hypoct Documentation

Release 0.1

Kenneth L. Ho

February 19, 2014

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

A hyperoctree is a geometrical tree data structure where each node is recursively subdivided into orthants. It is a multidimensional generalization of the binary tree in 1D, the quadtree in 2D, and the octree in 3D.

hypoct contains routines for constructing and manipulating point hyperoctrees. Its primary purpose is to support fast tree-based algorithms such as the fast multipole method (FMM). In this context, it provides an efficient hierarchical indexing scheme as well as encodes all near- and far-field information via neighbor and interaction lists.

Special features include:

- Compatibility with general elements by associating with each point a size. The effect of working with elements is that point distributions are no longer strictly contained within each node but can extend slightly beyond it. Elements are assigned to nodes that are at least four times their size and have a modified "two over" neighbor definition in order to fully capture the near field.
- Optimizations for non-uniform and high-dimensional data such as adaptive subdivision and pruning of empty leaves. In particular, short dimensions are not bisected in order to keep nodes as hypercubic as possible.
- Support for periodic domains.
- Support for "sparse elements" suitable for finite element computations. Sparse elements interact only by overlap and are assigned to nodes that are at least twice their size. The usual "one over" neighbor definition then suffices to capture all external interactions.

hypoct is written in Fortran with wrappers in C and Python. This documentation mainly covers the Python interface, the intended method for most users.

1.1 Motivation and design

To explain the motivation and design choices behind this library, it is perhaps easiest to begin by describing how a typical tree algorithm is implemented. For this, imagine a set of points distributed over some spatial region in d dimensions. Most codes proceed by first enclosing all of the points within a sufficiently large hypercube called the root node. If the root contains more than a prescribed maximum number of points, it is split into 2^d children of equal size via bisection in each dimension and its points distributed accordingly between them. Each of these children are in turn subdivided if they contain too many points, and the process repeated for each new node added. The leaves of the tree are those nodes that do not have any children; they obviously satisfy the prescribed maximum occupancy condition.

1.1.1 General elements

In the fast algorithm context, a key piece of information that one would like to query from the tree is which points lie in the near and far fields of a given cluster of points. A point is in the far field of a cluster if it is separated from that

cluster by at least the cluster's size (also called *well-separated*). The near field is just the complement of the far field and can sometimes afford a more convenient description. A tree provides a natural clustering of points via the nodes to which they belong.

For true points, which are zero-dimensional objects, the near field of a node is composed of precisely its neighbors, i.e., those nodes of the tree which immediately adjoin it. However, we are often interested in not just points but general elements, for example, triangles to describe a surface in 3D or cubes to tessellate a volume. These are characterized by a nonzero size, e.g., the diameter of the triangle or the extent of the cube. We will stick to the point formulation for tree construction here, but we associate with each point now a size corresponding to the size of the element which it represents. It is helpful to think of this size as the diameter of the element and the point as its centroid, but it can be any point in or on the element. Then the problem as compared to the pure point setting is that elements can now extend beyond the boundaries of the node to which they belong. We can also regard elements as continuous point distributions, in which case the point distribution belonging to a given node is no longer contained strictly within it but rather within a slightly enlarged 'halo' region.

A consequence of this is that the usual "one over" neighbor definition is insufficient to impose that non-neighbors be well-separated. At minimum, the neighbors of a halo extension must include nodes that are at least "two over". We choose this as our modified neighbor definition for elements, which constrains that we assign elements to nodes that are at least four times their size. In particular, this means that not all elements are pushed down to the next level upon subdivision; large elements are held back to ensure correctness.

Note: This also means that not all nodes need obey the maximum occupancy condition, though they will do so as much as possible given the size restriction. In particular, if a leaf contains only elements whose natural level is deeper than the leaf's level, then the leaf will satisfy the occupancy condition.

Note: The simple neighbor definition above characterizes the case when two nodes are at the same level. When one node is larger than the other, the situation becomes slightly more complicated.

There is also an additional facility for elements that interact only sparsely, i.e., by overlap. This is suitable for, e.g., a finite element discretization of a partial differential equation. Sparse elements are assigned to nodes that are at least twice their size, for which the usual "one over" neighbor definition suffices to capture all external interactions. This substantially reduces the extent of the near field as compared with general 'dense' elements.

1.1.2 Periodic domains

Problems involving periodic domains are encountered with some regularity and these, too, present troubles for standard tree codes. In this setting, the given data represent the unit cell, which is supplemented by additional parameters defining its extent. Building the tree itself is not problematic, but finding a neighbor structure compatible with the periodic geometry produces some difficulty. This is because the neighboring nodes across boundaries will generally not align with the node structure inside the unit cell. Thus, we must force the tree to conform to the problem geometry so that it is compatible with the imposed periodicity.

We therefore allow the extent of the root node to be specified by the user (and otherwise calculated from the data if not supplied). Since this can lead to highly skewed aspect ratios, we also institute an adaptive subdivision criterion such that new nodes are as hypercubic as possible. This is achieved by halving only those node dimensions that are at least $1/\sqrt{2}$ times the maximum node dimension at each level, which ensures a bound of $\sqrt{2}$ on the aspect ratio for nodes at levels for which all dimensions are bisected.

1.1.3 Further adaptivity

We have already seen some adaptivity in the splitting of nodes above, but for problems with non-uniform distributions or in high dimensions, further adaptivity is often desired. A potential bottleneck in such codes is the creation of all

2^d children for a given node upon subdivision, which for large d can become expensive. In this library, we therefore create only those nodes which are needed, allowing us to handle far larger dimensions than usual. This also has the effect of pruning empty leaves from the tree, which is a relatively standard feature.

1.1.4 Dynamic memory

Finally, we wanted also to make use of some modern Fortran features to streamline the algorithm as compared to classical Fortran 77 programs. The main beneficiary is the use of dynamic memory. Older Fortran-style codes would typically run the tree algorithm twice: first to go through and count up the amount of memory required and then again to perform the actual memory writes once a properly sized block has been allocated. Here, we do essentially both at the same time by doubling the size of our working array each time that more memory is requested. All arrays are resized appropriately on output. Granted, this is a pretty minor point given all of the other languages out there with dynamic memory, but it is rather useful from a Fortran 77 standpoint.

Other features that we took advantage of include modules for better encapsulation of data and routines, and some shorthand notation for clarity and, perhaps, optimizability.

1.2 Algorithmic overview

We have already discussed the tree construction process above. Briefly, to review, it consists of recursively subdividing nodes following a top-down sweep, alternately deciding which nodes to divide and then which points within those nodes to hold from further subdivision.

Finding neighbors similarly involves a top-down sweep. We first initialize the neighbors at the two coarsest levels as a base case, then at each finer level search for the neighbors of each node among the children of its parent's neighbors. This hence nests the neighbor search hierarchically and results in good performance.

For points, the neighbors of a given node consist of:

- All nodes at the same level immediately adjoining it ("one over").
- All non-empty nodes at a coarser level (parent or above) immediately adjoining it.

For elements, the neighbors of a given node consist of:

- All nodes at the same level separated by at most the node's size ("two over").
- All non-empty nodes at a coarser level (parent or above) whose halo extensions are separated its own extension by less than its extension's size.

Finally, for sparse elements, the neighbors consist of:

- All nodes at the same level immediately adjoining it ("one over").
- All non-empty nodes at a coarser level (parent or above) whose halo extensions overlap with its own extension.

In all cases, a node is not considered its own neighbor.

For FMM codes, it is also useful to have access to the interaction list of a node, which consists of:

- All nodes at the same level that are children of the neighbors of the node's parent but not neighbors of the node itself.
- All non-empty nodes at a coarser level (parent or above) that are neighbors of the node's parent but not neighbors of the node itself.

Interaction lists define a systematic multiscale tiling of space and provide an efficient organization of the main FMM computations. Here, we generate interaction lists as follows. First, we initialize the lists for the three coarsest levels.

Then for each node at a finer level, we simply apply the definition directly by searching among the children of its parent's neighbors.

For data distributions that are not too pathological, meaning here that the elements are not oversized and that the data do not consist of separate point clusters of vastly different scales (which, in principle, could be handled by constructing a tree on each cluster individually), the following complexity estimates hold, where N is the number of points:

- The running time to build a tree scales as $\mathcal{O}(N \log N)$, while its memory requirement is $\mathcal{O}(N)$.
- Both the time and memory complexities for finding all neighbors are $\mathcal{O}(N)$.
- Both the time and memory complexities for generating all interaction lists are $\mathcal{O}(N)$.

1.3 Licensing and availability

hypoct is freely available under the GNU GPL and can be downloaded at https://github.com/klho/hypoct. To request alternate licenses, please contact the author.

CHAPTER

TWO

INSTALLING

This section describes how to compile and install hypoct on Unix-like systems. Primary prerequisites include Git, GNU Make, and a Fortran compiler such as GFortran. Secondary prerequisites, depending on which options are desired, include a C compiler such as GCC (for the C interface); F2PY, Python, NumPy, and matplotlib (for the Python interface); and Sphinx and LaTeX (for the documentation).

hypoct has only been tested using the GFortran and GCC compilers; the use of all other compilers should be considered "at your own risk" (though they should really be fine).

2.1 Code repository

All source files for hypoct (including those for this documentation) are available at https://github.com/klho/hypoct. To download hypoct using Git, type the following command at the shell prompt:

```
$ git clone https://github.com/klho/hypoct /path/to/local/repository/
```

2.2 Compiling

There are several targets available to compile, namely:

- the main Fortran library;
- the C wrapper for the Fortran library;
- the Python wrapper for the Fortran library;
- this documentation; and
- driver programs calling the library from each of the above languages.

To see all available targets, switch the working directory to the root of the local repository and type:

```
$ make help
```

Hopefully the instructions are self-explanatory; for more explicit directions, please see below. Before beginning, view and edit the file `Makefile` to ensure that all options are properly set for your system. In particular, if you will not be using GFortran or GCC, be sure to set alternate compilers as appropriate.

To compile the main Fortran library, type:

```
$ make fortran
```

To compile the C wrapper, type:

```
$ make c
```

To compile the Python wrapper, type:

```
$ make python
```

To compile all three, type:

```
$ make
```

or:

```
$ make all
```

All object files are placed in the directory `bin`.

To compile the documentation files, type:

```
$ make doc
```

Output HTML and PDF files are placed in the directory `doc`.

2.3 Driver programs

hypoct also contains driver programs in Fortran, C, and Python to demonstrate the use of the library and its various wrappers. To compile the drivers, type:

```
$ make fortran_driver
```

or:

```
$ make c_driver
```

or:

```
$ make python_driver
```

as appropriate. The above commands also automatically execute the corresponding programs. The driver programs are discussed in more detail in *Tutorial*.

CHAPTER

THREE

TUTORIAL

We now present a tutorial on using the Python interface to hypoct. From here on, we will therefore assume that hypoct has been properly installed along with its Python wrapper; if this is not the case, please go back to *Installing*. The choice to cover only the Python interface is merely for convenience. For help regarding the Fortran or C interfaces, please consult the corresponding source code and driver programs.

3.1 Overview

The Python interface is located in the directory `python`, which contains the directory `hypoct`, organizing the main Python package, and `hypoct_python.so`, the F2PY-ed Fortran library. The file `hypoct_python.so` contains all wrapped routines and is imported by `hypoct`, which creates a somewhat more convenient (but still pretty bare-bones) object-oriented interface around it. For details on the Python modules, please see the *Python API*; for details on data formats, please refer to the Fortran source code.

We will now step through the process of running a program calling hypoct from Python, following the Python driver program as a guide.

3.2 Initializing

The first step is to import `hypoct` by issuing the command:

```
>>> import hypoct
```

at the Python prompt. This should work if you are in the `python` directory; otherwise, you may have to first type something like:

```
>>> import sys
>>> sys.path.append(/path/to/hypoct/python/)
```

in order to tell Python where to look.

Now let's generate some data. As an example, we consider points distributed uniformly on the unit circle:

```
>>> import numpy as np
>>> n = 100
>>> theta = np.linspace(0, 2*np.pi, n+1)[:n]
>>> x = np.array([np.cos(theta), np.sin(theta)], order='F')
```

Note the use of the flag `order='F'` in `numpy.array()`. This instantiates the array in Fortran-contiguous order and is important for avoiding data copying when passing to the backend.

3.3 Building the tree

Building a tree can be as easy as typing:

```
>>> tree = hypoct.Tree(x)
```

Of course, this uses only the default options, e.g., sorting with a maximum occupany parameter of one, treating the data as zero-dimensional points, and no control over the extent of the root. To specify any of these, we can use keyword arguments as explained in `hypoct.Tree()`. We also outline these briefly below.

3.3.1 Adaptivity setting

To switch the build mode from adaptive to uniform subdivision (i.e., all leaves are divided if any one of them violates the occupany condition), enter:

```
>>> tree = hypoct.Tree(x, adap='u')
```

The default is `adap='a'`.

3.3.2 Tree depth

To control the level of subdivision, we can set the maximum leaf occupancy using the `occ` keyword. For example, to subdivide until all leaves contain no more than five points, we can type:

```
>>> tree = hypoct.Tree(x, occ=5)
```

The default is `occ=1`.

It is also possible to set the maximum tree depth explicitly using the `lvlmax` keyword, e.g.:

```
>>> tree = hypoct.Tree(x, lvlmax=3)
```

The root is denoted as level zero. The default is `lvlmax=-1`, which specifies no maximum depth. Both `occ` and `lvlmax` can be employed together, with `lvlmax` setting a hard limit on the tree.

3.3.3 Element setting

To treat the points as elements, each with a size, first create an array containing the size of each point, then build the tree using the `elem` and `siz` keywords. For instance, to consider the points as elements each of size 0.1, write:

```
>>> tree = hypoct.Tree(x, elem='e', siz=0.1)
```

where we have used the shorthand that if `siz` is a single number, then it is automatically expanded into an array of the appropriate size. Similarly, to build a tree on "sparse elements", write:

```
>>> tree =  hypoct.Tree(x, elem='s', siz=0.1)
```

The defaults are `elem='p'`, corresponding to points, and `siz=0`.

3.3.4 Root extent

The extent of the root node can be specified using the `ext` keyword, e.g.,

```
>>> tree = hypoct.Tree(x, ext=[10, 0])
```

This tells the code to set the length of the root along the first dimension to 10; its length along the second dimension is calculated from the data (since the corresponding entry is nonpositive). This is often useful if there is some external parameter governing the problem geometry, for example, periodicity conditions. Like `siz`, `ext` can also be given as a single number, in which case it is automatically expanded as appropriate. The default is `ext=0`.

3.3.5 Remarks

All options can be combined with each other. The output is stored as a `hypoct.Tree` instance, which is a thin wrapper for the arrays output from Fortran. On our machine, running:

```
>>> tree = hypoct.Tree(x)
>>> tree.lvlx
```

gives:

```
array([[  0,   1,   5,  17,  45,  97, 177, 193],
       [  6,   0,   3,   3,   3,   3,   3,   3]], dtype=int32)
```

which indicates that the tree has 6 levels (beyond the root at level 0) with 193 nodes in total. See the Fortran source code for details.

3.4 Generating auxiliary data

The base tree output is stored in a rather spartan manner; it contains only the bare minimum necessary to reconstruct the data for the entire tree. This is not always convenient and it is sometimes useful to have the data in a more easily accessible form. For instance, the base tree representation contains only parent and child identifier information that only really allows you to traverse a tree from the bottom up. To traverse a tree from the top down, we have to generate child pointers, which we can do via:

```
>>> tree.generate_child_data()
```

We can also generate geometry information (center and extent) for each node by using:

```
>>> tree.generate_geometry_data()
```

These commands create the arrays `tree.chldp`, and `tree.l`, and `tree.ctr`, respectively.

3.5 Finding neighbors

To find the neighbors of each node, type:

```
>>> tree.find_neighbors()
```

which creates the neighbor index and pointer arrays `tree.nbori` and `tree.nborp`, respectively. The method also accepts the keyword `per` indicating whether the root is periodic in a given dimension. For example, to impose that the root is periodic in the first but not the second dimension, set:

```
>>> tree.find_neighbors(per=[True, False])
```

It is worth emphasizing that the size of the unit cell cannot be directly controlled here; for this, use the `ext` keyword in `hypoct.Tree()`. As with the `siz` and `ext` keywords for `hypoct.Tree()`, we can also use shorthand by writing just, e.g.:

```
>>> tree.find_neighbors(per=True)
```

for double periodicity. The default is `per=False`.

The method `hypoct.Tree.find_neighbors()` requires that the child data from `hypoct.Tree.generate_child_data()` have already been generated; if this is not the case, then this is done automatically.

Note: Recall that neighbors are defined differently for points vs. elements as described briefly in *Introduction*.

3.6 Getting interaction lists

Interaction lists are often used in fast multipole-type algorithms to systematically cover the far field. To get interaction lists for all nodes, type:

```
>>> tree.get_interaction_lists()
```

This command requires that the neighbor data from `hypoct.Tree.find_neighbors()` have already been generated; if this is not the case, then this is done automatically using default settings. Outputs include the index and pointer arrays `tree.ilsti` and `tree.ilstp`, respectively.

3.7 Searching the tree

It is often also useful to be able to search the tree for a given set of points. This can be done via:

```
>>> trav = tree.search(x)
```

where `x` is the set of points to search for. The output `trav` is an array that records the tree traversal history for each point: the node containing the point `x[:,i]` at level `j` has index `trav[i,j]`; if no such node exists, then `trav[i,j] = 0`. By default, the tree is traversed fully from top to bottom. To limit the maximum tree depth searched, use the keyword `mlvl`.

If we have a tree on elements, then we can also attach a size to each point using the keyword `siz`. Each node in the tree traversal array, if it exists, must fully contain the point based on its size. The default is `siz=0`.

This command requires that child and geometry data have already been generated; if this is not the case, then this is done automatically.

3.8 Putting it all together

A complete example program for building a tree and generating all auxiliary data is given as follows:

```
import hypoct, numpy as np

# initialize points
n = 100
theta = np.linspace(0, 2*np.pi, n+1)[:n]
x = np.array([np.cos(theta), np.sin(theta)], order='F')

# build tree
tree = hypoct.Tree(x, occ=4)
```

```
tree.generate_child_data()
tree.generate_geometry_data()
tree.find_neighbors()
tree.get_interaction_lists()
```

This is a slightly modified and abridged version of the driver program `examples/hypoct_driver.py`.

3.9 Viewing trees in 1D and 2D

Trees in 1D and 2D can be viewed graphically using the `hypoct.tools.TreeViewer` class. To use the viewer, type:

```
>>> from hypoct.tools import TreeViewer
>>> view = TreeViewer(tree)
>>> view.draw_interactive()
```

This brings up an interactive session where each node in the tree is highlighted in turn, displaying its geometry, contained points, and neighbor and interaction list information, if available. Press `Enter` to step through the tree. All plot options can be controlled using `matplotlib`-style keywords.

Note: The `hypoct.tools.TreeViewer` class was written originally for 2D trees. It was extended to 1D trees by trivially lifting into 2D.

CHAPTER

FOUR

EXAMPLES

We give some further examples on using hypoct in this section. Some of these can also be considered tests. Insert into the preamble of each code snippet the following:

```
import hypoct, numpy as np
```

4.1 Degenerate distributions

A good test of robustness is to run hypoct on data that is degenerate in one or more dimensions:

```
x = np.random.rand(2, 100)
x[0,:] = 0
tree = hypoct.Tree(x)
```

The following output shows that the test succeeds:

```
>>> tree.lvlx

array([[  0,   1,   3,   7,  15,  31,  61, 105, 159, 193, 215, 231, 239,
       245, 249],
       [ 13,   0,   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
         2,   2]], dtype=int32)
```

Note the string of twos in `tree.lvlx[1,:]`, which indicates that subdivision occurred only along the second dimension and not the first (which has zero extent).

4.2 High-dimensional data

Another good test is on high-dimensional data. Here, we try $d = 30$, for which $2^d \sim 10^9$, near the upper limit of four-byte integer values:

```
x = np.random.rand(30, 100)
tree = hypoct.Tree(x)
tree.find_neighbors()
```

The code executes successfully with outputs:

```
>>> tree.lvlx

array([[         0,          1,        101],
       [         1,          0, 1073741823]], dtype=int32)
```

```
>>> tree.nborp

array([   0,    0,   99,  198,  297,  396,  495,  594,  693,  792,  891,
        990, 1089, 1188, 1287, 1386, 1485, 1584, 1683, 1782, 1881, 1980,
       2079, 2178, 2277, 2376, 2475, 2574, 2673, 2772, 2871, 2970, 3069,
       3168, 3267, 3366, 3465, 3564, 3663, 3762, 3861, 3960, 4059, 4158,
       4257, 4356, 4455, 4554, 4653, 4752, 4851, 4950, 5049, 5148, 5247,
       5346, 5445, 5544, 5643, 5742, 5841, 5940, 6039, 6138, 6237, 6336,
       6435, 6534, 6633, 6732, 6831, 6930, 7029, 7128, 7227, 7326, 7425,
       7524, 7623, 7722, 7821, 7920, 8019, 8118, 8217, 8316, 8415, 8514,
       8613, 8712, 8811, 8910, 9009, 9108, 9207, 9306, 9405, 9504, 9603,
       9702, 9801, 9900], dtype=int32)

>>> tree.nbori

array([  3,   4,   5, ...,  98,  99, 100], dtype=int32)
```

For higher dimensions, hypoct must be modified and recompiled to use extended precision integers, e.g., `integer*8` in Fortran, `long long` in C, and `int64` in Python (NumPy).

4.3 Periodic data

To construct a triply periodic tree with unit cell extents of, say, 10, 2, and 2, respectively, in the first, second, and third dimensions, write, e.g.:

```
x = np.random.rand(3, 100)
tree = hypoct.Tree(x, ext=[10, 2, 2])
tree.find_neighbors(per=True)
tree.get_interaction_lists()
```

The periodicity of the interaction lists inherits from that of the neighbor list.

4.4 Tree on triangles

The following code demonstrates how to build a tree on triangles:

```
# format for 'vert': vertex 'i' of triangle 'j' has coordinates 'vert[:,i,j]'
n = vert.shape[2]

# compute triangle centroids
cent = vert.mean(axis=1)

# compute triangle diameters
diam = np.empty(n)
for i in range(n):
  diam[i] = max(vert[:,:,i].max(axis=1) - vert[:,:,i].min(axis=1))

# build tree
tree = hypoct.Tree(cent, elem='e', siz=diam)
```

4.5 Changing plot styles for TreeViewer

Plot styles for `hypoct.tools.TreeViewer.draw_interactive()` can be changed by specifying `matplotlib`-type keywords. For example, using:

```
from hypoct.tools import TreeViewer
view = TreeViewer(tree)
view.draw_interactive(node_alpha=0.2, point_c='g', nbor_color='y', ilst_color='r')
```

sets the transparency level for the current node patch to 0.2, the color for points contained within the current node to green, the color of neighboring node patches to yellow, and the color of node patches in the interaction list to red.

CHAPTER

FIVE

PYTHON API

This section provides the auto-generated API for the Python modules `hypoct` and `hypoct.tools`.

5.1 hypoct

Python module for interfacing with *hypoct*.

class `hypoct.`**`Tree`**(*x*, *adap='a'*, *elem='p'*, *siz=0*, *occ=1*, *lvlmax=-1*, *ext=0*)

Build hyperoctree.

Parameters

- **x** (`numpy.ndarray`) – Point coordinates, where the coordinate of point *i* is *x[:,i]*.
- **adap** ({*'a'*, *'u'*}) – Adaptivity setting: adaptive (*'a'*), uniform (*'u'*). This specifies whether nodes are divided adaptively or to a uniformly fine level.
- **elem** ({*'p'*, *'e'*, *'s'*}) – Element type: point (*'p'*), element (*'e'*), sparse element (*'s'*). This specifies whether the input points represent true points or general elements (with sizes) that can extend beyond node boundaries. The element type determines how points are assigned to nodes and also how node neighbors are defined (see `find_neighbors()`). If *elem* = *'p'*, then *siz* is ignored. Points and elements are intended to interact densely with each other, whereas sparse elements are intended to interact only by overlap.
- **siz** (`numpy.ndarray`) – Sizes associated with each point. If *siz* is a single float, then it is automatically expanded into an appropriately sized constant array. Not accessed if *elem* = *'p'*.
- **occ** (*int*) – Maximum leaf occupancy. Requires *occ* > *0*.
- **lvlmax** (*int*) – Maximum tree depth. The root is defined to have level zero. No maximum if *lvlmax* < *0*.
- **ext** (`numpy.ndarray`) – Extent of root node. If *ext[i]* <= *0*, then the extent in dimension *i* is calculated from the data. Its primary use is to force nodes to conform to a a specified geometry (see `find_neighbors()`). If *ext* is a single float, then it is automatically expanded into an appropriately sized constant array.

`find_neighbors`(*per=False*)

Find neighbors.

The definition of a neighbor depends on the element type (see `Tree()`).

For points (*elem* = *'p'*), the neighbors of a given node consist of:

•All nodes at the same level immediately adjoining it (“one over”).

- All non-empty nodes at a coarser level (parent or above) immediately adjoining it.

For elements and sparse elements (*elem = 'e'* or *'s'*), first let the extension of a node be the spatial region corresponding to all possible point distributions belonging to that node.

Then for elements (*elem = 'e'*), the neighbors of a given node consist of:

- All nodes at the same level separated by at most the node's size ("two over")
- All non-empty nodes at a coarser level (parent or above) whose extensions are separated from its own extension by less than its extension's size.

Finally, for sparse elements (*elem = 's'*), the neighbors consist of:

- All nodes at the same level immediately adjoining it ("one over").
- All non-empty nodes at a coarser level (parent or above) whose extensions overlap with its own extension.

In all cases, a node is not considered its own neighbor.

This routine requires that the child and geometry data have already been generated. If this is not the case, then this is done automatically.

See `generate_child_data()` and `generate_geometry_data()`.

Parameters **per** (`numpy.ndarray`) – Periodicity of root note. The domain is periodic in dimension *i* if *per[i] = True*. If *per* is a single bool, then it is automatically expanded into an appropriately sized constant array. Use *ext* in `Tree()` to control the extent of the root.

generate_child_data()
Generate child data.

generate_geometry_data()
Generate geometry data.

get_interaction_lists()
Get interaction lists.

The interaction list of a given node consists of:

- All nodes at the same level that are children of the neighbors of the node's parent but not neighbors of the node itself.
- All non-empty nodes at a coarser level (parent or above) that are neighbors of the node's parent but not neighbors of the node itself.

This routine requires that the neighbor data have already been generated. If this is not the case, then this is done automatically (at default settings).

See `find_neighbors()`.

search(*x*, *siz=0*, *mlvl=-1*)
Search hyperoctree.

The element type of the points to search for are assumed to be the same as that used to construct the tree (see `Tree()`).

This routine requires that the child and geometry data have already been generated. If this is not the case, then this is done automatically.

See `generate_child_data()` and `generate_geometry_data()`.

Parameters

- **x** (numpy.ndarray) – Point coordinates to search for, where the coordinate of point *i* is x[:,i]‘.
- **siz** (numpy.ndarray) – Sizes associated with each point. If *siz* is a single float, then it is automatically expanded into an appropriately sized constant array. Not accessed if *elem = ‘p’*.
- **mlvl** (*int*) – Maximum tree depth to search. Defaults to full tree depth if *mlvl < 0*.

Returns Tree traversal array. The node containing point *i* at level *j* has index *trav[i,j]*; if no such node exists, then *trav[i,j] = 0*.

Return type numpy.ndarray

5.2 hypoct.tools

Additional tools for hypoct.

class hypoct.tools.**TreeViewer**(*tree*)

View binary and quadtrees (1D and 2D hyperoctrees).

Parameters **tree** (hypoct.Tree) – Hyperoctree.

draw_base(*c='k'*, ***kwargs*)

Draw wireframe outlines of all nodes in the tree.

Accepts all matplotlib.pyplot.plot() keyword arguments.

draw_interactive(*draw_neighbors=True*, *draw_interaction_list=True*, *base_c='k'*, *node_alpha=0.5*, *node_color='b'*, *nbor_alpha=0.5*, *nbor_color='r'*, *ilst_alpha=0.5*, *ilst_color='g'*, *point_c='k'*, ***kwargs*)

Draw each node in the tree sequentially via an interactive session (press Enter to continue) along with all neighbor and interaction list data, if available.

All keyword arguments prefaced with *‘base_’* are passed to draw_base(); those prefaced with *‘node_’* are passed to draw_node() when drawing each node; those prefaced with *‘nbor_’* are passed to draw_node() when drawing each node neighbor; and those prefaced with *‘ilst_’* are passed to draw_node() when drawing each node in the interaction list.

See draw_base() and draw_node().

Parameters

- **draw_neighbors** (*bool*) – Whether to draw neighbors.
- **draw_interaction_list** (*bool*) – Whether to draw interaction lists.

draw_node(*index*, *level=None*, *draw_points=True*, *update=True*, *node_alpha=0.5*, *node_color='b'*, *point_c='k'*, ***kwargs*)

Draw node patch and, optionally, all points contained within it.

The node is drawn as a matplotlib.patches.Rectangle instance. Points are drawn using matplotlib.pyplot.scatter().

All keyword arguments prefaced with *‘node_’* are passed to the node drawing routine with the prefix stripped, and, similarly all arguments prefixed with *‘point_’* are passed to the point drawing routine with the prefix stripped. For example, setting *node_color='r'* and *point_c='b'* passes the keyword argument *color='r'* to the node drawer and *c='b'* to the point drawer.

Parameters

- **index** (*int*) – Node index.

- **level** (*int*) – Level of node. If *None*, the level is found automatically.
- **draw_points** (*bool*) – Whether to draw points contained in the node.
- **update** (*bool*) – Whether to update the plot after drawing the node.

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

h

INDEX