hand side) of matrices $\mathbf{A}$ and $\mathbf{B}$ so that they have the same shape.
- **translate** (*bool, optional*) – If True, both arrays are centered at origin (columns of the arrays will have mean zero).
- **scale** (*bool, optional*) – If True, both arrays are normalized with respect to the Frobenius norm, i.e., Tr $\left[\mathbf{A}^{\dagger}\mathbf{A}\right] = 1$ and Tr $\left[\mathbf{B}^{\dagger}\mathbf{B}\right] = 1$.
- **unpad_col** (*bool, optional*) – If True, zero columns (with values less than 1.0e-8) on the right-hand side of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.
- **unpad_row** (*bool, optional*) – If True, zero rows (with values less than 1.0e-8) at the bottom of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.
- **check_finite** (*bool, optional*) – If True, convert the input to an array, checking for NaNs or Infs.
- **weight** (*ndarray, optional*) – The 1D-array representing the weights of each row of $\mathbf{A}$. This defines the elements of the diagonal matrix $\mathbf{W}$ that is multiplied by $\mathbf{A}$ matrix, i.e., $\mathbf{A} \rightarrow \mathbf{WA}$.

- **lapack_driver** (*{'gesvd', 'gesdd'}, optional*) – Whether to use the more efficient divide-and-conquer approach ('gesdd') or the more robust general rectangular approach ('gesvd') to compute the singular-value decomposition with *scipy.linalg.svd*.

**Returns** **res** – The Procrustes result represented as a class:*utils.ProcrustesResult* object.

**Return type** *ProcrustesResult*

### Notes

The optimal orthogonal matrix is obtained by,

$$\mathbf{Q}^{\text{opt}} = \arg \underbrace{\min}_{\{\mathbf{Q}|\mathbf{Q}^{-1}=\mathbf{Q}^{\dagger}\}} \|\mathbf{AQ} - \mathbf{B}\|_F^2 = \arg \underbrace{\max}_{\{\mathbf{Q}|\mathbf{Q}^{-1}=\mathbf{Q}^{\dagger}\}} \text{Tr}\left[\mathbf{Q}^{\dagger}\mathbf{A}^{\dagger}\mathbf{B}\right]$$

The solution is obtained using the singular value decomposition (SVD) of the $\mathbf{A}^{\dagger}\mathbf{B}$ matrix,

$$\mathbf{A}^{\dagger}\mathbf{B} = \tilde{\mathbf{U}}\tilde{\mathbf{\Sigma}}\tilde{\mathbf{V}}^{\dagger}$$
$$\mathbf{Q}^{\text{opt}} = \tilde{\mathbf{U}}\tilde{\mathbf{V}}^{\dagger}$$

The singular values are always listed in decreasing order, with the smallest singular value in the bottom-right-hand corner of $\tilde{\mathbf{\Sigma}}$.

### Examples

```
>>> import numpy as np
>>> from scipy.stats import ortho_group
>>> from procrustes import orthogonal
>>> a = np.random.rand(5, 3)   # random input matrix
>>> q = ortho_group.rvs(3)     # random orthogonal transformation
>>> b = np.dot(a, q) + np.random.rand(1, 3)   # random target matrix
>>> result = orthogonal(a, b, translate=True, scale=False)
>>> print(result.error)      # error (should be zero)
>>> print(result.t)          # transformation matrix (same as q)
>>> print(result.new_a)      # translated array a
>>> print(result.new_b)      # translated array b
```

procrustes.orthogonal.**orthogonal_2sided**(*a: numpy.ndarray*, *b: numpy.ndarray*, *single: bool = True*, *pad: bool = True*, *translate: bool = False*, *scale: bool = False*, *unpad_col: bool = False*, *unpad_row: bool = False*, *check_finite: bool = True*, *weight: Optional[numpy.ndarray] = None*, *lapack_driver: str = 'gesvd'*) → *procrustes.utils.ProcrustesResult*

Perform two-sided orthogonal Procrustes with one- or two-transformations.

**Two Transformations:** Given a matrix $\mathbf{A}_{m\times n}$ and a reference matrix $\mathbf{B}_{m\times n}$, find two $n \times n$ orthogonal transformation matrices $\mathbf{Q}_1^{\dagger}$ and $\mathbf{Q}_2$ that makes $\mathbf{Q}_1^{\dagger}\mathbf{A}\mathbf{Q}_2$ as close as possible to $\mathbf{B}$. In other words,

$$\underbrace{\min}_{\left\{\begin{smallmatrix}\mathbf{Q}_1\\\mathbf{Q}_2\end{smallmatrix}\middle|\begin{smallmatrix}\mathbf{Q}_1^{-1}=\mathbf{Q}_1^{\dagger}\\\mathbf{Q}_2^{-1}=\mathbf{Q}_2^{\dagger}\end{smallmatrix}\right\}} \|\mathbf{Q}_1^{\dagger}\mathbf{A}\mathbf{Q}_2 - \mathbf{B}\|_F^2$$

**Single Transformations:** Given a **symmetric** matrix $\mathbf{A}_{n\times n}$ and a reference $\mathbf{B}_{n\times n}$, find one orthogonal transformation matrix $\mathbf{Q}_{n\times n}$ that makes $\mathbf{A}$ as close as possible to $\mathbf{B}$. In other words,

$$\underbrace{\min}_{\{\mathbf{Q}|\mathbf{Q}^{-1}=\mathbf{Q}^{\dagger}\}} \|\mathbf{Q}^{\dagger}\mathbf{A}\mathbf{Q} - \mathbf{B}\|_F^2$$

This Procrustes method requires the $\mathbf{A}$ and $\mathbf{B}$ matrices to have the same shape, which is gauranteed with the default `pad` argument for any given $\mathbf{A}$ and $\mathbf{B}$ matrices. In preparing the $\mathbf{A}$ and $\mathbf{B}$ matrices, the (optional) order of operations is: **1)** unpad zero rows/columns, **2)** translate the matrices to the origin, **3)** weight entries of $\mathbf{A}$, **4)** scale the matrices to have unit norm, **5)** pad matrices with zero rows/columns so they have the same shape.

**Parameters**

- **a** (*ndarray*) – The 2D-array $\mathbf{A}$ which is going to be transformed.
- **b** (*ndarray*) – The 2D-array $\mathbf{B}$ representing the reference matrix.
- **single** (*bool, optional*) – If True, single transformation is used (i.e., $\mathbf{Q}_1 = \mathbf{Q}_2 = \mathbf{Q}$), otherwise, two transformations are used.
- **pad** (*bool, optional*) – Add zero rows (at the bottom) and/or columns (to the right-hand side) of matrices $\mathbf{A}$ and $\mathbf{B}$ so that they have the same shape.
- **translate** (*bool, optional*) – If True, both arrays are centered at origin (columns of the arrays will have mean zero).
- **scale** (*bool, optional*) – If True, both arrays are normalized with respect to the Frobenius norm, i.e., $\mathrm{Tr}\left[\mathbf{A}^\dagger\mathbf{A}\right] = 1$ and $\mathrm{Tr}\left[\mathbf{B}^\dagger\mathbf{B}\right] = 1$.
- **unpad_col** (*bool, optional*) – If True, zero columns (with values less than 1.0e-8) on the right-hand side of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.
- **unpad_row** (*bool, optional*) – If True, zero rows (with values less than 1.0e-8) at the bottom of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.
- **check_finite** (*bool, optional*) – If True, convert the input to an array, checking for NaNs or Infs.
- **weight** (*ndarray, optional*) – The 1D-array representing the weights of each row of $\mathbf{A}$. This defines the elements of the diagonal matrix $\mathbf{W}$ that is multiplied by $\mathbf{A}$ matrix, i.e., $\mathbf{A} \to \mathbf{W}\mathbf{A}$.
- **lapack_driver** (*{"gesvd", "gesdd"}, optional*) – Used in the singular value decomposition function from SciPy. Only allowed two options, with "gesvd" being less-efficient than "gesdd" but is more robust. Default is "gesvd".

**Returns** **res** – The Procrustes result represented as a class:*utils.ProcrustesResult* object.

**Return type** *ProcrustesResult*

### Notes

**Two-Sided Orthogonal Procrustes with Two Transformations:** The optimal orthogonal transformations are obtained by:

$$\mathbf{Q}_1^{\text{opt}}, \mathbf{Q}_2^{\text{opt}} = \arg \underbrace{\min}_{\left\{ \substack{\mathbf{Q}_1 \\ \mathbf{Q}_2} \middle| \substack{\mathbf{Q}_1^{-1}=\mathbf{Q}_1^\dagger \\ \mathbf{Q}_2^{-1}=\mathbf{Q}_2^\dagger} \right\}} \|\mathbf{Q}_1^\dagger \mathbf{A} \mathbf{Q}_2 - \mathbf{B}\|_F^2 = \arg \underbrace{\max}_{\left\{ \substack{\mathbf{Q}_1 \\ \mathbf{Q}_2} \middle| \substack{\mathbf{Q}_1^{-1}=\mathbf{Q}_1^\dagger \\ \mathbf{Q}_2^{-1}=\mathbf{Q}_2^\dagger} \right\}} \mathrm{Tr}\left[\mathbf{Q}_2^\dagger \mathbf{A}^\dagger \mathbf{Q}_1 \mathbf{B}\right]$$

This is solved by taking the singular value decomposition (SVD) of $\mathbf{A}$ and $\mathbf{B}$,

$$\mathbf{A} = \mathbf{U}_A \mathbf{\Sigma}_A \mathbf{V}_A^\dagger$$
$$\mathbf{B} = \mathbf{U}_B \mathbf{\Sigma}_B \mathbf{V}_B^\dagger$$

Then the two optimal orthogonal matrices are given by,

$$\mathbf{Q}_1^{\text{opt}} = \mathbf{U}_A \mathbf{U}_B^\dagger$$
$$\mathbf{Q}_2^{\text{opt}} = \mathbf{V}_A \mathbf{V}_B^\dagger$$

**Two-Sided Orthogonal Procrustes with Single-Transformation:** The optimal orthogonal transformation is obtained by:

$$\mathbf{Q}^{\text{opt}} = \arg \underbrace{\min}_{\{\mathbf{Q}|\mathbf{Q}^{-1}=\mathbf{Q}^{\dagger}\}} \|\mathbf{Q}^{\dagger}\mathbf{A}\mathbf{Q} - \mathbf{B}\|_F^2 = \arg \underbrace{\max}_{\{\mathbf{Q}|\mathbf{Q}^{-1}=\mathbf{Q}^{\dagger}\}} \text{Tr}\left[\mathbf{Q}^{\dagger}\mathbf{A}^{\dagger}\mathbf{Q}\mathbf{B}\right]$$

Using the singular value decomposition (SVD) of $\mathbf{A}$ and $\mathbf{B}$,

$$\mathbf{A} = \mathbf{U}_A\mathbf{\Lambda}_A\mathbf{U}_A^{\dagger}$$
$$\mathbf{B} = \mathbf{U}_B\mathbf{\Lambda}_B\mathbf{U}_B^{\dagger}$$

The optimal orthogonal matrix $\mathbf{Q}^{\text{opt}}$ is obtained through,

$$\mathbf{Q}^{\text{opt}} = \mathbf{U}_A\mathbf{S}\mathbf{U}_B^{\dagger}$$

where $\mathbf{S}$ is a diagonal matrix with $\pm 1$ elements,

$$\mathbf{S} = \begin{bmatrix} \pm 1 & 0 & \cdots & 0 \\ 0 & \pm 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \pm 1 \end{bmatrix}$$

The matrix $\mathbf{S}$ is chosen to be the identity matrix.

### Examples

```
>>> import numpy as np
>>> a = np.array([[30, 33, 20], [33, 53, 43], [20, 43, 46]])
>>> b = np.array([[ 22.78131838, -0.58896768,-43.00635291, 0., 0.],
...               [ -0.58896768, 16.77132475,  0.24289990, 0., 0.],
...               [-43.00635291,  0.2428999 , 89.44735687, 0., 0.],
...               [  0.        ,  0.        ,  0.        , 0., 0.]])
>>> res = orthogonal_2sided(a, b, single=True, pad=True, unpad_col=True)
>>> res.t
array([[ 0.25116633,  0.76371527,  0.59468855],
       [-0.95144277,  0.08183302,  0.29674906],
       [ 0.17796663, -0.64034549,  0.74718507]])
>>> res.error
1.9646186414076689e-26
```

## 1.10 procrustes.permutation

Permutation Procrustes Module.

procrustes.permutation.**permutation**(*a: numpy.ndarray*, *b: numpy.ndarray*, *pad: bool = True*, *translate: bool = False*, *scale: bool = False*, *unpad_col: bool = False*, *unpad_row: bool = False*, *check_finite: bool = True*, *weight: Optional[numpy.ndarray] = None*) → *procrustes.utils.ProcrustesResult*

Perform one-sided permutation Procrustes.

Given matrix $\mathbf{A}_{m\times n}$ and a reference matrix $\mathbf{B}_{m\times n}$, find the permutation transformation matrix $\mathbf{P}_{n\times n}$ that makes $\mathbf{AP}$ as close as possible to $\mathbf{B}$. In other words,

$$\underbrace{\min}_{\left\{\mathbf{P}\left|\begin{smallmatrix}[\mathbf{P}]_{ij}\in\{0,1\}\\ \sum_{i=1}^{n}[\mathbf{P}]_{ij}=\sum_{j=1}^{n}[\mathbf{P}]_{ij}=1\end{smallmatrix}\right.\right\}} \|\mathbf{AP}-\mathbf{B}\|_F^2$$

This Procrustes method requires the $\mathbf{A}$ and $\mathbf{B}$ matrices to have the same shape, which is guaranteed with the default `pad=True` argument for any given $\mathbf{A}$ and $\mathbf{B}$ matrices. In preparing the $\mathbf{A}$ and $\mathbf{B}$ matrices, the (optional) order of operations is: **1)** unpad zero rows/columns, **2)** translate the matrices to the origin, **3)** weight entries of $\mathbf{A}$, **4)** scale the matrices to have unit norm, **5)** pad matrices with zero rows/columns so they have the same shape.

**Parameters**

- **a** (*ndarray*) – The 2D-array $\mathbf{A}$ which is going to be transformed.
- **b** (*ndarray*) – The 2D-array $\mathbf{B}$ representing the reference matrix.
- **pad** (*bool, optional*) – Add zero rows (at the bottom) and/or columns (to the right-hand side) of matrices $\mathbf{A}$ and $\mathbf{B}$ so that they have the same shape.
- **translate** (*bool, optional*) – If True, both arrays are centered at origin (columns of the arrays will have mean zero).
- **scale** (*bool, optional*) – If True, both arrays are normalized with respect to the Frobenius norm, i.e., $\mathrm{Tr}\left[\mathbf{A}^{\dagger}\mathbf{A}\right]=1$ and $\mathrm{Tr}\left[\mathbf{B}^{\dagger}\mathbf{B}\right]=1$.
- **unpad_col** (*bool, optional*) – If True, zero columns (with values less than 1.0e-8) on the right-hand side are removed.
- **unpad_row** (*bool, optional*) – If True, zero rows (with values less than 1.0e-8) at the bottom are removed.
- **check_finite** (*bool, optional*) – If True, convert the input to an array, checking for NaNs or Infs.
- **weight** (*ndarray, optional*) – The 1D-array representing the weights of each row of $\mathbf{A}$. This defines the elements of the diagonal matrix $\mathbf{W}$ that is multiplied by $\mathbf{A}$ matrix, i.e., $\mathbf{A}\rightarrow\mathbf{WA}$.

**Returns** **res** – The Procrustes result represented as a class:*utils.ProcrustesResult* object.

**Return type** *ProcrustesResult*

### Notes

The optimal $n\times n$ permutation matrix is obtained by,

$$\mathbf{P}^{\mathrm{opt}}=\arg\underbrace{\min}_{\left\{\mathbf{P}\left|\begin{smallmatrix}[\mathbf{P}]_{ij}\in\{0,1\}\\ \sum_{i=1}^{n}[\mathbf{P}]_{ij}=\sum_{j=1}^{n}[\mathbf{P}]_{ij}=1\end{smallmatrix}\right.\right\}}\|\mathbf{AP}-\mathbf{B}\|_F^2=\underbrace{\max}_{\left\{\mathbf{P}\left|\begin{smallmatrix}[\mathbf{P}]_{ij}\in\{0,1\}\\ \sum_{i=1}^{n}[\mathbf{P}]_{ij}=\sum_{j=1}^{n}[\mathbf{P}]_{ij}=1\end{smallmatrix}\right.\right\}}\mathrm{Tr}\left[\mathbf{P}^{\dagger}\mathbf{A}^{\dagger}\mathbf{B}\right]$$

The solution is found by relaxing the problem into a linear programming problem. The solution to a linear programming problem is always at the boundary of the allowed region. So,

$$\underbrace{\max}_{\left\{\mathbf{P}\left|\begin{smallmatrix}[\mathbf{P}]_{ij}\in\{0,1\}\\ \sum_{i=1}^{n}[\mathbf{P}]_{ij}=\sum_{j=1}^{n}[\mathbf{P}]_{ij}=1\end{smallmatrix}\right.\right\}}\mathrm{Tr}\left[\mathbf{P}^{\dagger}\mathbf{A}^{\dagger}\mathbf{B}\right]=\underbrace{\max}_{\left\{\mathbf{P}\left|\begin{smallmatrix}[\mathbf{P}]_{ij}\geq 0\\ \sum_{i=1}^{n}[\mathbf{P}]_{ij}=\sum_{j=1}^{n}[\mathbf{P}]_{ij}=1\end{smallmatrix}\right.\right\}}\mathrm{Tr}\left[\mathbf{P}^{\dagger}\left(\mathbf{A}^{\dagger}\mathbf{B}\right)\right]$$

This is a matching problem and can be solved by the Hungarian algorithm. The cost matrix is defined as $\mathbf{A}^{\dagger}\mathbf{B}$ and the *scipy.optimize.linear_sum_assignment* is used to solve for the permutation that maximizes the linear sum assignment problem.

procrustes.permutation.**permutation_2sided**(*a: numpy.ndarray*, *b: numpy.ndarray*, *single: bool = True*, *method: str = 'kopt'*, *guess_p1: Optional[numpy.ndarray] = None*, *guess_p2: Optional[numpy.ndarray] = None*, *pad: bool = False*, *unpad_col: bool = False*, *unpad_row: bool = False*, *translate: bool = False*, *scale: bool = False*, *check_finite: bool = True*, *options: Optional[dict] = None*, *weight: Optional[numpy.ndarray] = None*, *lapack_driver: str = 'gesvd'*) → *procrustes.utils.ProcrustesResult*

Perform two-sided permutation Procrustes.

**Parameters**

- **a** (*ndarray*) – The 2D-array $\mathbf{A}$ which is going to be transformed.
- **b** (*ndarray*) – The 2D-array $\mathbf{B}$ representing the reference matrix.
- **single** (*bool, optional*) – If *True*, the single-transformation Procrustes is performed to obtain $\mathbf{P}$. If *False*, the two-transformations Procrustes is performed to obtain $\mathbf{P}_1$ and $\mathbf{P}_2$.
- **method** (*str, optional*) – The method to solve for permutation matrices. For *single=False*, these include "flip-flop" and "k-opt" methods. For *single=True*, these include "approx-normal1", "approx-normal2", "approx-umeyama", "approx-umeyama-svd", "k-opt", "soft-assign", and "nmf".
- **guess_p1** (*np.ndarray, optional*) – Guess for $\mathbf{P}_1$ matrix given as a 2D-array. This is only required for the two-transformations case specified by setting *single=False*.
- **guess_p2** (*np.ndarray, optional*) – Guess for $\mathbf{P}_2$ matrix given as a 2D-array.
- **pad** (*bool, optional*) – Add zero rows (at the bottom) and/or columns (to the right-hand side) of matrices $\mathbf{A}$ and $\mathbf{B}$ so that they have the same shape.
- **unpad_col** (*bool, optional*) – If True, zero columns (with values less than 1.0e-8) on the right-hand side are removed.
- **unpad_row** (*bool, optional*) – If True, zero rows (with values less than 1.0e-8) at the bottom are removed.
- **translate** (*bool, optional*) – If True, both arrays are centered at origin (columns of the arrays will have mean zero).
- **scale** (*bool, optional*) – If True, both arrays are normalized with respect to the Frobenius norm, i.e., Tr $\left[\mathbf{A}^{\dagger}\mathbf{A}\right] = 1$ and Tr $\left[\mathbf{B}^{\dagger}\mathbf{B}\right] = 1$.
- **check_finite** (*bool, optional*) – If True, convert the input to an array, checking for NaNs or Infs.
- **options** (*dict, optional*) – A dictionary of method options.
- **weight** (*ndarray, optional*) – The 1D-array representing the weights of each row of $\mathbf{A}$. This defines the elements of the diagonal matrix $\mathbf{W}$ that is multiplied by $\mathbf{A}$ matrix, i.e., $\mathbf{A} \to \mathbf{WA}$.
- **lapack_driver** (*{'gesvd', 'gesdd'}, optional*) – Whether to use the more efficient divide-and-conquer approach ('gesdd') or the more robust general rectangular approach ('gesvd') to compute the singular-value decomposition with *scipy.linalg.svd*.

**Returns** **res** – The Procrustes result represented as a class:*utils.ProcrustesResult* object.

**Return type** *ProcrustesResult*

### Notes

Given matrix $\mathbf{A}_{n\times n}$ and a reference $\mathbf{B}_{n\times n}$, find a permutation of rows/columns of $\mathbf{A}_{n\times n}$ that makes it as close as possible to $\mathbf{B}_{n\times n}$. I.e.,

$$\begin{aligned}
&\underbrace{\min}_{\left\{\mathbf{P}\middle|\substack{p_{ij}\in\{0,1\}\\ \sum_{i=1}^{n} p_{ij}=\sum_{j=1}^{n} p_{ij}=1}\right\}} \|\mathbf{P}^{\dagger}\mathbf{A}\mathbf{P}-\mathbf{B}\|_F^2 \\
=&\underbrace{\min}_{\left\{\mathbf{P}\middle|\substack{p_{ij}\in\{0,1\}\\ \sum_{i=1}^{n} p_{ij}=\sum_{j=1}^{n} p_{ij}=1}\right\}} \mathrm{Tr}\left[\left(\mathbf{P}^{\dagger}\mathbf{A}\mathbf{P}-\mathbf{B}\right)^{\dagger}\left(\mathbf{P}^{\dagger}\mathbf{A}\mathbf{P}-\mathbf{B}\right)\right] \\
=&\underbrace{\max}_{\left\{\mathbf{P}\middle|\substack{p_{ij}\in\{0,1\}\\ \sum_{i=1}^{n} p_{ij}=\sum_{j=1}^{n} p_{ij}=1}\right\}} \mathrm{Tr}\left[\mathbf{P}^{\dagger}\mathbf{A}^{\dagger}\mathbf{P}\mathbf{B}\right]
\end{aligned}$$

Here, $\mathbf{P}_{n\times n}$ is the permutation matrix. Given an intial guess, the best local minimum can be obtained by the iterative procedure,

$$p_{ij}^{(n+1)} = p_{ij}^{(n)} \sqrt{\frac{2\left[\mathbf{T}^{(n)}\right]_{ij}}{\left[\mathbf{P}^{(n)}\left(\left(\mathbf{P}^{(n)}\right)^{T}\mathbf{T}+\left(\left(\mathbf{P}^{(n)}\right)^{T}\mathbf{T}\right)^{T}\right)\right]_{ij}}}$$

where,

$$\mathbf{T}^{(n)} = \mathbf{A}\mathbf{P}^{(n)}\mathbf{B}$$

Using an initial guess, the iteration can stops when the change in $d$ is below the specified threshold,

$$d = \mathrm{Tr}\left[\left(\mathbf{P}^{(n+1)}-\mathbf{P}^{(n)}\right)^{T}\left(\mathbf{P}^{(n+1)}-\mathbf{P}^{(n)}\right)\right]$$

The outcome of the iterative procedure $\mathbf{P}^{(\infty)}$ is not a permutation matrix. So, the closest permutation can be found by setting `refinement=True`. This uses `procrustes.permutation.PermutationProcrustes` to find the closest permutation; that is,

$$\underbrace{\min}_{\left\{\mathbf{P}\middle|\substack{p_{ij}\in\{0,1\}\\ \sum_{i=1}^{n} p_{ij}=\sum_{j=1}^{n} p_{ij}=1}\right\}} \|\mathbf{P}-\mathbf{P}^{(\infty)}\|_F^2 = \underbrace{\max}_{\left\{\mathbf{P}\middle|\substack{p_{ij}\in\{0,1\}\\ \sum_{i=1}^{n} p_{ij}=\sum_{j=1}^{n} p_{ij}=1}\right\}} \mathrm{Tr}\left[\mathbf{P}^{\dagger}\mathbf{P}^{(\infty)}\right]$$

The answer to this problem is a heuristic solution for the matrix-matching problem that seems to be relatively accurate.

**Initial Guess:**

Two possible initial guesses are inferred from the Umeyama procedure. One can find either the closest permutation matrix to $\mathbf{U}_{\text{Umeyama}}$ or to $\mathbf{U}_{\text{Umeyama}}^{\text{approx.}}$.

Considering the `procrustes.permutation.PermutationProcrustes`, the resulting permutation matrix can be specified as initial guess through `guess=umeyama` and `guess=umeyama_approx`, which solves:

$$\begin{aligned}
&\underbrace{\max}_{\left\{\mathbf{P}\middle|\substack{p_{ij}\in\{0,1\}\\ \sum_{i=1}^{n} p_{ij}=\sum_{j=1}^{n} p_{ij}=1}\right\}} \mathrm{Tr}\left[\mathbf{P}^{\dagger}\mathbf{U}_{\text{Umeyama}}\right] \\
&\underbrace{\max}_{\left\{\mathbf{P}\middle|\substack{p_{ij}\in\{0,1\}\\ \sum_{i=1}^{n} p_{ij}=\sum_{j=1}^{n} p_{ij}=1}\right\}} \mathrm{Tr}\left[\mathbf{P}^{\dagger}\mathbf{U}_{\text{Umeyama}}^{\text{approx.}}\right]
\end{aligned}$$

Another choice is to start by solving a normal permutation Procrustes problem. In other words, write new matrices, $\mathbf{A}^0$ and $\mathbf{B}^0$, with columns like,

$$\begin{bmatrix} a_{ii} \\ p \cdot \text{sgn}\left(a_{ij_{\text{max}}}\right) \underbrace{\text{max}}_{1 \leq j \leq n} \left(|a_{ij}|\right) \\ p^2 \cdot \text{sgn}\left(a_{ij_{\text{max}-1}}\right) \underbrace{\text{max} - 1}_{1 \leq j \leq n} \left(|a_{ij}|\right) \\ \vdots \end{bmatrix}$$

Here, $\text{max} - 1$ denotes the second-largest absolute value of elements, $\text{max} - 2$ is the third-largest abosule value of elements, etc.

The matrices $\mathbf{A}^0$ and $\mathbf{B}^0$ have the diagonal elements of $\mathbf{A}$ and $\mathbf{B}$ in the first row, and below the first row has the largest off-diagonal element in row $i$, the second-largest off-diagonal element, etc. The elements are weighted by a factor $0 < p < 1$, so that the smaller elements are considered less important for matching. The matrices can be truncated after a few terms; for example, after the size of elements falls below some threshold. A reasonable choice would be to stop after $\lfloor \frac{-2 \ln 10}{\ln p} + 1 \rfloor$ rows; this ensures that the size of the elements in the last row is less than 1% of those in the first off-diagonal row.

There are obviously many different ways to construct the matrices $\mathbf{A}^0$ and $\mathbf{B}^0$. Another, even better, method would be to try to encode not only what the off-diagonal elements are, but which element in the matrix they correspond to. One could do that by not only listing the diagonal elements, but also listing the associated off-diagonal element. I.e., the columns of $\mathbf{A}^0$ and $\mathbf{B}^0$ would be,

$$\begin{bmatrix} a_{ii} \\ p \cdot a_{j_{\text{max}} j_{\text{max}}} \\ p \cdot \text{sgn}\left(a_{ij_{\text{max}}}\right) \underbrace{\text{max}}_{1 \leq j \leq n} \left(|a_{ij}|\right) \\ p^2 \cdot a_{j_{\text{max}-1} j_{\text{max}-1}} \\ p^2 \cdot \text{sgn}\left(a_{ij_{\text{max}-1}}\right) \underbrace{\text{max} - 1}_{1 \leq j \leq n} \left(|a_{ij}|\right) \\ \vdots \end{bmatrix}$$

In this case, you would stop the procedure after $m = \left\lfloor \frac{-4 \ln 10}{\ln p} + 1 \right\rfloor$ rows.

Then one uses the `procrustes.permutation.PermutationProcrustes` to match the constructed matrices $\mathbf{A}^0$ and $\mathbf{B}^0$ instead of $\mathbf{A}$ and $\mathbf{B}$. I.e.,

$$\underbrace{\text{max}}_{\left\{\mathbf{P} \middle| \substack{p_{ij} \in \{0,1\} \\ \sum_{i=1}^{n} p_{ij} = \sum_{j=1}^{n} p_{ij} = 1} \right\}} \text{Tr}\left[\mathbf{P}^\dagger \left(\mathbf{A^0}^\dagger \mathbf{B^0}\right)\right]$$

Please note that the "umeyama_approx" might give inaccurate permutation matrix. More specificity, this is a approximated Umeyama method. One example we can give is that when we compute the permutation matrix that transforms $A$ to $B$, the "umeyama_approx" method can not give the exact permutation transformation matrix while "umeyama", "normal1" and "normal2" do.

$$A = \begin{bmatrix} 4 & 5 & -3 & 3 \\ 5 & 7 & 3 & -5 \\ -3 & 3 & 2 & 2 \\ 3 & -5 & 2 & 5 \end{bmatrix}$$

$$B = \begin{bmatrix} 73 & 100 & 73 & -62 \\ 100 & 208 & -116 & 154 \\ 73 & -116 & 154 & 100 \\ -62 & 154 & 100 & 127 \end{bmatrix}$$

## 1.11 procrustes.softassign

The Softassign Procrustes Module.

procrustes.softassign.**softassign**(*a: numpy.ndarray*, *b: numpy.ndarray*, *pad: bool = True*, *translate: bool = False*, *scale: bool = False*, *unpad_col: bool = False*, *unpad_row: bool = False*, *check_finite: bool = True*, *weight: Optional[numpy.ndarray] = None*, *iteration_soft: int = 50*, *iteration_sink: int = 200*, *beta_r: float = 1.1*, *beta_f: float = 100000.0*, *epsilon: float = 0.05*, *epsilon_soft: float = 0.001*, *epsilon_sink: float = 0.001*, *k: float = 0.15*, *gamma_scaler: float = 1.01*, *n_stop: int = 3*, *adapted: bool = True*, *beta_0: Optional[float] = None*, *m_guess: Optional[float] = None*, *iteration_anneal: Optional[int] = None*, *kopt: bool = False*, *kopt_k: int = 3*) → *procrustes.utils.ProcrustesResult*

Find the transformation matrix for 2-sided permutation Procrustes with softassign algorithm.

**Parameters**

- **a** (*ndarray*) – The 2D-array $\mathbf{A}_{m\times n}$ which is going to be transformed.
- **b** (*ndarray*) – The 2D-array $\mathbf{B}_{m\times n}$ representing the reference.
- **pad** (*bool, optional*) – Add zero rows (at the bottom) and/or columns (to the right-hand side) of matrices $\mathbf{A}$ and $\mathbf{B}$ so that they have the same shape.
- **translate** (*bool, optional*) – If True, both arrays are centered at origin (columns of the arrays will have mean zero).
- **scale** (*bool, optional*) – If True, both arrays are normalized with respect to the Frobenius norm, i.e., Tr $\left[\mathbf{A}^\dagger\mathbf{A}\right] = 1$ and Tr $\left[\mathbf{B}^\dagger\mathbf{B}\right] = 1$.
- **unpad_col** (*bool, optional*) – If True, zero columns (with values less than 1.0e-8) on the right-hand side of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.
- **unpad_row** (*bool, optional*) – If True, zero rows (with values less than 1.0e-8) at the bottom of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.
- **check_finite** (*bool, optional*) – If true, convert the input to an array, checking for NaNs or Infs. Default=True.
- **weight** (*ndarray, optional*) – The 1D-array representing the weights of each row of $\mathbf{A}$. This defines the elements of the diagonal matrix $\mathbf{W}$ that is multiplied by $\mathbf{A}$ matrix, i.e., $\mathbf{A} \to \mathbf{WA}$.
- **iteration_soft** (*int, optional*) – Number of iterations for softassign loop.
- **iteration_sink** (*int, optional*) – Number of iterations for Sinkhorn loop.
- **beta_r** (*float, optional*) – Annealing rate which should greater than 1.
- **beta_f** (*float, optional*) – The final inverse temperature.
- **epsilon** (*float, optional*) – The tolerance value for annealing loop.
- **epsilon_soft** (*float, optional*) – The tolerance value used for softassign.
- **epsilon_sink** (*float, optional*) – The tolerance value used for Sinkhorn loop. If adapted version is used, it will use the adapted tolerance value for Sinkhorn instead.
- **k** (*float, optional*) – This parameter controls how much tighter the coverage threshold for the interior loop should be than the coverage threshold for the loops outside. It has be be within the integral $(0, 1)$.

- **gamma_scaler** (*float, optional*) – This parameter ensures the quadratic cost function including self-amplification positive define.
- **n_stop** (*int, optional*) – Number of running steps after the calculation converges in the relaxation procedure.
- **adapted** (*bool, optional*) – If adapted, this function will use the tighter covergence threshold for the interior loops.
- **beta_0** (*float, optional*) – Initial inverse temperature.
- **beta_f** – Final inverse temperature.
- **m_guess** (*ndarray, optional*) – The initial guess of the doubly-stochastic matrix.
- **iteration_anneal** (*int, optional*) – Number of iterations for annealing loop.
- **kopt** (*bool, optional*) – If True, the k_opt heuristic search will be performed.
- **kopt_k** (*int, optional*) – Defines the oder of k-opt heuristic local search. For example, kopt_k=3 leads to a local search of 3 items and kopt_k=2 only searches for two items locally.
- **weight** – The weighting matrix.

**Returns** **res** – The Procrustes result represented as a class:*utils.ProcrustesResult* object.

**Return type** *ProcrustesResult*

### Notes

Quadratic assignment problem (QAP) has played a very special but fundamental role in combinatorial optimization problems. The problem can be defined as a optimization problem to minimize the cost to assign a set of facilities to a set of locations. The cost is a function of the flow between the facilities and the geographical distances among various facilities.

The objective function (also named loss function in machine learning) is defined as

$$E_{qap}(M,\mu,\nu) = -\frac{1}{2}\Sigma_{aibj}C_{ai;bj}M_{ai}M_{bj} + \Sigma_a\mu_a(\Sigma_i M_{ai} - 1)$$
$$+\Sigma_i\nu_i(\Sigma_i M_{ai} - 1) - \frac{\gamma}{2}\Sigma_{ai}{M_{ai}}^2 + \frac{1}{\beta}\Sigma_{ai}M_{ai}\log M_{ai}$$

where $C_{ai,bj}$ is the benefit matrix, $M$ is the desired $N \times N$ permutation matrix. $E$ is the energy function which comes along with a self-amplification term with *gamma*, two Lagrange parameters $\mu$ and $\nu$ for constrained optimization and $M_{ai}\log M_{ai}$ servers as a barrier function which ensures positivity of $M_{ai}$. The inverse temperature $\beta$ is a deterministic annealing control parameter.

### Examples

```
>>> import numpy as np
>>> array_a = np.array([[4, 5, 3, 3], [5, 7, 3, 5],
...                     [3, 3, 2, 2], [3, 5, 2, 5]])
    # define a random matrix
>>> perm = np.array([[0., 0., 1., 0.], [1., 0., 0., 0.],
...                  [0., 0., 0., 1.], [0., 1., 0., 0.]])
    # define b by permuting array_a
>>> b = np.dot(perm.T, np.dot(a, perm))
>>> new_a, new_b, M_ai, error = softassign(a,b,unpad_col=False,unpad_row=False)
```

(continues on next page)

(continued from previous page)

```
>>> M_ai # the permutation matrix
array([[0., 0., 1., 0.],
       [1., 0., 0., 0.],
       [0., 0., 0., 1.],
       [0., 1., 0., 0.]])
>>> error # the error
0.0
```

## 1.12 procrustes.rotational

Rotational-Orthogonal Procrustes Module.

procrustes.rotational.**rotational**(*a: numpy.ndarray*, *b: numpy.ndarray*, *pad: bool = True*, *translate: bool = False*, *scale: bool = False*, *unpad_col: bool = False*, *unpad_row: bool = False*, *check_finite: bool = True*, *weight: Optional[numpy.ndarray] = None*, *lapack_driver: str = 'gesvd'*) → *procrustes.utils.ProcrustesResult*

Perform rotational Procrustes.

Given a matrix $\mathbf{A}_{m\times n}$ and a reference matrix $\mathbf{B}_{m\times n}$, find the rotational transformation matrix $\mathbf{R}_{n\times n}$ that makes $\mathbf{A}$ as close as possible to $\mathbf{B}$. In other words,

$$\underbrace{\min}_{\left\{\mathbf{R}\middle|\substack{\mathbf{R}^{-1}=\mathbf{R}^{\dagger}\\|\mathbf{R}|=1}\right\}} \|\mathbf{AR}-\mathbf{B}\|_F^2$$

This Procrustes method requires the $\mathbf{A}$ and $\mathbf{B}$ matrices to have the same shape, which is gauranteed with the default `pad` argument for any given $\mathbf{A}$ and $\mathbf{B}$ matrices. In preparing the $\mathbf{A}$ and $\mathbf{B}$ matrices, the (optional) order of operations is: **1)** unpad zero rows/columns, **2)** translate the matrices to the origin, **3)** weight entries of $\mathbf{A}$, **4)** scale the matrices to have unit norm, **5)** pad matrices with zero rows/columns so they have the same shape.

**Parameters**

- **a** (*ndarray*) – The 2D-array $\mathbf{A}$ which is going to be transformed.
- **b** (*ndarray*) – The 2D-array $\mathbf{B}$ representing the reference matrix.
- **pad** (*bool, optional*) – Add zero rows (at the bottom) and/or columns (to the right-hand side) of matrices $\mathbf{A}$ and $\mathbf{B}$ so that they have the same shape.
- **translate** (*bool, optional*) – If True, both arrays are centered at origin (columns of the arrays will have mean zero).
- **scale** (*bool, optional*) – If True, both arrays are normalized with respect to the Frobenius norm, i.e., Tr $\left[\mathbf{A}^{\dagger}\mathbf{A}\right] = 1$ and Tr $\left[\mathbf{B}^{\dagger}\mathbf{B}\right] = 1$.
- **unpad_col** (*bool, optional*) – If True, zero columns (with values less than 1.0e-8) on the right-hand side of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.
- **unpad_row** (*bool, optional*) – If True, zero rows (with values less than 1.0e-8) at the bottom of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.
- **check_finite** (*bool, optional*) – If True, convert the input to an array, checking for NaNs or Infs.
- **weight** (*ndarray, optional*) – The 1D-array representing the weights of each row of $\mathbf{A}$. This defines the elements of the diagonal matrix $\mathbf{W}$ that is multiplied by $\mathbf{A}$ matrix, i.e., $\mathbf{A} \rightarrow \mathbf{WA}$.

- **lapack_driver** (*{'gesvd', 'gesdd'}, optional*) – Whether to use the more efficient divide-and-conquer approach ('gesdd') or the more robust general rectangular approach ('gesvd') to compute the singular-value decomposition with *scipy.linalg.svd*.

**Returns** **res** – The Procrustes result represented as a class:*utils.ProcrustesResult* object.

**Return type** *ProcrustesResult*

### Notes

The optimal rotational matrix is obtained by,

$$\mathbf{R}_{\text{opt}} = \arg \underbrace{\min}_{\left\{\mathbf{R}\middle|\substack{\mathbf{R}^{-1}=\mathbf{R}^{\dagger}\\|\mathbf{R}|=1}\right\}} \|\mathbf{AR}-\mathbf{B}\|_F^2 = \arg \underbrace{\max}_{\left\{\mathbf{R}\middle|\substack{\mathbf{R}^{-1}=\mathbf{R}^{\dagger}\\|\mathbf{R}|=1}\right\}} \text{Tr}\left[\mathbf{R}^{\dagger}\mathbf{A}^{\dagger}\mathbf{B}\right]$$

The solution is obtained by taking the singular value decomposition (SVD) of the $\mathbf{A}^{\dagger}\mathbf{B}$ matrix,

$$\mathbf{A}^{\dagger}\mathbf{B} = \tilde{\mathbf{U}}\tilde{\mathbf{\Sigma}}\tilde{\mathbf{V}}^{\dagger}$$
$$\mathbf{R}_{\text{opt}} = \tilde{\mathbf{U}}\tilde{\mathbf{S}}\tilde{\mathbf{V}}^{\dagger}$$

where $\tilde{\mathbf{S}}_{n\times m}$ is almost an identity matrix,

$$\tilde{\mathbf{S}}_{m\times n} \equiv \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \ddots & \vdots & 0 \\ 0 & \ddots & \ddots & 0 & \vdots \\ \vdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0\cdots & 0 & \text{sgn}\left(\left|\mathbf{U}\mathbf{V}^{\dagger}\right|\right) \end{bmatrix}$$

in which the smallest singular value is replaced by

$$\text{sgn}\left(\left|\tilde{\mathbf{U}}\tilde{\mathbf{V}}^{\dagger}\right|\right) = \begin{cases} +1 & \left|\tilde{\mathbf{U}}\tilde{\mathbf{V}}^{\dagger}\right| \geq 0 \\ -1 & \left|\tilde{\mathbf{U}}\tilde{\mathbf{V}}^{\dagger}\right| < 0 \end{cases}$$

### Examples

```
>>> import numpy as np
>>> array_a = np.array([[1.5, 7.4], [8.5, 4.5]])
>>> array_b = np.array([[6.29325035,  4.17193001, 0., 0,],
...                     [9.19238816, -2.82842712, 0., 0.],
...                     [0.,          0.,         0., 0.]])
>>> res = rotational(array_a,array_b,translate=False,scale=False)
>>> res.t   # rotational transformation
array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]])
>>> res.error   # one-sided Procrustes error
1.483808210011695e-17
```

## 1.13 procrustes.symmetric

Symmetric Procrustes Module.

procrustes.symmetric.**symmetric**(*a: numpy.ndarray*, *b: numpy.ndarray*, *pad: bool = True*, *translate: bool = False*, *scale: bool = False*, *unpad_col: bool = False*, *unpad_row: bool = False*, *check_finite: bool = True*, *weight: Optional[numpy.ndarray] = None*, *lapack_driver: str = 'gesvd'*) → *procrustes.utils.ProcrustesResult*

Perform symmetric Procrustes.

Given a matrix $\mathbf{A}_{m \times n}$ and a reference matrix $\mathbf{B}_{m \times n}$ with $m \geqslant n$, find the symmetrix transformation matrix $\mathbf{X}_{n \times n}$ that makes $\mathbf{AX}$ as close as possible to $\mathbf{B}$. In other words,

$$\underbrace{\min}_{\{\mathbf{X} | \mathbf{X}=\mathbf{X}^{\dagger}\}} \|\mathbf{AX} - \mathbf{B}\|_F^2$$

This Procrustes method requires the $\mathbf{A}$ and $\mathbf{B}$ matrices to have the same shape with $m \geqslant n$, which is guaranteed with the default `pad` argument for any given $\mathbf{A}$ and $\mathbf{B}$ matrices. In preparing the $\mathbf{A}$ and $\mathbf{B}$ matrices, the (optional) order of operations is: **1)** unpad zero rows/columns, **2)** translate the matrices to the origin, **3)** weight entries of $\mathbf{A}$, **4)** scale the matrices to have unit norm, **5)** pad matrices with zero rows/columns so they have the same shape.

**Parameters**

- **a** (*ndarray*) – The 2D-array $\mathbf{A}$ which is going to be transformed.
- **b** (*ndarray*) – The 2D-array $\mathbf{B}$ representing the reference matrix.
- **pad** (*bool, optional*) – Add zero rows (at the bottom) and/or columns (to the right-hand side) of matrices $\mathbf{A}$ and $\mathbf{B}$ so that they have the same shape.
- **translate** (*bool, optional*) – If True, both arrays are centered at origin (columns of the arrays will have mean zero).
- **scale** (*bool, optional*) – If True, both arrays are normalized with respect to the Frobenius norm, i.e., $\text{Tr}\left[\mathbf{A}^{\dagger}\mathbf{A}\right] = 1$ and $\text{Tr}\left[\mathbf{B}^{\dagger}\mathbf{B}\right] = 1$.
- **unpad_col** (*bool, optional*) – If True, zero columns (with values less than 1.0e-8) on the right-hand side of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.
- **unpad_row** (*bool, optional*) – If True, zero rows (with values less than 1.0e-8) at the bottom of the intial $\mathbf{A}$ and $\mathbf{B}$ matrices are removed.
- **check_finite** (*bool, optional*) – If True, convert the input to an array, checking for NaNs or Infs.
- **weight** (*ndarray, optional*) – The 1D-array representing the weights of each row of $\mathbf{A}$. This defines the elements of the diagonal matrix $\mathbf{W}$ that is multiplied by $\mathbf{A}$ matrix, i.e., $\mathbf{A} \rightarrow \mathbf{WA}$.
- **lapack_driver** (*{'gesvd', 'gesdd'}, optional*) – Whether to use the more efficient divide-and-conquer approach ('gesdd') or the more robust general rectangular approach ('gesvd') to compute the singular-value decomposition with *scipy.linalg.svd*.

**Returns** **res** – The Procrustes result represented as a class:*utils.ProcrustesResult* object.

**Return type** *ProcrustesResult*

### Notes

The optimal symmetrix matrix is obtained by,

$$\mathbf{X}_{\text{opt}} = \arg \underbrace{\min}_{\{\mathbf{X}|\mathbf{X}=\mathbf{X}^\dagger\}} \|\mathbf{AX}-\mathbf{B}\|_F^2 = \underbrace{\min}_{\{\mathbf{X}|\mathbf{X}=\mathbf{X}^\dagger\}} \text{Tr}\left[(\mathbf{AX}-\mathbf{B})^\dagger(\mathbf{AX}-\mathbf{B})\right]$$

Considering the singular value decomposition of $\mathbf{A}$,

$$\mathbf{A}_{m\times n} = \mathbf{U}_{m\times m}\mathbf{\Sigma}_{m\times n}\mathbf{V}^\dagger_{n\times n}$$

where $\mathbf{\Sigma}_{m\times n}$ is a rectangular diagonal matrix with non-negative singular values $\sigma_i = [\mathbf{\Sigma}]_{ii}$ listed in descending order, define

$$\mathbf{C}_{m\times n} = \mathbf{U}^\dagger_{m\times m}\mathbf{B}_{m\times n}\mathbf{V}_{n\times n}$$

with elements denoted by $c_{ij}$. Then we compute the symmetric matrix $\mathbf{Y}_{n\times n}$ with

$$[\mathbf{Y}]_{ij} = \begin{cases} 0 \\ i \text{ and } j > \text{rank}\,(\mathbf{A}) \\ \frac{\sigma_i c_{ij}+\sigma_j c_{ji}}{\sigma_i^2+\sigma_j^2} \\ \text{otherwise} \end{cases}$$

It is worth noting that the first part of this definition only applies in the unusual case where $\mathbf{A}$ has rank less than $n$. The $\mathbf{X}_{\text{opt}}$ is given by

$$\mathbf{X}_{\text{opt}} = \mathbf{VYV}^\dagger$$

### Examples

```
>>> import numpy as np
>>> a = np.array([[5., 2., 8.],
...               [2., 2., 3.],
...               [1., 5., 6.],
...               [7., 3., 2.]])
>>> b = np.array([[ 52284.5, 209138. , 470560.5],
...               [ 22788.5,  91154. , 205096.5],
...               [ 46139.5, 184558. , 415255.5],
...               [ 22788.5,  91154. , 205096.5]])
>>> res = symmetric(a, b, pad=True, translate=True, scale=True)
>>> res.t   # symmetric transformation array
array([[0.0166352 , 0.06654081, 0.14971682],
       [0.06654081, 0.26616324, 0.59886729],
       [0.14971682, 0.59886729, 1.34745141]])
>>> res.error   # error
4.483083428047388e-31
```

CHAPTER

TWO

# INDICES AND TABLES

- genindex
- modindex
- search

# BIBLIOGRAPHY


[1] John C. Gower. Procrustes methods. *WIREs Computational Statistics*, 2(4):503–508, 2010. doi:10.1002/wics.107.

[2] Peter H. Schönemann. A generalized solution of the orthogonal procrustes problem. *Psychometrika*, 31(1):1–10, Mar 1966. doi:10.1007/BF02289451.

[3] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1330–1334, Nov 2000. doi:10.1109/34.888718.

[4] John C Gower and Garmt B Dijksterhuis. *Procrustes Problems*. Volume 30. Oxford University Press, 2004. doi:10.1093/acprof:oso/9780198510581.001.0001.

[5] Frank B Brokken. Orthogonal procrustes rotation maximizing congruence. *Psychometrika*, 48(3):343–352, 1983. doi:10.1007/BF02293679.

[6] JL Farrell, JC Stuelpnagel, RH Wessner, JR Velman, and JE Brook. A least squares estimate of satellite attitude (grace wahba). *SIAM Review*, 8(3):384–386, 1966. doi:10.1137/1008080.

[7] Nicholas J Higham. The symmetric procrustes problem. *BIT Numerical Mathematics*, 28(1):133–143, 1988. doi:10.1007/BF01934701.

[8] René Escalante and Marcos Raydan. Dykstra's algorithm for constrained least-squares rectangular matrix problems. *Computers & Mathematics with Applications*, 35(6):73–79, 1998. doi:10.1016/S0898-1221(98)00020-0.

[9] Juan Peng, Xi-Yan Hu, and Lei Zhang. The (m, n)-symmetric procrustes problem. *Applied Mathematics and Computation*, 198(1):24–34, 2008. doi:10.1016/j.amc.2007.08.094.

[10] Farnaz Heidar Zadeh and Paul W Ayers. Molecular alignment as a penalized permutation procrustes problem. *Journal of Mathematical Chemistry*, 51(3):927–936, 2013. doi:10.1007/s10910-012-0119-2.

[11] Peter H Schönemann. On two-sided orthogonal procrustes problems. *Psychometrika*, 33(1):19–33, 1968. doi:10.1007/BF02289673.

[12] Shinji Umeyama. An eigendecomposition approach to weighted graph matching problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(5):695–703, 1988. doi:10.1109/34.6778.

[13] Pythagoras Papadimitriou. *Parallel Solution of SVD-Related Problems, with Applications*. PhD thesis, University of Manchester, 1993. URL: http://vummath.ma.man.ac.uk/~higham/links/theses/papad93.pdf.

[14] Chris Ding, Tao Li, and Michael I Jordan. Nonnegative matrix factorization for combinatorial optimization: spectral clustering, graph matching, and clique finding. In *ICDM '08: Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, 183–192. IEEE, 2008. doi:10.1109/ICDM.2008.130.

[15] Mikkel B Stegmann and David Delgado Gomez. A brief introduction to statistical shape analysis. *Informatics and Mathematical Modelling, Technical University of Denmark, DTU*, 2002. URL: http://www2.compute.dtu.dk/pubdb/pubs/403-full.html.

[16] John C Gower. Generalized procrustes analysis. *Psychometrika*, 40(1):33–51, 1975. doi:10.1007/BF02291478.

[17] Jos MF Ten Berge. Orthogonal procrustes rotation for two or more matrices. *Psychometrika*, 42(2):267–276, 1977. doi:10.1007/BF02294053.

[18] Ingwer Borg and Patrick JF Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer Science & Business Media, 2005. doi:10.1007/0-387-28981-X.

[19] JJ Kosowsky and Alan L Yuille. The invisible hand algorithm: solving the assignment problem with statistical physics. *Neural Networks*, 7(3):477–490, 1994. doi:10.1016/0893-6080(94)90081-7.

[20] Steven Gold and Anand Rangarajan. Softassign versus softmax: benchmarks in combinatorial optimization. In *Advances in Neural Information Processing Systems*, 626–632. 1996. doi:10.5555/2998828.2998917.

[21] Anand Rangarajan, Alan L Yuille, Steven Gold, and Eric Mjolsness. A convergence proof for the softassign quadratic assignment algorithm. In *Advances in Neural Information Processing Systems*, 620–626. 1997. doi:10.5555/2998981.2999069.

[22] Jiahui Wang, Xiaoshuang Zeng, Wenjie Luo, and Wei An. The application of neural network in multiple object tracking. *DEStech Transactions on Computer Science and Engineering*, pages 358–264, 2018. doi:10.12783/dtcse/csse2018/24504.

[23] Steven Gold, Anand Rangarajan, and others. Softmax to softassign: neural network algorithms for combinatorial optimization. *Journal of Artificial Neural Networks*, 2(4):381–399, 1996. doi:10.5555/235912.235919.

[24] Yu Tian, Junchi Yan, Hequan Zhang, Ya Zhang, Xiaokang Yang, and Hongyuan Zha. On the convergence of graph matching: graduated assignment revisited. In *European Conference on Computer Vision*, 821–835. Springer, 2012. doi:10.1007/978-3-642-33712-3_59.

[25] Z Sheikhbahaee, R Nakajima, T Erben, P Schneider, H Hildebrandt, and AC Becker. Photometric calibration of the combo-17 survey with the softassign procrustes matching method. *Monthly Notices of the Royal Astronomical Society*, 471(3):3443–3455, 2017. doi:10.1093/mnras/stx1810.

[26] Alan L Yuille and JJ Kosowsky. Statistical physics algorithms that converge. *Neural Computation*, 6(3):341–356, 1994. doi:10.1162/neco.1994.6.3.341.

# PYTHON MODULE INDEX

## p

# INDEX