

---

# **triarray Documentation**

***Release 0.2.1***

**Jared Lumpe**

**Sep 20, 2021**



---

## Contents

---

<b>1</b>	<b>Example</b>
----------	----------------

<b>3</b>
----------



**triarray** is a Python package for working with symmetric matrices in non- redundant format. This format stores only the elements in the upper or lower triangle, thus halving memory requirements.

When storing symmetric matrices in standard array format about half of the elements are redundant, meaning you are using twice as much memory or disk space as you need to. This is especially common in scientific applications when working with large distance or similarity matrices.

Space can be saved by storing only the lower or upper triangle of the array, but standard operations like getting an element by row and column become awkward. **triarray** provides tools for working with data in this format.

**triarray** uses [Numba](#) 's just-in-time compilation to generate high-performance C code that works with any data type and is easily extendable (including within a Jupyter notebook).



# CHAPTER 1

---

## Example

---

The `scipy.spatial.distance.pdist()` function calculates pairwise distances between all rows of a matrix and returns only the upper triangle of the full distance matrix:

```
import numpy as np
from scipy.spatial.distance import pdist

vectors = np.random.rand(1000, 10)

dists = pdist(vectors)  # Shape is (499500,) instead of (1000, 1000)
```

The `TriMatrix` class wraps a 1D Numpy array storing the condensed data and exposes an interface that lets you treat it as if it was still in matrix format:

```
from triarray import TriMatrix

matrix = TriMatrix(dists, upper=True, diag_val=0)

matrix.size  # Number of rows/columns in matrix
>>> 1000

matrix[0, 1]  # Distance between 0th and 1st vector
>>> 1.1610289956390953

matrix[0, 0]  # Diagonals are zero
>>> 0.0

matrix[0]  # 0th row of matrix
>>> array([ 0.          ,  1.161029   ,  1.03467554,  1.32559121,  1.26185034,
          ...
```

It even supports Numpy's [advanced indexing](#) with integer arrays of arbitrary shape:

```
rows, cols = np.ix_([0, 1, 2], [3, 4, 5])
rows, cols
```

(continues on next page)

(continued from previous page)

```
>>> (array([[0],
           [1],
           [3]]), array([[4, 5, 6]]))

matrix[rows, cols]
>>> array([[ 1.26185034,  1.08800206,  1.30490993],
           [ 0.99262394,  1.33044029,  1.20373382],
           [ 1.42524039,  1.36195143,  1.70404005]])
```

## 1.1 Documentation contents

### 1.1.1 Basic Usage

TODO

## 1.2 API

### 1.2.1 Python API

#### Indexing

Tools for converting between 2D indices of full matrices and 1D indices of condensed arrays.

#### Array conversion

Convert full 2D matrices to and from condensed triangular format.

#### I/O

Read and write full matrices to and from disk in condensed format.

#### Matrix interface

### 1.2.2 Numba API

Use these functions if you wish to extend **triarray** using Numba.

This package makes heavy use of [Numba](#) to compile Python functions into high-performance C code. Many of these compiled functions are hidden behind Python functions to expose a more friendly API, but I chose Numba over Cython for this purpose in part because it is much easier to extend existing code (especially in the Jupyter Notebook).

If you plan on using this package in your own Numba code it will greatly improve performance to use the Numba compiled functions directly.

TODO

Fill this out. . .



## 1.3 Indices and tables

- [genindex](#)
- [search](#)