Journal of Statistical Software

MMMMMM YYYY, Volume VV, Code Snippet II. doi: 10.18637/jss.v000.i00

PHT: a Python Package for Computing Persistent Homology Transform to Metricise Shapes

Hanyu Song
Duke University

Azeem Zaman
Duke University

Abstract

In this paper we present a package implementing the results of the paper by Turner et. al (2014). This package is mainly intended for computing the persistent homology transform (PHT) of objects in $\mathbb{R}^3$ and shapes in $\mathbb{R}^2$. PHT is a statistic computed from a collection of persistence diagrams, which (according to a result in the paper) completely describes a shape or surface using topological properties. With the PHT we are able to construct a metric on the space of piecewise linear shapes. This metric opens up many different possibilities for statistical analysis, such as clustering shapes.

Keywords: persistent homology transform, Python.

1. Background

Turner et al.(2014)'s paper introduces a tool that can be used to perform statistical shape analysis on objects in $\mathbb{R}^3$ and shapes in $\mathbb{R}^2$. The result can be of interest to topological data analysists, researchers modeling shapes (such as medical imaging) and morphologists. One of the paper authors implements the result to compute the distances between heel bones in primates and generate a tree, which can be compared with a tree generated from the genetic distances between primate species.

The paper presents an approach to obtain a representation of a shape that can be used in statistical models. In particular, the paper shows that persistence diagrams, which are explained in detail below, are sufficient statistics for shape and surface models. This is an important result because it demonstrates that persistent homology, an essential tool in topological data analysis (TDA), preserves information.

2. Code Overview

Here is a list of functions contained in the package:

1. Functions to read in files containing the data
2. A function to construct a persistence diagram given a direction
3. Functions to calculate the distance between persistence diagrams
4. Functions to generate directions for the construction of persistence diagrams
5. Functions to caluclate the distance between objects
6. Functions to generate diagrams for many objects given a set of directions
7. Main function to be called by user

The following packages are required for implementation: **math**, **multiprocessing**, **NumPy**, **SciPy**, **glob** and **Numba**.

2.1. Functions for reading shapes

Two functions for reading in data are included in the package. The first `read_file` is for reading in text files saved with raw shape data; the second `read_closed_shape` is used to read MATLAB `.mat` files saved with closed shape data. Note that each file contains data of only one shape. Both functions can read all relevant files in a specified directory; both return a list of vertices and edges of each shape, with the vertices and edges saved in two separate `numpy.ndarray`'s.

2.2. Functions for construction of persistence diagrams

A function to construct a persistence diagram given a direction is included in the package. Below are the detailed explanations of persistence diagrams.

A persistence diagram is a filtration. We start with an object and "build" the object in a certain direction. We record when each point in the object first appears (is "born") and when it merges into an object that already exists (it "dies"). Consider the shape given in Figure 1. We will construct this figure in the direction $v = (0, 1)$. If we imagine moving upwards across the figure, the first height at which we will see any points of the diagram is $h = -1$. We see that vertices 1 and 7 are born at $h = -1$, as shown in Figure 2. The next height at which something something interesting happens is $h = 0$, at which time three more points are born. All of these points, however, die immediately. Vertex 2 merges with vertex 1 and vertices 4 and 5 merge with vertex 7. At this time we have two unconnected components. This is shown in Figure 3. Once we reach $h = 1$, we have finished constructing the diagram. Vertex 6 dies immediately because it merges with vertex 5. Vertex 3 joints the two components that were previously disjoint. Since vertices 1 and 7 were both born at $h = -1$, we make a convention that lower numbered vertices will be considered the root and higher numbered vertices will merge with them. Thus at time $h = 1$ vertex 7 dies, as it merges with vertex 1. A plot of the persistence diagram is given in Figure 5. The red point represents vertex 1, which never

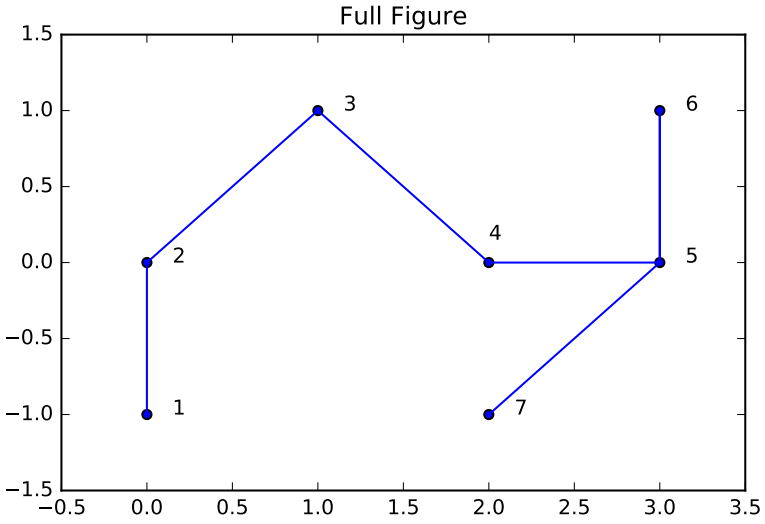


Figure 1: A shape that we are interested in modeling.

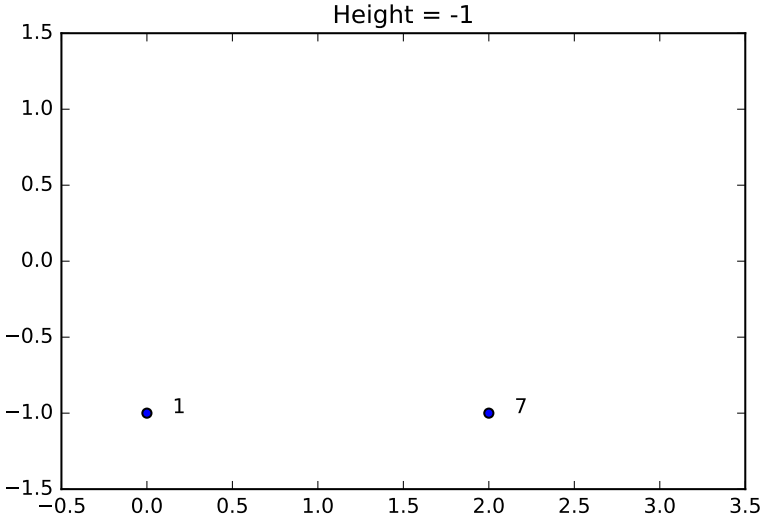


Figure 2: The figure at height $h = -1$.

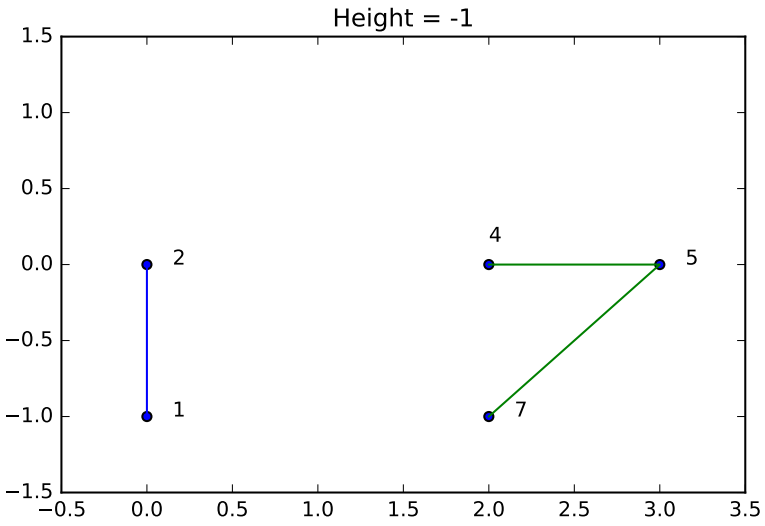


Figure 3: The persistence diagram in the direction v at time $h = 0$.

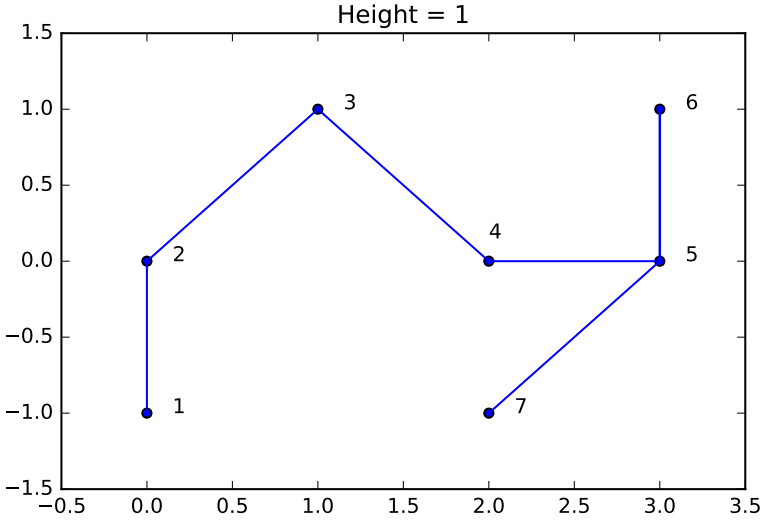


Figure 4: The figure at $h = 1$.

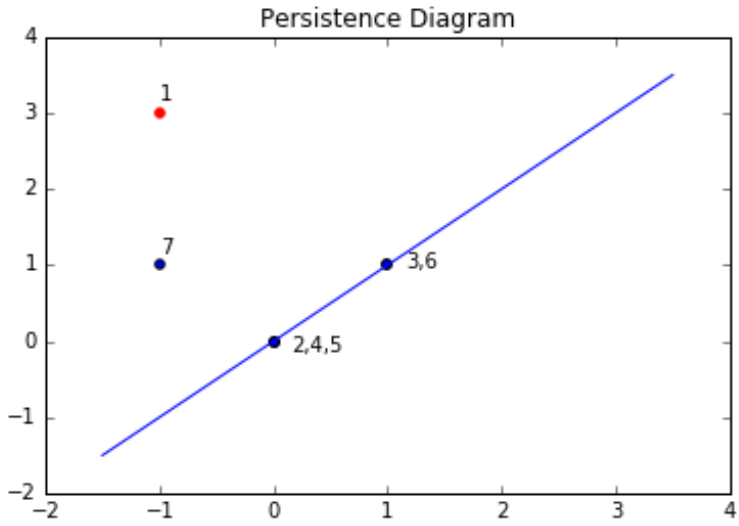


Figure 5: The persistence diagram for our shape.

dies. We consider the point to be at $(-1, \infty)$. The line in the figure is the diagonal. A point on the diagonal is one that is born and dies at the same instant.

To implement this process of constructing persistence diagrams, we need to be able to tell when a vertex is born and when the vertex merges with another component. For a fixed direction v (which we will always take to be unit length), we can easily determine the height of a vertex. The height of a vertex p with respect to a direction v is simply the inner product of these two vectors. The harder part of the algorithm is determining when components merge. To keep track of this, we will use a modified version of the Union-Find algorithm. We define a class called `Tree`. A `Tree` object has a parent and every tree begins with its own parent. For convenience, it also has a name. We will name a tree after a vertex. So for each vertex i there will be a tree such that `Tree.name = i`.

The function `Find` finds the root of a tree by traveling up the tree until we find a tree that is equal to its parent. The function `height_Union` joins two trees in a specific way. If the two input trees have the same root, nothing is done. If the two trees have different heights, we make the root of one tree equal to the root of the other tree, effectively merging the trees. As persistence diagrams are built upwards, we make the root of the joint tree the vertex that has the smaller height. If the vertices have the same height, we make the root the vertex with the lower number. To increase the efficiency of the search, we update the root of each tree with each search. This reduces the cost of using the recursive `Find` function.

These functions are used internally and will not be called by a user.

```
class Tree:
def __init__(self, name):
self.parent = self
self.name = name
self.rank = 0
```

```
def Find(x):
"""
This function determines the root of the tree
that x is in.  It works recursively. x should
be an object of class Tree.
"""
if x.parent != x:
x.parent = Find(x.parent)
return x.parent

def height_Union(x, y, dict_heights):
"""
This function takes the union of two nodes.
It does this by changing the root one tree
to be the root of the other tree.  It changes the
root based on height.  The root becomes the node with
the lowest height.  So the node that is born first
becomes the root.

If the two roots have the same height, the lowest
number becomes the root.  For example, if we have vertex
1 and vertix 3 at the same height, vertex 1 will become
the root.

Inputs:
x,y:  objects of Tree class
dict_heights:  a map v-> h, where v is a vertex,
which should be the .name of some tree and h is
the height with respect to some direction

    Returns:
        No returns.  Tree objects are merged.
"""
x_root = Find(x)
y_root = Find(y)
if x_root == y_root:
return None
if dict_heights[x_root.name] < dict_heights[y_root.name]:
y_root.parent = x_root
elif dict_heights[x_root.name] == dict_heights[y_root.name]:
if x.name < y.name:
y_root.parent = x_root
else:
x_root.parent = y_root
else:
x_root.parent = y_root
```

3. Distance between persistence diagrams

Once we have a diagram, we can consider the distance between diagrams. If X and Y are diagrams, we can consider bijections ϕ between the points and copies of the diagonal in X with the points and copies of the diagonal in Y. Bijections always exist because we can add enough copies of the diagonal to create one. We define the distance between X and Y to be

$$dist_p(X, Y) = \left(\inf_{\phi: X \to Y} \sum_{x \in X} ||x - \phi(x)||_p^p \right)^{1/p}. \tag{1}$$

The paper suggests that $p = 1$ performs best in practice (the L_1 norm). The existence of an optimal (but not necessarily unique) bijection is proved in Turner (2013). In practice, we see that we can add the projection onto the diagonal for each point. This is an optimisation problem that can be solved using the Hungarian (Munkres) algorithm. We can calculate the distance between the points in X and the points in Y and the distance from the points in X to the diagonal. This can be represented as a cost matrix, where the ij entry is the L_1 distance between point i in X and point j in Y. The original algorithm is $O(n^4)$, but it was later improved to $O(n^3)$ (see Kuhn (1955), Kuhn (1956), Munkres (1957)).

As we will discuss in detail later, this is the bottleneck in our code. The algorithm is somewhat difficult to implement, so we used an existing implementation. Some of the existing and publicly available implementations of this algorithm suffer from errors that cause the code to become stuck in an infinite loop. We settled on using a function from `scipy.optimize`, `linear_sum_assignment`. This is the distance between the finite points. We also need to consider the points with one component at infinity. We will pair these amongst themselves and simply consider the distance between the birth times. If two diagrams have a different number of points at infinity, then we would have to pair a point at infinity with a finite point or a point on the diagonal, which is infinitely far away. The distance between such diagrams is said to be infinite.

4. Distance between objects

Now that we can calculate the distance between two persistence diagrams, we are ready to compute the distance between two objects in $\mathbb{R}^2$ or $\mathbb{R}^3$. For the objects we are considering, the distance (which is indeed a distance based on the results in the paper) is

$$dist(M_1, M_2) := \int_{S^{d-1}} dist(X(M_1, v), X(M_2, v))\, dv, \tag{2}$$

where $X(M_i, v)$ is the persistence diagram for object M_i in direction v. Stability results from Turner, Mukherjee, and Boyer (2014) reassure that the error in approximating the distance using only a finite sampling of directions should be small. Thus we approximate this integral by using a sample of points from the circle (in $\mathbb{R}^2$) or the sphere (in $\mathbb{R}^3$), i.e. we evaluate the integral using either Monte Carlo integration (if we randomly select directions) or numeric integration (if we select evenly spaced vectors). It will generally be preferable to select evenly spaced vectors to numerically approximate the integral, as is noted in the next paragraph.

If is often useful to consider objects modulo a group of transformations, typically scaling, rotation, and translation. For instance, if one object is simply a rotation of another object,

Table 1: Distance Matrix Generated by PHT

Shape	Square 1	Square 2	Triangle 1	Triangle 2
Square 1	0	0.00215	0.00530	0.00523
Square 2	0.00215	0	0.00530	0.00523
Triangle 1	0.00530	0.00530	0	0.00144
Triangle 2	0.00523	0.00523	0.00144	0

we would like the distance between the objects to be zero. Descriptions of how to implement these transformations can be found in the Turner *et al.* (2014). The code that centers and scales the objects can be found in the package. The implementation of rotation is slightly different. To calculate the distance modulo rotation, we must consider all possible rotations of the second object and select the one that minimises the distance. In practice, we select some finite set of rotations to consider. Furthermore, we do not actually rotate the object. Instead, we note that the persistence diagram of a rotated object is the persistence diagram of a rotated direction. Thus if we pick evenly spaced vectors, we simply need to relabel persistence diagrams to rotate objects.

5. Algorithms

One algorithm used is the Hungarian (or Munkres) algorithm. The algorithm is used in situations where assignments with an associated cost must be made and the goal is to select the assignment to minimise the cost. Here we use this algorithm to calculate the distance between persistence diagrams. The distance between persistence diagrams is the sum of the distances between the points of the first persistence diagram paired with the points of the second diagram and additional points on the diagonal. Selecting the pairing that minimises this distance can be achieved using the Munkres algorithm.

Another algorithm used in our code is the Union-Find algorithm. This algorithm is used in the construction of the persistence diagrams. During the construction we must keep track of when disjoint components merge. We view each component as a tree. When two components merge we join the roots of the trees. This allows us to find when disjoint components merge.

6. Tests and Examples

6.1. Test on simulated data

In this section we test the functions on simple shapes, two squares and two triangles. The shapes are similar, but they are slightly different in size and rotation. We will use the scaling distance function to test whether we can cluster these shapes. The simpler shapes are shown below.

Our code generates the following distance matrix:

We see that the distance between the triangles (the first two shapes) is less than the distance between the squares. In fact, the distance between the triangle and the two squares is almost identical. Similarly, the distance between the squares and both triangles are almost equal.

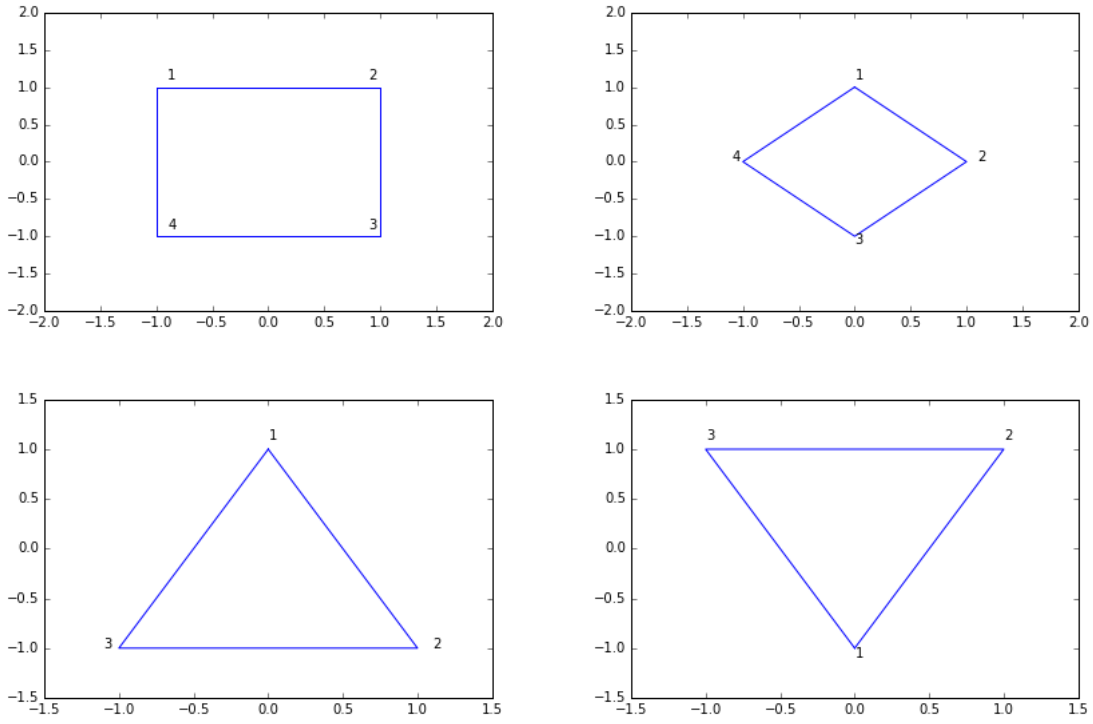


Figure 6:

The distances here are so distinct that we can successively cluster the objects using simple hierarchical clustering.

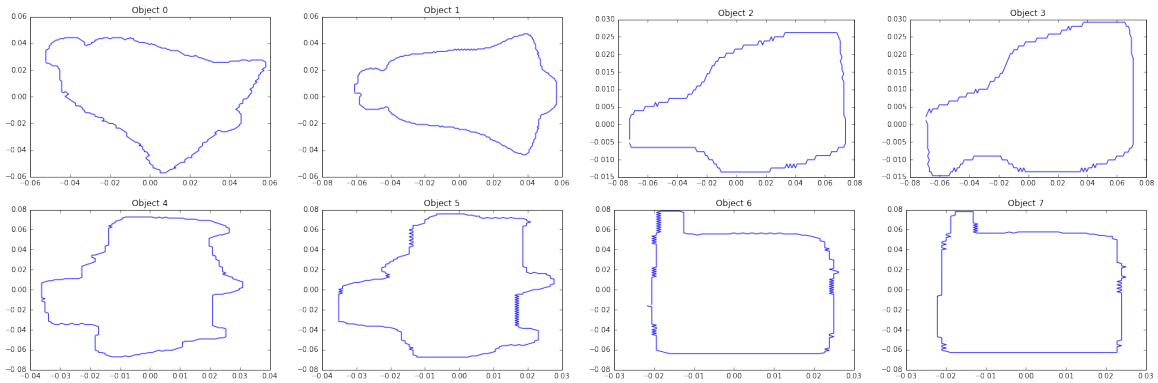
7. Test on real data

We will now test the code on real data from the MPEG-7 data set. These are images that have been converted to outlines. They are stored as `.mat` files, we will read them in using the function we wrote for that purpose. The objects are displayed below. We are testing the code on two objects from each of four different classes.

8. Optimization

8.1. Code Optimization

We tried optimising the performance of the codes using `numba` just-in-time (JIT) compilation, `Cython`, embarrassingly parallel processing. Based on the input file `Class1_Sample1.mat`, most functions perform very well, taking only milliseconds to process such a shape with 375 vertices. Assuming we want to measure the distance between such a shape to another shape with 375 vertices, it takes 33.6s for the current Munkres algorithm to finish running, suggesting room for improvement.


Object 0
Object 1
Object 2
Object 3
Object 4
Object 5
Object 6
Object 7

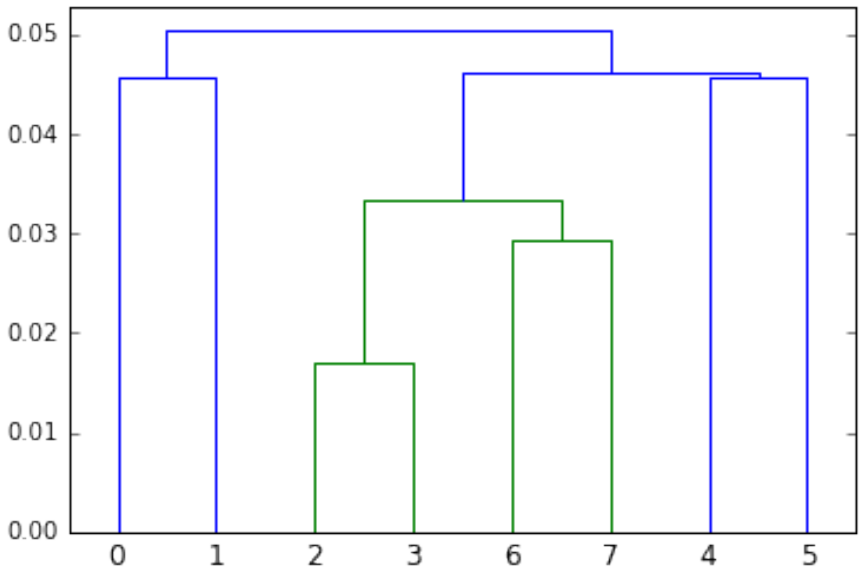

0.05
0.04
0.03
0.02
0.01
0.00
0
1
2
3
6
7
4
5

Figure 7:

Table 2: Function Speeds

Code	Original Function	numba JIT	Cython	Cython with static typing and `cNumPy`
Wall Time	1 min 41s	1 min 27s	1 min 46s	1 min 42s

We used the function `linear_sum_assignment` from `scipy.optimize`. To improve its performance, we tried numba JIT computation, simple compilation in `Cython`, cythonizing via static typing and `cnumpy` iteration and wrapping `C` codes. However, no correct and easily accessible `C` or `C++` functions have been found. Neither have we achieve significant improvement in `Python`. Specifically, the major problem in cythonizing concerns the fast array declarations in `cdef` classes. Fast array declarations, however, are currently not accessible in the fields of cdef classes or as global variables, according to Cython documentation.

We benchmarked the different optimisation strategies based on a $1,000 \times 1,000$ cost matrix. The performances are summarized in the table below:

8.2. Algorithm Optimisation

We sped up the function `make_diagram` via better algorithm. When constructing a `diagram`, we exclude points that are born and die at the same time,i.e. points that fall on the diagonal (as shown below), which largely reduces the size of the cost matrix, thereby leading to a significant speed-up of `linear_sum_assignment`, the function we discussed above.

The chunk that helps speed up the function is shown below:

```
for v in dict_heights.keys():
        # if time of death is none, add a point at infinity
        if dict_deaths[v] == None:
            diag.addinfpt(dict_heights[v])
        else:
            # only add point if not on the diagonal
            # this is designed to reduce the complexity
            # of the problem for the Munkres algorithm
            if dict_heights[v] != dict_deaths[v]:
                diag.addpt((dict_heights[v], dict_deaths[v]))
```

test Turner *et al.* (2014)

References

Kuhn HW (1955). "The Hungarian Method for the assignment problem." *Naval Research Logistics Quarterly*, **2**, 83–97.

Kuhn HW (1956). "Variants of the Hungarian method for assignment problems." *Naval Research Logistics Quarterly*, **3**, 253–258.

Munkres J (1957). "Algorithms for the Assignment and Transportation Problems." *Journal of the Society for Industrial and Applied Mathematics*, **5**, 32–38.

Turner K (2013). "Means and medians of sets of persistence diagrams." *ArXiv e-prints*.

Turner K, Mukherjee S, Boyer DM (2014). "Persistent Homology Transform for Modelling Shapes and Surfaces." *Information and Inference: A Journal of the IMA*, **3**, 310–344. doi:10.1093/imaiai/iau011.

Affiliation:

Hanyu Song, Azeem Zaman
Department of Statistical Science
Duke University
Durham, NC 27708-0251
E-mail: hanyu.song@duke.edu, azeem.zaman@duke.edu
.

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Code Snippet II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd