GriSPy: A Python package for Fixed-Radius Nearest Neighbors Search

Martin Chalela\textsuperscript{a,b,c}, Emanuel Sillero\textsuperscript{a,b,c}, Luis Pereyra\textsuperscript{a,b,c}, Mario Alejandro García\textsuperscript{d}, Juan B. Cabral\textsuperscript{e,a}, Marcelo Lares\textsuperscript{a}, Manuel Merchán\textsuperscript{e,b}

\textsuperscript{a} Instituto De Astronomía Teórica Y Experimental - Observatorio Astronómico Córdoba (IATE, UNC–CONICET), Córdoba, Argentina.
\textsuperscript{b} Observatorio Astronómico de Córdoba, Universidad Nacional de Córdoba, Laprida 854, X5000BGR, Córdoba, Argentina.
\textsuperscript{c} Facultad de Matemática, Astronomía y Física Universidad Nacional de Córdoba (FaMAF–UNC) Blvd. Medina Allende s/n, Ciudad Universitaria, X5000BGR, Córdoba, Argentina.
\textsuperscript{d} Universidad Tecnológica Nacional, Facultad Regional Córdoba (UTN–FRC), Maestro M. Lopez esq. Cruz Roja Argentina, Ciudad Universitaria - Córdoba Capital.
\textsuperscript{e} Centro Internacional Franco Argentino de Ciencias de la Información y de Sistemas (CIFASIS, CONICET–UNR), Ocampo y Esmeralda, S2000EZP, Rosario, Argentina.

Abstract

We present a new regular grid search algorithm for quick fixed-radius nearest-neighbor lookup developed in Python. This module indexes a set of k-dimensional points in a regular grid, with optional periodic conditions, providing a fast approach for nearest neighbors queries. In this first installment we provide three types of queries: bubble, shell and the nth-nearest; as well as three different metrics of interest in astronomy: the euclidean and two distance functions in spherical coordinates of varying precision, haversine and Vincenty; and the possibility of providing a custom distance function. This package results particularly useful for large datasets where a brute-force search turns impractical.

Keywords: Data mining: Nearest-neighbor search; Methods: Data analysis; Astroinformatics; Python Package

1. Introduction

The nearest neighbor search (NNS) problem can be defined as follows: given a set \( P \) of \( n \) points defined in the multidimensional space \( X \) with distance function \( D \), run an algorithm that, given a query point \( q \in X \), finds the point \( \min_{p \in P} D(q, p) \). This problem arises in a wide range of scientific fields, including machine learning, robotics, chemistry, astronomy and many other areas of application (e.g. Shakhnarovich et al. 2006, Teofili & Lin 2019, Devlin et al. 2015, Calle-Vallejo et al. 2015).

In the particular field of astronomy, the everyday increasing amount of observational and simulated data requires algorithms that can handle the computational demands. Most modern cosmological simulations consist of over 10\(^{10}\) particles, e.g. the Illustris Project (Springel et al. 2018), Vogelsberger et al. 2014, the MultiDark Simulation (Klypin et al. 2016) or the Millennium Simulation (Boylan-Kolchin et al. 2009), with the additional feature of being in a 3D box with periodic boundary conditions. Even smaller scale simulations may consist of 10\(^6\) particles. On the other hand, the observational community is also facing this problem thanks to large-scale sky surveys such as the Sloan Digital Sky Survey (Alam et al. 2015) and the Dark Energy Survey (Zuntz et al. 2018), and will face even greater challenges with upcoming projects like the Large Synoptic Survey Telescope (Ivezic et al. 2019).

Several methods have been proposed for solving the NNS problem and according to their solution they can be broadly divided in approximate or exact. Approximate solutions are usually of interest when working with high dimensional datasets and they retrieve points that may fall outside the query radius by a given uncertainty parameter \( \epsilon \), such that \( D(q, p) \leq (1+\epsilon)D(q, p)^\ast \), where \( D(q, p)^\ast \) is the true distance (Maneewongvatana & Mount 1999).

The simplest and direct solution to the problem is the brute force method, which requires to compute the distance \( D(q, p) \) for every point \( p \in P \). The data structure required by this method is quite simple, namely an array with the original set of points, and thus the overhead is very small. However, given that it performs every possible distance calculation, for large number of points this becomes computationally expensive and a different approach is needed. The most popular method is to apply a partitioning-indexing scheme to track the approximate location of points in the multidimensional space. Among the algorithms that apply this concept are the binary-tree and cell techniques. The former one is an iterative method that divides the space into two nodes, or branches, in each iteration until a certain number of particles is reached. The overhead and construction time of the tree structures can be quite large but in exchange they offer fairly short query times. For a detailed review of binary-trees the reader is referred to the seminal works by Friedman et al. (1977) and Bentley (1975). On the other hand, cell techniques create a regular grid, or hypercube, in the multidimensional domain and through a simple math operation every point is assigned an integer lattice that points to its corresponding cell. A hash table can then be used for future queries, where the same math operation is applied to the query point to know which cell it belongs to. The distance to every point in the cell, and probably in the contiguous cells as well, has to be computed to return only the points that meet the query condi-
GriSPy adopts the cell technique approach to solve the NNS problem in a multidimensional space. This method provides the perfect particularities that can be exploited by the extremely problem in a multidimensional space. This method provides the tion.

After the grid boundaries are defined, the coordinates of a data point, \( x_i \), can be converted to grid coordinates using:

\[
g_i = \text{int} \left( N \frac{x_i - w_{li}}{w_{ri} - w_{li}}, \quad i = 1, 2, ..., k \right)
\]

where \( w_{li} \) and \( w_{ri} \) are the left and right wall coordinates, respectively, for the coordinate \( i \). Once the grid coordinates are computed, a hash table is created where the key is each cell co-
ordinate and the value is a list containing the indices of every point within that particular cell. For GriSPy we implemented a Python dictionary using a tuple as key and a list as value.

\[ D(x_0, x) \leq r_{\text{max}} \]

In the case of a shell query, i.e. points within a minimum \( r_{\text{min}} \) and maximum \( r_{\text{max}} \) radius, we remove from the distance computation those inner cells untouched by the minimum radius to exclude points that we know apriori are outside the distance bounds. We then retrieve those points that meet the condition:

\[ r_{\text{min}} < D(x_0, x) \leq r_{\text{max}} \]

As a feature of GriSPy, a different radius can be provided for each centre in both types of queries.

The last type of query implemented is the \( n \)-th nearest neighbors. Given that it is not possible to know beforehand exactly how many neighboring cells needs to be opened, we make an initial estimation using the length of a cell diagonal as the ra-

2.2. Searching

Once the hash table is created, the query for neighbors within a given radius is straightforward. First, we extract the box of cells that contains the hyper-sphere using equation 1 and then keep only those cells touched by the hyper-sphere. We then retrieve every data point contained within those cells using the hash table and compute the distance to remove those points lo-
cated outside the hyper-sphere:

\[ D(x_0, x) \leq r_{\text{max}} \]

2.3. Distance metrics

To compute the distance between two points we implemented three metrics for the first version of the package that are of interest in astronomy. The euclidean distance defined as:

\[ D(x_0, x) = \sqrt{\sum_{i=1}^{k} (x_{0i} - x_i)^2} \]

And two distance functions defined in the surface of a unit sphere. In these cases the set of points and centres coordinates are two-dimensional and correspond to longitude and latitude, i.e. \((\lambda, \varphi)\). One of them is the haversine formula which deter-

\[ D(x_0, x) = 2 \arcsin \sqrt{\sin^2 \frac{\Delta \varphi}{2} + \cos \varphi_0 \cos \varphi \sin^2 \frac{\Delta \lambda}{2}} \]
The last distance function is the Vincenty formula (Vincenty 1975) which solves numerical problems for very close points and antipodal points at the expense of more computing time. The general formula gives the distance between two points on the surface of an ellipsoid. However, we are interested in the distance of the surface. An ellipsoid. However, we are interested in the distance of the surface.

$$D(x_0, x) = \arctan \left( \frac{\sqrt{E^2 + F^2}}{G} \right)$$

$$E = \cos \varphi \sin(\Delta \varphi)$$

$$F = \cos \varphi_0 \sin \varphi - \sin \varphi_0 \cos \varphi (\Delta \lambda)$$

$$G = \sin \varphi_0 \sin \varphi + \cos \varphi_0 \cos \varphi (\Delta \lambda)$$

### 2.4. Periodicity

Periodicity is a key ingredient in many simulations, where it is beyond practical capabilities to simulate an extremely large box. Instead, a smaller, representative box, with periodic boundary conditions is used. Particles near the box walls experience the effects caused by the presence of a ghost box that starts exactly where the main box ends.

When searching for neighbors of a centre with a search radius that extends beyond the box edge, the algorithm needs to retrieve points located on the opposite side of the box. To implement this behavior we create ghost centres located at a distance \( L_{\text{box}} \) in the opposite direction as shown in Figure 2. In GriSPy we implemented axis-independent periodic conditions, i.e. each dimension may or may not present periodic boundaries.

### 3. Technical details about the GriSPy package

Throughout the entire implementation of GriSPy we make heavy use of NumPy (van der Walt et al. 2011) vectorized methods and array broadcasting properties to achieve high performance. NumPy provides efficient implementation of numerical computations in a high-level language like Python but completely compiled in C, resulting in a significant speed improvement and in code that is both transparent and easy to maintain.

3.1. User functionalities

GriSPy is an object oriented package that expose the main grid constructions as a GriSPy() class. In the configuration step the user provides the set of k-dimensional points to be indexed, and optionally some other configuration parameters such as periodicity conditions, number of cells and the distance metric. Besides the three distance metrics provided by GriSPy, the user has the possibility of providing a callable custom distance function in the metric argument.

The instance of the GriSPy class has the following queries implemented as methods:

- **bubble_neighbors()**: find neighbors within a given radius. A different radius for each centre can be provided. Neighbors can be sorted by distance.
- **shell_neighbors()**: find neighbors within given lower and upper radius. Different lower and upper radius can be provided for each centre. Neighbors can be sorted by distance.
- **nearest_neighbors()**: find the n-th nearest neighbors for each centre. Neighbors can be sorted by distance.

And the following method is available:

- **set_periodicity()**: optional periodic boundary conditions can be provided for each axis individually.

An in depth description of the methods parameters can be found in the documentation (see Section 3.3).

### 3.2. Application example

As a simple usage application we show how to compute the two-point correlation function with the Davis & Peebles estimator (Davis & Peebles 1983) in a gravitational N-body simulation. We use the last snapshot (redshift zero) of a dark matter only simulation of 512\(^3\) particles in a periodic box of 500\(h^{-1}\)Mpc side with cosmological parameters \(\Omega_m = 0.258\), \(\Omega_{\Lambda} = 0.742\), \(h = 0.719\) and the normalisation parameter is \(\sigma_8 = 0.796\). The simulation was evolved using the public version of GADGET-2 code (Springel 2005) and used in other works (e.g. Paz et al. 2011).

```python
>>> import numpy as np

# import the class from the grispypackage
>>> from grispypy import GriSPy

# number of bins
>>> Nbins = 20
>>> r_min, r_max = 0.5, 30.0
>>> bins = np.geomspace(r_min, r_max, Nbins+1)

# Box with periodic conditions
>>> periodic = {0: (0, lbox),
... 1: (0, lbox),
... 2: (0, lbox)}

# Build GriSPy object
>>> gsp = GriSPy(Pos, periodic=periodic)
```
# Query Distances

```python
>>> shell_dist, shell_ind = gsp.shell_neighbors(
...     Pos, distance_lower_bound=r_min,
...     distance_upper_bound=r_max)
```

# Count particle pairs per bin

```python
>>> counts_DD = np.zeros(Nbins)
>>> for ss in shell_dist:
...     cc, _ = np.histogram(ss, bins)
...     counts_DD += cc
```

# Compute TPCF with theoretical randoms

```python
>>> npart = len(Pos)
>>> rho = npart / lbox**3
>>> vol_shell = np.diff(
...     ...     4.0 * np.pi / 3.0 * bins**3)
>>> count_DR = npart * rho * vol_shell
>>> xi_r = count_DD / count_DR - 1
```

3.3. Quality assurance

To ensure the proper software quality of the GriSPy package and the development process, we provide standard qualitative and quantitative metrics, in particular unit-testing and code-coverage, and endorse the PEP 8 style guide throughout the entire project.

The purpose of unit-testing is to validate that the individual components of the software work as expected (Jazayeri 2007). Code-coverage measures how much of the code is covered by the test suite, expressed as a percentage of executed sentences (Miller & Maloney 1963). Providing an exhaustive code-coverage prevents major parts of the code from being untested. In the GriSPy project we provide four suites of unit-tests covering different sections of the code, reaching 99% of code-coverage. As we are interested in the maintainability of the project, we adopted the PEP 8 – Style Guide for Python Code (van Rossum et al. 2001) to improve the readability and consistency of the code by using the flake8 tool, which ensures that there are no deviations in style and will help minimize the “code-entropy” of future versions.

The complete source code is under the MIT-license (Initiative 2019), and available in a public repository. Changes and new versions committed to this repository are automatically tested with a continuous-integration service. Documentation is automatically generated from GriSPy docstrings and made public in the read-the-docs service.

At last, GriSPy is available for installation on the Python-Package-Index (PyPI). The interested user can install it via the command `pip install grispy`; and finally the project is in the process of registration with the “Astrophysics Source Code Library” (ASCL). Allen & Schmidt 2014

3.4. Benchmarking

As previously seen, the GriSPy algorithm can be divided in two steps: build and query. The time taken by each one of them to return results will highly depend on their respective input parameters. In order to assess their time performance we created a series of scenarios where key parameters are varied and the user time is measured.

Every input parameter has an impact on the time taken to return a given neighbors query. For example, if the parameter `sorted=True` is passed as an argument to `bubble_neighbors()`, it will naturally take longer to return results because the neighbors will be ordered according to their distances. Given that build and query times (hereafter BT and QT, respectively) depend on a complicated way on every input parameter, we focus on those of most interest: number of dimensions (k), number of grid cells (N_cells), number of data points (N) and number of query centres (N_centres). Of these parameters the number of grid cells is the only one that can be modified to optimize the queries, the rest depend on the particular problem and most of the time can not be changed. For this reason we analyze the time dependence with respect to the number of cells, varying a given parameter and fixing the rest. This will also give us a helpful insight about the optimal choice of the default value. Two cases are considered for the analysis: a uniform random distribution and the N-body simulation used in the example (see Section 3.2).

We first created a random uniform distribution with values in the range (0, 1) in each dimension. For the queries we used the `bubble_neighbors()` method with a search radii of 0.01. All distances are computed with the euclidean metric. In Figure 4 we show the results of the analysis where the relation time vs. N_cells is studied for three cases:

(a) varying dimension (k) for fixed number of data-points (N) and centres (N_centres): The BT increases for increasing number of cells, despite the dimension k. However,
Figure 5 shows the result of the scaling relation time vs. $N$ number of data-points $(N_{\text{cells}}, k)$ scenario of a gravitational $N$-body simulation. When considering different centres, however, there is a slight increase in the QT only for the larger set of points ($N \approx 10^7$).

In this paper we present the first version of GriSPy: Grid Search in Python, a module for fast nearest neighbors searches. This algorithm indexes a set of $k$-dimensional points in a regular grid or hypercube. Through a simple math operation evidence from any corruption or using the periodicity of up to 1 boxsize, individually adjustable on each axis.

On the other hand, cKDTree presents almost all these characteristics, extending the periodicity up to $n$ boxsizes, and incorporates some more as the possibility of using multiprocesses in the search. However, unlike the previous two classes which implement several metrics of Euclidean and non-Euclidean geometries, cKDTree is limited to use only a Minkowski $p$-norm, where $p$ can vary in each search, and the user can not provide a custom distance function.

Finally, cKDTree incorporates other additional features, such as protecting the original construction instance and indexing of the space through a regular grid. However, unlike the previous two classes which incorporate some more as the possibility of using multiprocesses in the search. However, unlike the previous two classes which implement several metrics of Euclidean and non-Euclidean geometries, cKDTree is limited to use only a Minkowski $p$-norm, where $p$ can vary in each search, and the user can not provide a custom distance function.

In the end, since these three packages are free and implemented in Python, all can take advantage of the Python scientific-stack synergy.

4. Conclusions

In this paper we present the first version of GriSPy: Grid Search in Python, a module for fast nearest neighbors searches. This algorithm indexes a set of $k$-dimensional points in a regular grid or hypercube. Through a simple math operation every point is assigned an integer lattice that points to its corresponding cell. Then a hash table is constructed to save this.

**OS:** CentOS Linux 7 (Core) 64bits

**Software:** Python 3.7.5, NumPy 1.17.3 and SciPy 1.3.1

3.5. Short comparison with similar projects

For the near neighbors search, GriSPy is based on the partitioning and indexing of the space through a regular grid. However, as we mentioned earlier, there are other solutions (in addition to brute force) for this purpose. For example, the best known packages make use of binary trees to address the search. In particular, the Scipy library (Virtanen et al. 2019) implements the KDTree algorithm in Cython, cKDTree$^8$ while Scikit-learn$^9$ also incorporates a BallTree$^9$ scheme.

By contrasting the two comparable methods (search of the k nearest neighbor and all points within a given radius), the three classes exhibit a very similar ease of use, being able to deal with N-dimensional data.

Specifically, GriSPy presents almost the same utilities exposed by BallTree, except for the possibility of individually selecting the number of neighbors to search around each particular centre. However, GriSPy incorporates other additional features, such as protecting the original construction instance from any corruption or using the periodicity of up to 1 boxsize, individually adjustable on each axis.

On the other hand, cKDTree presents almost all these characteristics, extending the periodicity up to $n$ boxsizes, and incorporates some more as the possibility of using multiprocesses in the search. However, unlike the previous two classes which implement several metrics of Euclidean and non-Euclidean geometries, cKDTree is limited to use only a Minkowski $p$-norm, where $p$ can vary in each search, and the user can not provide a custom distance function.

Finally, cKDTree incorporates other additional search schemes, but among them is not the shell query method that distinguishes GriSPy. Full details of these comparisons with the most popular alternative packages can be found in the Appendix A.

In the end, since these three packages are free and implemented in Python, all can take advantage of the Python scientific-stack synergy.

4. Conclusions

In this paper we present the first version of GriSPy: Grid Search in Python, a module for fast nearest neighbors searches. This algorithm indexes a set of $k$-dimensional points in a regular grid or hypercube. Through a simple math operation every point is assigned an integer lattice that points to its corresponding cell. Then a hash table is constructed to save this.

---

8. https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html#scipy.spatial.KDTree

9. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html#sklearn.neighbors.BallTree

---

CPU: Intel Xeon CPU E5-2660v4 @ 2.00GHz

RAM: 128 GB DDR4 (1200-2001 MHz)
Figure 4: Time benchmark of GriSPy in a uniform distribution of points using the bubble_neighbors method with a search radii of 1% of the box size. Each column represents the time spent in each step, from left to right: Build, Query and Total time. First row: varying dimension for fixed number of data-points (N) and centres (Ncent). Second row: varying number of data-points (N) for fixed dimension and number of centres (Ncent). Third row: varying number of centres (Ncent) for fixed dimension and number of data-points (N). In all cases the error bars are the standard deviation of 10 realizations.
information for later queries. In this first installment we provide the following types of query: bubble_neighbors(), to find neighbors within a given radius; shell_neighbors(), to find neighbors within given lower and upper radius; and nearest_neighbors(), to find the n-th nearest neighbors. We also implemented the following features: possibility of working with periodic boundary conditions in each dimension; individual query radius can be provided for each centre; three distance functions of interest in astronomy can be used (euclidean, haversine and Vincenty), and the possibility of providing a custom distance function.

4.1. Caveats and future work

The reader may have noticed that this first version of GriSPy has some limitations. The most notable is the fact that both, build and query, are performed in a single process. Parallelization is currently being developed, however to deliver the most efficient implementation further work is required. Another aspect to improve is the algorithm behind the nearest_neighbors() method, which has proven to be suboptimal. The main reason behind this is the fact that the entire cell-technique scheme was thought to have a high performance in fixed-radius queries and not in n-th nearest neighbors searches. Nevertheless, new ideas will be tested to deliver a practical method.

Finally, future releases of GriSPy will include new implementations such as new distance metrics, methods to return only counters instead of distances and indices, the possibility of computing two-point and three-point correlation functions, conditional n-th nearest neighbor queries (i.e. find the n-th nearest neighbors within a subset of data points that satisfy a given condition, for example a difference in magnitude: \( |m_{\text{points}} - m_{\text{centre}}| < 3 \)).

References

Alam, S., Albareti, F. D., Allende Prieto, C., et al. 2015, ApJS, 219, 12
Allen, A. & Schmidt, J. 2014, arXiv preprint arXiv:1407.5378

Figure 5: Time benchmark of GriSPy in a gravitational N-body simulation using the bubble_neighbors method with a search radii of 1% of the box size. We compare the difference of considering uniformly distributed random centres against centres in a highly clustered region, i.e. dark matter halos.
## Appendix A. Comparative table for Nearest-Neighbors search packages

In the following table we summarize some of the most recognized Near Neighbors search exact implementations in Python. See [subsection 3.5](#) for a full discussion.

| Class      | cKDtree  | BallTree | GriSPy     |
|------------|----------|----------|------------|
| Package    | scipy    | scikit-learn | grispy    |
| Module     | spatial  | neighbors | -          |
| Version    | 1.3.3    | 0.22     | 0.0.2      |
| Indexing structure | Binary tree | Binary tree | Fixed grid |
| Data dimension | N        | N        | N          |
| Periodicity | n lbox   | No       | lbox       |
| Distance metrics | Minkowski p-norm | Internal or user defined | Internal or user defined |
| Copy data  | Yes      | No       | Yes        |
| Multiprocessing | Yes      | No       | Yes        |

### Search methods

| method | cKDtree | BallTree | GriSPy     |
|--------|---------|----------|------------|
| bubble* | query_ball_point | query_radius | bubble_neighbors |
| search radii | Multiples | Multiples | Multiples |
| return sorted | Opt      | Opt      | Opt        |
| count only | Opt      | Opt      | No         |
| return dist  | No       | Opt      | Ever       |
| k-NN* | query | query | nearest_neighbors |
| search neighbors | Multiples | Multiples | Single |
| return sorted | Ever | Opt | Ever |
| return dist  | Ever | Opt | Ever |
| dist upper bound | Opt | No | No |
| Others# | count_neighbors | - | shell_neighbors |
| query_ball_tree | query_pairs |

* Find all points within given distances of each centre.
* Find the k nearest-neighbors for each centre.
# Query methods that are not comparable between classes. See packages documentation for more details.