contact_map Documentation

Release 0.2.0

David W.H. Swenson

Jan 20, 2018

Contents

1 Installation **3**

2 Examples **5**

3 API Reference **19**

This package provides tools for analyzing and exploring contacts (residue-residue and atom-atom) from a trajectory generated by molecular dynamics. It builds on the excellent tools provided by MDTraj.

Contacts can be an important tool for defining (meta)stable states in processes involving biomolecules. For example, an analysis of contacts can be particularly useful when defining bound states during a binding processes between proteins, DNA, and small molecules (such as potential drugs).

The contacts analyzed by `contact_map` can be either intermolecular or intramolecular, and can be analyzed on a residue-residue basis or an atom-atom basis.

This package makes it very easy to answer questions like:

- What contacts are present in a trajectory?
- Which contacts are most common in a trajectory?
- What is the difference between the frequency of contacts in one trajectory and another? (Or with a specific frame, such as a PDB entry.)
- For a particular residue-residue contact pair of interest, which atoms are most frequently in contact?

It also facilitates visualization of the contact matrix, with colors representing the fraction of trajectory time that the contact was present.

CHAPTER 1

Installation

If you're just planning to use the code, you'll want to perform a basic installation. If you're planning to develop for the code, or if you want to stay on the bleeding edge, then you should perform a developer installation.

1.1 Basic Installation

There are two recommended approaches for a basic installation: `conda`-based, or `pip`-based. Using `conda` is much easier, and will continue to be easier for anything else you install. However, the disadvantage is that you must put your entire Python environment under `conda`. If you already have a highly customized Python environment, you might prefer the `pip` install. But otherwise, we highly recommend installing `conda`, either using the full Ananconda distribution or the smaller-footprint miniconda. Once `conda` is installed and in your path, installation is as simple as:

```
conda install -c conda-forge contact_map
```

which tells `conda` to get `contact_map` from the conda-forge channel, which manages our `conda`-based installation recipe.

If you would prefer to use `pip`, that takes a few extra steps, but will work on any Python setup (`conda` or not). Because of some weirdness in how `pip` handles packages (such as MDTraj) that have a particular types of requirements from Numpy, you should install Cython and Numpy separately, so the whole install is:

```
pip install cython
pip install numpy
pip install contact_map
```

If you already have Numpy installed, you may need to re-install it with `pip install -U --force-reinstall numpy`. Note that some systems may require you to preface `pip install` commands with `sudo` (depending on where Python keeps its packages).

1.2 Developer installation

If you plan to work with the source, or if you want to stay on the bleeding edge, you can install a version so that your downloaded/cloned version of this git repository is the live code your Python interpreter sees. We call that a "developer installation."

This is a three-step process:

1. **Download or clone the repository.** If you plan to contribute changes back to the repository, please fork it on GitHub and then clone your fork. Otherwise, you can download or clone the main repository. You can follow GitHub's instructions on how to do this, and apply those steps to forking our repository at http://github.com/dwhswenson/contact_map.
2. **Install the requirements.** This can be done using either `pip` or `conda`. First, change into the directory for the repository. You should see `setup.py` and `requirements.txt` (among many other things) in that directory. Using conda:

   ```
   conda install -y --file requirements.txt
   ```

   Or, using `pip`:

   ```
   pip install cython
   pip install numpy
   pip install -r requirements.txt
   ```

   In some cases, you may need to add `-U --force-reinstall` to the Numpy step.
3. **Install the package.** Whether you get the requirements with `pip` or with `conda`, you can install the package (again, from the directory containing `setup.py`) with:

   ```
   pip install -e .
   ```

   The `-e` means that the installation is "editable" (developer version; the stuff in this directory will be the live code your Python interpreted uses) and the `.` tells it to find `setup.py` in the current directory.

1.3 Testing your installation

However you have installed it, you should test that your installation works. To do so, first check that the new package can be imported. This can be done with

```
python -c "import contact_map"
```

If your Python interpreter can find the newly-installed package, that should exit without complaint.

For a more thorough check that everything works, you should run our test suite. This can be done by installing `pytest` (using either `pip` or `conda`) and then running the command:

```
py.test --pyargs contact_map -v
```

This will run the tests on the installed version of `contact_map`. All tests should either pass or skip.

CHAPTER 2

Examples

So far, we only have one major example. We will add others here as they are written.

2.1 Contact Maps

The `contact_map` package includes some tricks to study contact maps in protein dynamics, based on tools in MDTraj. This notebook shows examples and serves as documentation.

As an example, we'll use part of a trajectory of the KRas protein bound to GTP, which was provided by Sander Roet. KRas is a protein that plays a role in many cancers. For simplicity, the waters were removed from the trajectory (although ions are still included). To run this notebook, download the example files from https://figshare.com/s/453b1b215cf2f9270769 (total download size about 1.2 MB). Download all files, and extract in the same directory that you started Jupyer from (so that you have a directory called `5550217` in your current working directory).

```
In [1]: %matplotlib inline
        import mdtraj as md
        traj = md.load("5550217/kras.xtc", top="5550217/kras.pdb")
        topology = traj.topology
```

```
In [2]: from contact_map import ContactMap, ContactFrequency, ContactDifference
```

2.1.1 Look at a single frame: `ContactMap`

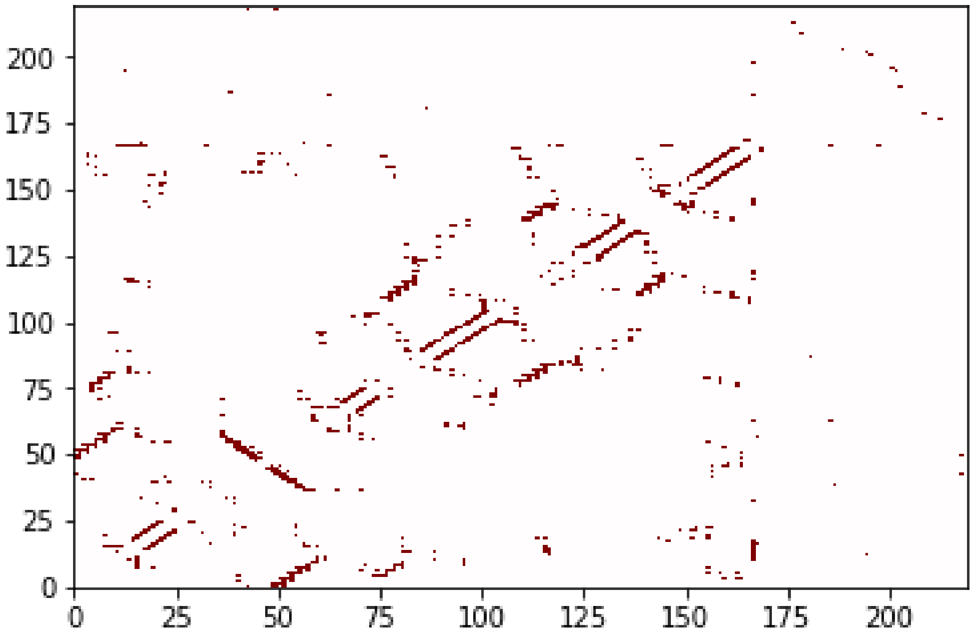
First we make the contact map for the 0th frame. For default parameters (and how to change them) see section "Changing the defaults" below.

```
In [3]: %%time
        frame_contacts = ContactMap(traj[0])
```

```
CPU times: user 123 ms, sys: 18.9 ms, total: 142 ms
Wall time: 133 ms
```

```
In [4]: %%time
        (fig, ax) = frame_contacts.residue_contacts.plot(cmap='seismic', vmin=-1, vmax=1)
```

```
CPU times: user 868 ms, sys: 19.1 ms, total: 888 ms
Wall time: 1.01 s
```



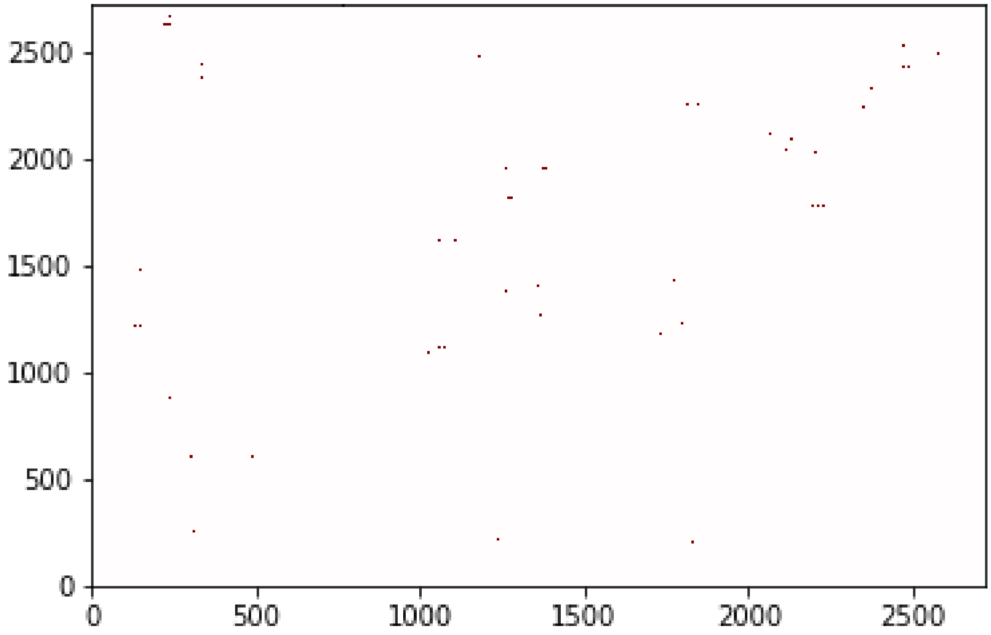
The plotting function return the `matplotlib Figure` and `Axes` objects, which allow you to make more manipulations to them later. I'll show an example of that in the "Changing the defaults" section.

We can also plot the atom-atom contacts, although it takes a little time. The built-in plotting function is best if there are not many contacts (if the matrix is sparse). If there are lots of contacts, sometimes other approaches can plot more quickly. See an example in the "Changing the defaults" section.

```
In [5]: %%time
        frame_contacts.atom_contacts.plot(cmap='seismic', vmin=-1, vmax=1);
```

```
CPU times: user 5.74 s, sys: 134 ms, total: 5.87 s
Wall time: 5.94 s
```

```
Out[5]: (<matplotlib.figure.Figure at 0x10ad66150>,
         <matplotlib.axes._subplots.AxesSubplot at 0x10bb88090>)
```



You'll notice that you don't see many points here. That is because the points are typically smaller than a single pixel at this resolution. To fix that, increase the figure's size or dpi. (Future updates to `contact_map` may provide an option to require that each point be at least one pixel in size)

2.1.2 Look at a trajectory: `ContactFrequency`

`ContactFrequency` finds the fraction of frames where each contact exists.

```
In [6]: %%time
        trajectory_contacts = ContactFrequency(traj)
```

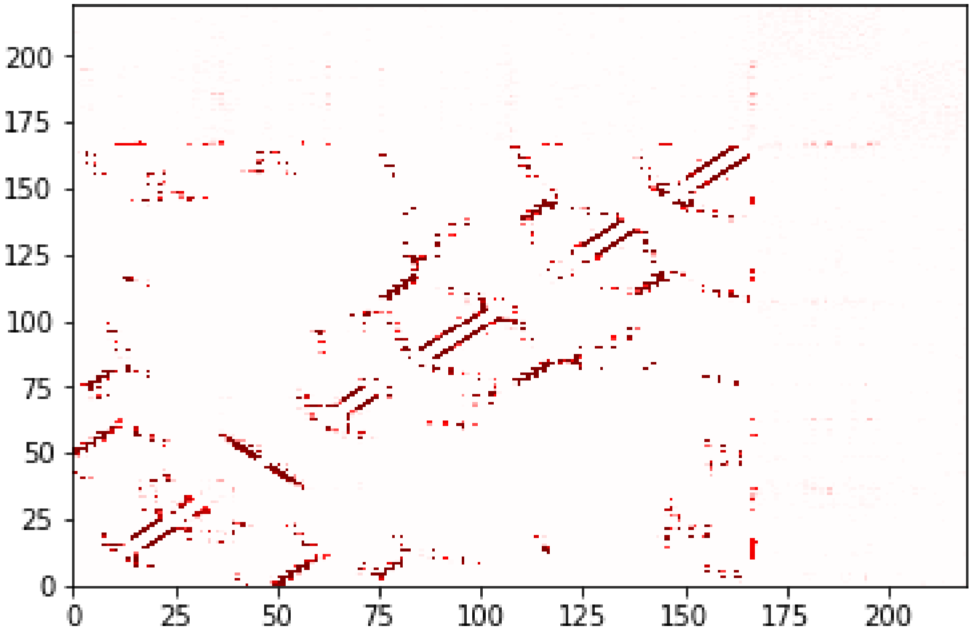
```
CPU times: user 5.02 s, sys: 31.7 ms, total: 5.05 s
Wall time: 5.05 s
```

```
In [7]: # if you want to save this for later analysis
        trajectory_contacts.save_to_file("traj_contacts.p")
        # then load with ContactFrequency.from_file("traj_contacts.p")
```

```
In [8]: %%time
        trajectory_contacts.residue_contacts.plot(cmap='seismic', vmin=-1, vmax=1);
```

```
CPU times: user 3.38 s, sys: 39.2 ms, total: 3.42 s
Wall time: 3.44 s
```

```
Out[8]: (<matplotlib.figure.Figure at 0x10f7f5310>,
         <matplotlib.axes._subplots.AxesSubplot at 0x114e8f0d0>)
```



2.1.3 Compare two: `ContactDifference`

If you want to compare two frequencies, you can use the `ContactDifference` class (or the shortcut for it, which is to subtract a contact frequency/map from another.)

The example below will compare the trajectory to its first frame.

```
In [9]: %%time
        diff = trajectory_contacts - frame_contacts
```

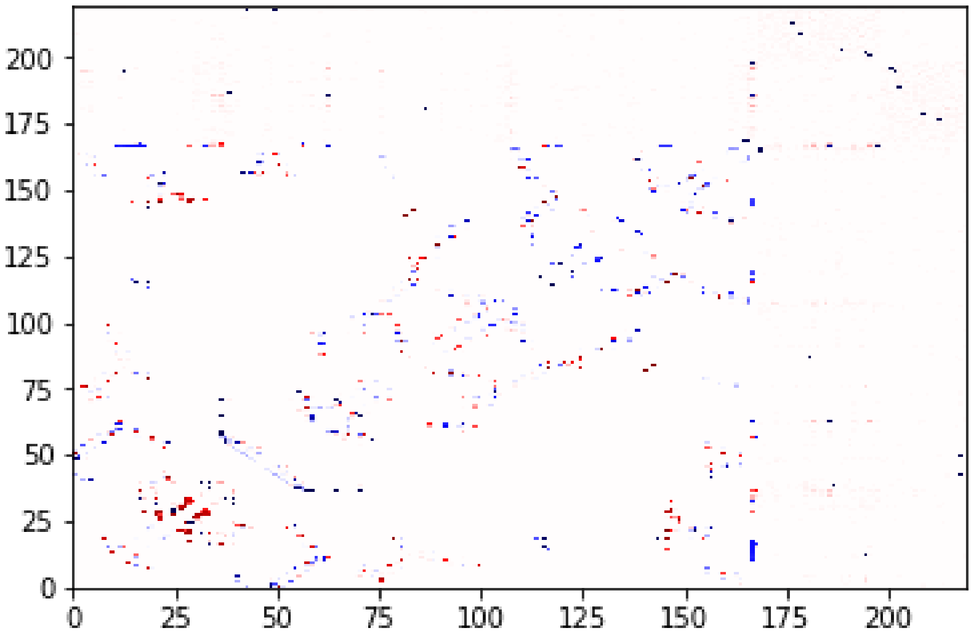
```
CPU times: user 1.13 ms, sys: 618 µs, total: 1.75 ms
Wall time: 1.28 ms
```

A contact that appears in trajectory, but not in the frame, will be at +1 and will be shown in red below. A contact that appears in the frame, but not the trajectory, will be at -1 and will be shown in blue below. The values are the difference in the frequencies (of course, for a single frame, the frequency is always 0 or 1).

```
In [10]: %%time
         diff.residue_contacts.plot(cmap='seismic', vmin=-1, vmax=1);
```

```
CPU times: user 3.17 s, sys: 50.4 ms, total: 3.22 s
Wall time: 3.22 s
```

```
Out[10]: (<matplotlib.figure.Figure at 0x112afa410>,
          <matplotlib.axes._subplots.AxesSubplot at 0x10f373950>)
```



You could have created the same object with:

```
diff = ContactDifference(trajectory_contact, frame_contacts)
```

but the simple notation using `-` is much more straightforward. However, note that `ContactDifference` makes a difference between the *frequencies* in the two objects, not the absolute count. Otherwise the trajectory would swamp the single frame, and there would be no blue in that picture!

List the residue contacts that show the most difference

First we look at the contacts that are much more important in the trajectory than the frame. Then we look at the contacts that are more important in the frame than the trajectory.

The `.most_common()` method gives a list of the contact pairs and the frequency, sorted by frequency. See also `collections.Counter.most_common()` in the standard Python `collections` module.

Here we do this with the `ContactDifference` we created, although it works the same for `ContactFrequency` and `ContactMap` (with the single-frame contact map, the ordering is a bit nonsensical, since every entry is either 0 or 1).

```
In [11]: %%time
         # residue contact more important in trajectory than in frame (near +1)
         diff.residue_contacts.most_common()[:10]
```

```
CPU times: user 6.4 ms, sys: 2.96 ms, total: 9.37 ms
Wall time: 7.05 ms
```

```
Out[11]: [([ALA146, GLN22], 0.9900990099009901),
          ([PHE82, PHE141], 0.9801980198019802),
          ([ALA83, LYS117], 0.9702970297029703),
          ([ILE84, GLU143], 0.9702970297029703),
          ([PHE90, ALA130], 0.9702970297029703),
          ([ALA146, ASN116], 0.9702970297029703),
          ([ALA155, VAL152], 0.9504950495049505),
          ([LEU113, ILE139], 0.9504950495049505),
          ([LEU19, LEU79], 0.9405940594059405),
          ([VAL81, ILE93], 0.9405940594059405)]
```

```
In [12]: # residue contact more important in frame than in trajectory (near -1)
         list(reversed(diff.residue_contacts.most_common()))[:10]
         # alternate: diff.residue_contacts.most_common()[:-10:-1] # (thanks Sander!)
```

```
Out[12]: [([NA6828, THR87], -0.9900990099009901),
          ([CL6849, NA6842], -0.9900990099009901),
          ([NA6834, SER39], -0.9900990099009901),
          ([PRO34, ASP38], -0.9900990099009901),
          ([ALA59, GLU37], -0.9900990099009901),
          ([GLN25, ASP30], -0.9900990099009901),
          ([NA6842, GLY13], -0.9900990099009901),
          ([CL6865, GLN43], -0.9900990099009901),
          ([TYR40, TYR32], -0.9900990099009901),
          ([SER65, GLU37], -0.9900990099009901)]
```

List the atoms contacts most common within a given residue contact

First let's select a few residues from the topology. Note that GTP has residue ID 201 in the PDB sequence, even though it is only residue 166 (counting from 0) in the topology. This is because some of the protein was removed, and therefore the PDB is missing those residues. The topology only counts the residues that are actually present.

```
In [13]: val81 = topology.residue(80)
         asn116 = topology.residue(115)
         gtp201 = topology.residue(166)
         print val81, asn116, gtp201
```

```
VAL81 ASN116 GTP201
```

We extended the standard `.most_common()` to take an optional argument. When the argument is given, it will filter the output to only include the ones where that argument is part of the contact. For example, the following gives the residues most commonly in contact with GTP.

```
In [14]: for contact in trajectory_contacts.residue_contacts.most_common(gtp201):
             if contact[1] > 0.1:
                 print contact
```

```
([GTP201, LEU120], 0.6435643564356436)
([GTP201, ASP119], 0.6237623762376238)
([LYS147, GTP201], 0.6138613861386139)
([ALA146, GTP201], 0.594059405940594)
([SER145, GTP201], 0.594059405940594)
([LYS117, GTP201], 0.594059405940594)
([ASP33, GTP201], 0.5742574257425742)
([GLY12, GTP201], 0.5643564356435643)
([GLY13, GTP201], 0.5544554455445545)
([VAL14, GTP201], 0.5346534653465347)
([ALA11, GTP201], 0.5346534653465347)
([GTP201, GLY15], 0.5247524752475248)
([GTP201, LYS16], 0.5247524752475248)
([SER17, GTP201], 0.5247524752475248)
([ALA18, GTP201], 0.5247524752475248)
([ASN116, GTP201], 0.4752475247524752)
([ASP57, GTP201], 0.40594059405940597)
([GTP201, GLU63], 0.39603960396039606)
([GLU37, GTP201], 0.3465346534653465)
([VAL29, GTP201], 0.297029702970297)
([NA6833, GTP201], 0.2079207920792079)
```

```
([NA6843, GTP201], 0.18811881188118812)
([THR35, GTP201], 0.1485148514851485)
([PRO34, GTP201], 0.1485148514851485)
([NA6829, GTP201], 0.1485148514851485)
```

We can also find all the atoms, for all residue contacts, that are in contact with a given residue, and return that sorted by frequency.

```
In [15]: diff.most_common_atoms_for_residue(gtp201)[:15]
```

```
Out[15]: [([GTP201-C6, LYS117-CB], 0.5346534653465347),
          ([LYS117-CA, GTP201-O6], 0.5247524752475248),
          ([GTP201-C6, LYS117-CA], 0.5247524752475248),
          ([GTP201-C8, GLY15-CA], 0.5148514851485149),
          ([GTP201-N7, GLY15-CA], 0.5148514851485149),
          ([GTP201-O2', ASP33-CG], 0.5148514851485149),
          ([GLY13-C, GTP201-PB], 0.49504950495049505),
          ([LYS117-N, GTP201-O6], 0.49504950495049505),
          ([GTP201-C2, LYS147-CB], 0.49504950495049505),
          ([GTP201-O3A, GLY13-C], 0.48514851485148514),
          ([GTP201-O2', ASP33-OD2], 0.48514851485148514),
          ([ASN116-OD1, GTP201-O6], 0.4752475247524752),
          ([ASN116-CG, GTP201-O6], 0.45544554455445546),
          ([GTP201-O6, LYS117-CB], 0.45544554455445546),
          ([GTP201-N7, ASN116-ND2], 0.45544554455445546)]
```

Finally, we can look at which atoms are most commonly in contact within a given residue contact pair.

```
In [16]: trajectory_contacts.most_common_atoms_for_contact([val81, asn116])
```

```
Out[16]: [([ASN116-CB, VAL81-CG1], 0.9702970297029703),
          ([ASN116-CG, VAL81-CG1], 0.24752475247524752),
          ([VAL81-CG1, ASN116-ND2], 0.21782178217821782),
          ([VAL81-CG1, ASN116-N], 0.0594059405940594)]
```

2.1.4 Changing the defaults

This sections covers several options that you can modify to make the contact maps faster, and to focus on what you're most interested in.

The first three options change which atoms are included as possible contacts. We call these `query` and `haystack`, and although they are conceptually equivalent, the algorithm is designed such that the query should have fewer atoms than the haystack.

Both of these options take a list of atom index numbers. These are most easily created using MDTraj's atom selection language.

```
In [17]: # the default selection is
         default_selection = topology.select("not water and symbol != 'H'")
         print len(default_selection)
```

```
1408
```

Using a different query

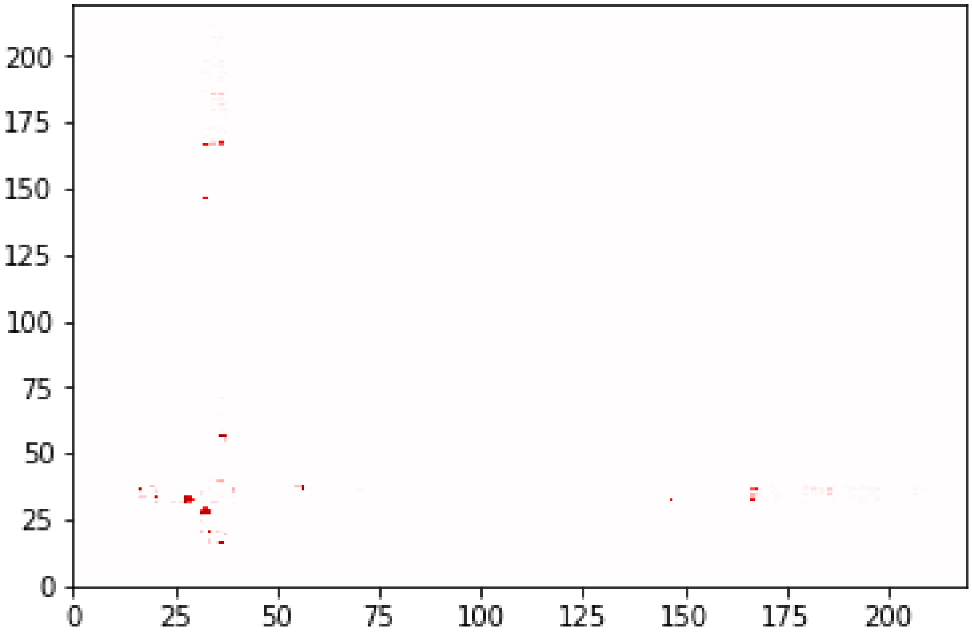
```
In [18]: switch1 = topology.select("resSeq 32 to 38 and symbol != 'H'")
         switch2 = topology.select("resSeq 59 to 67 and symbol != 'H'")
         gtp = topology.select("resname GTP and symbol != 'H'")
         mg = topology.select("element Mg")
         cations = topology.select("resname NA or resname MG")
         sodium = topology.select("resname NA")
```

```
In [19]: %%time
         sw1_contacts = ContactFrequency(trajectory=traj, query=switch1)
```

```
CPU times: user 1.81 s, sys: 8.27 ms, total: 1.82 s
Wall time: 1.83 s
```

```
In [20]: sw1_contacts.residue_contacts.plot(cmap='seismic', vmin=-1, vmax=1)
```

```
Out[20]: (<matplotlib.figure.Figure at 0x11a936210>,
          <matplotlib.axes._subplots.AxesSubplot at 0x10b006a90>)
```



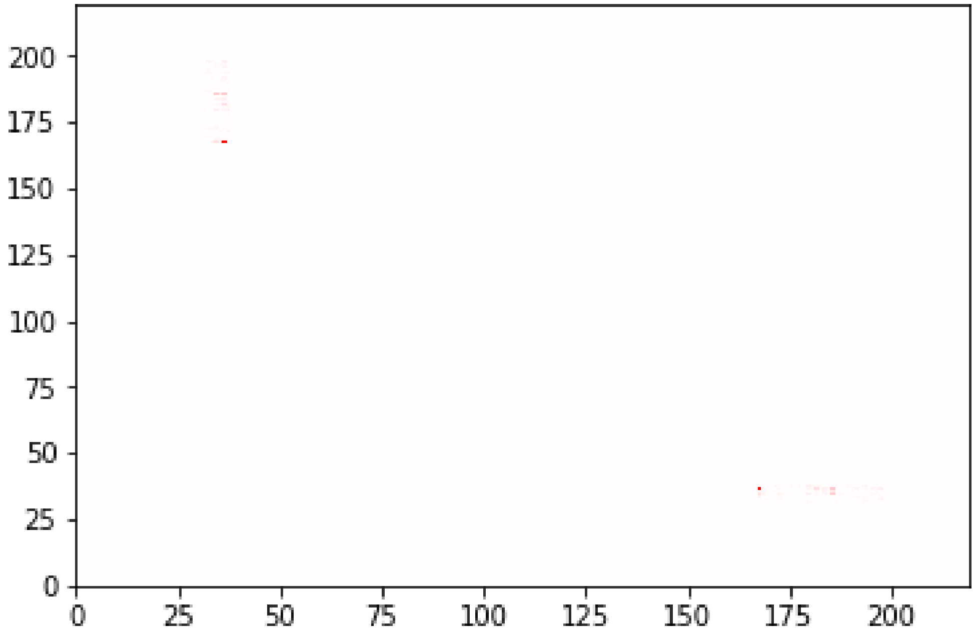
Using a different haystack

Currently, changing the haystack has essentially no effect on the performance. However, I expect to change that in the future (requires making some modifications to MDTraj).

```
In [21]: %%time
         cations_switch1 = ContactFrequency(trajectory=traj, query=cations, haystack=switch1)
```

```
CPU times: user 1.66 s, sys: 3.27 ms, total: 1.66 s
Wall time: 1.67 s
```

```
In [22]: (fig, ax) = cations_switch1.residue_contacts.plot(cmap='seismic', vmin=-1, vmax=1)
```



Let's zoom in on that. To do this, we'll do a little MDTraj magic so that we can change the *atom* ID numbers, which are what go into our `cations` and `switch1` objects, into *residue* ID numbers (and we'll use Python `sets` to remove repeats):

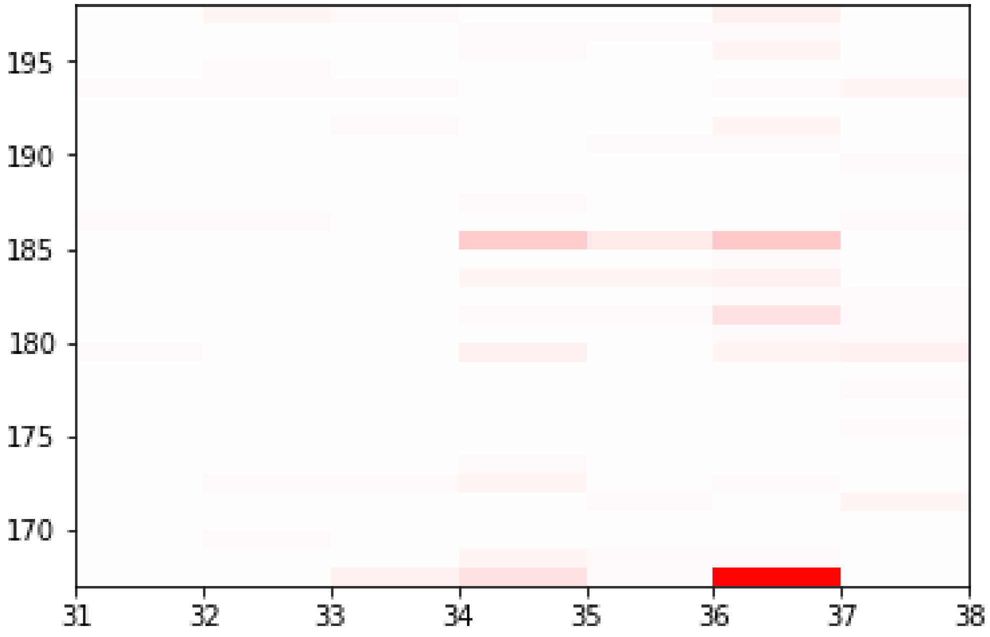
```
In [23]: def residue_for_atoms(atom_list, topology):
             return set([topology.atom(a).residue.index for a in atom_list])
```

```
In [24]: switch1_residues = residue_for_atoms(switch1, traj.topology)
         cation_residues = residue_for_atoms(cations, traj.topology)
```

Now we'll plot again, but we'll change the `x` and `y` axes so that we only see switch 1 along `x` and cations along `y`:

```
In [25]: (fig, ax) = cations_switch1.residue_contacts.plot(cmap='seismic', vmin=-1, vmax=1)
         ax.set_xlim(min(switch1_residues), max(switch1_residues) + 1)
         ax.set_ylim(min(cation_residues), max(cation_residues) + 1)
```

```
Out[25]: (167, 198)
```



Here, of course, the boxes are much larger, and are long rectangles instead of squares. The box represents the residue number that is to its left and under it. So the most significant contacts here are between residue 36 and the ion listed as residue 167. Let's see just how frequently that contact is made:

```
In [26]: print cations_switch1.residue_contacts.counter[frozenset([36, 167])]
```

```
0.485148514851
```

So about half the time. Now, which residue/ion are these? Remember, these indices start at 0, even though the tradition in science (and the PDB) is to count from 1. Furthermore, the PDB residue numbers for the ions skip the section of the protein that has been removed. But we can easily obtain the relevant residues:

```
In [27]: print traj.topology.residue(36)
         print traj.topology.residue(167)
```

```
GLU37
MG202
```

So this is a contact between the Glu37 and the magnesium ion (which is listed as residue 202 in the PDB).

Changing how many neighboring residues are ignored

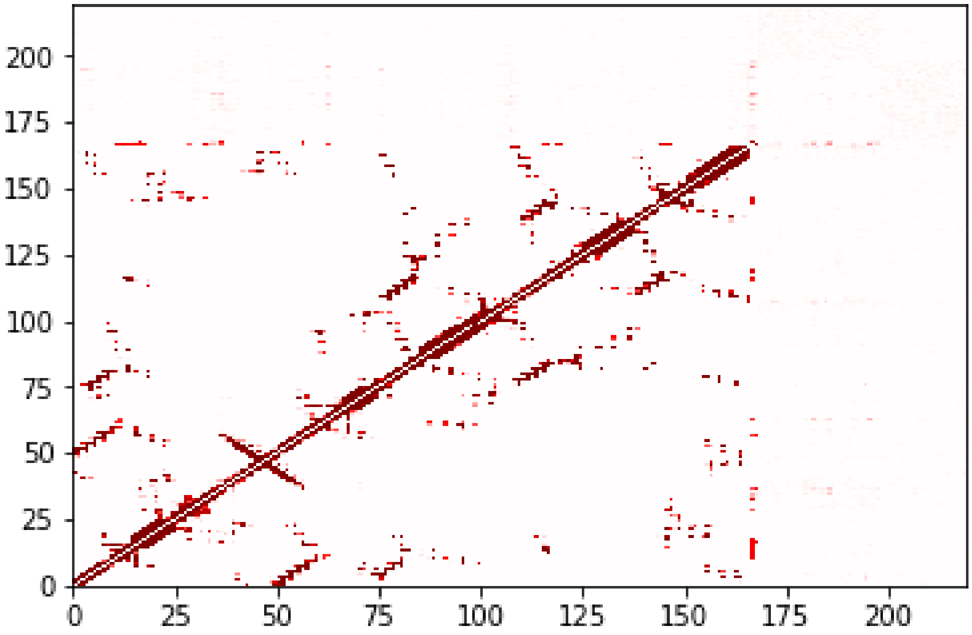
By default, we ignore atoms from 2 residues on either side of the given residue (and in the same `chain`). This is easily changed. However, even when you say to ignore no neighbors, you still ignore the residue's interactions with itself.

Note: for non-protein contacts, the `chain` is often poorly defined. In this example, the GTP and the Mg are listed sequentially in residue order, and therefore they are considered "neighbors" and their contacts are ignored.

```
In [28]: %%time
         ignore_none = ContactFrequency(trajectory=traj, n_neighbors_ignored=0)
```

```
CPU times: user 7.13 s, sys: 30.8 ms, total: 7.16 s
Wall time: 7.17 s
```

```
In [29]: ignore_none.residue_contacts.plot(cmap='seismic', vmin=-1, vmax=1);
```



Using a different cutoff

The size of the cutoff has a large effect on the performance. The default is (currently) 0.45nm.

```
In [30]: %%time
         large_cutoff = ContactFrequency(trajectory=traj, cutoff=1.5)
```

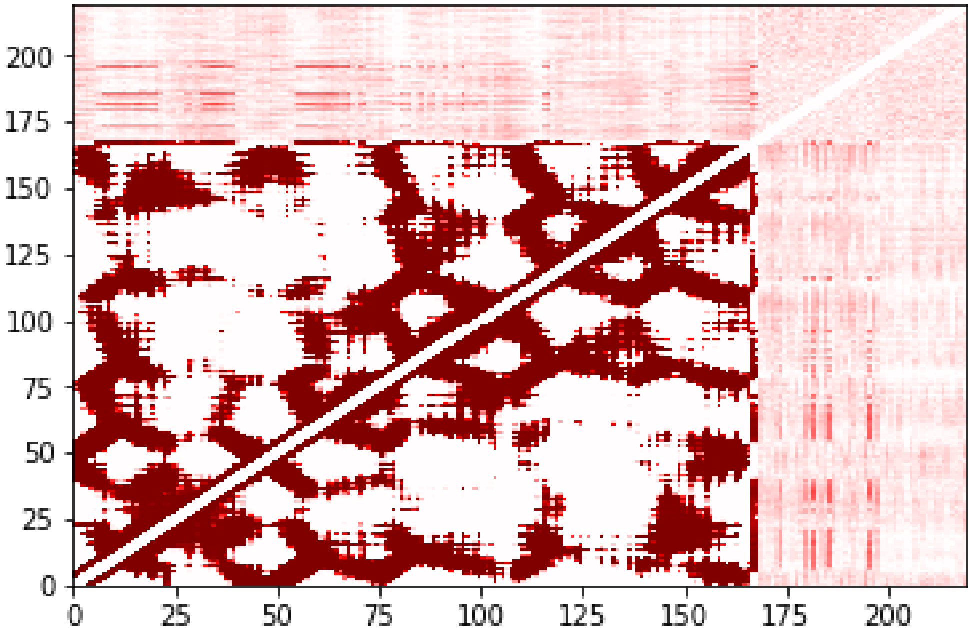
```
CPU times: user 2min 23s, sys: 1.62 s, total: 2min 24s
Wall time: 2min 25s
```

The cost of the built-in plot function also depends strongly on the number of contacts that are made. It is designed to work well for sparse matrices; as the matrix gets less sparse, other approaches may be better. Here's an example:

```
In [31]: %%time
         large_cutoff.residue_contacts.plot(cmap='seismic', vmin=-1, vmax=1);
```

```
CPU times: user 32 s, sys: 952 ms, total: 32.9 s
Wall time: 33 s
```

```
Out[31]: (<matplotlib.figure.Figure at 0x11e148610>,
          <matplotlib.axes._subplots.AxesSubplot at 0x117c91290>)
```

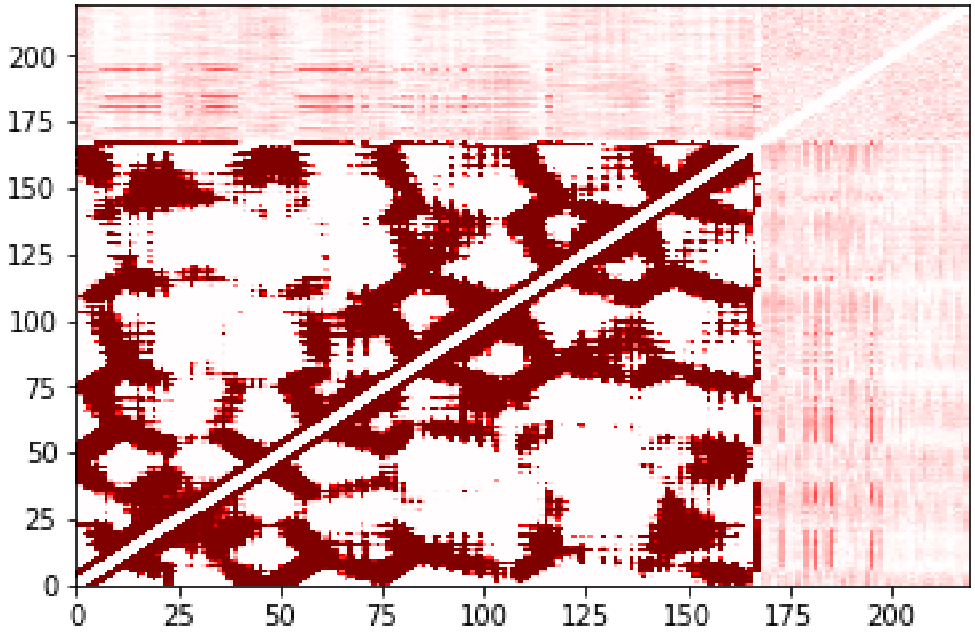


```
In [32]: %%time
         import matplotlib
         cmap = matplotlib.pyplot.get_cmap('seismic')
         norm = matplotlib.colors.Normalize(vmin=-1, vmax=1)

         plot = matplotlib.pyplot.pcolor(large_cutoff.residue_contacts.df, cmap='seismic', vmin=-1, v
         plot.cmap.set_under(cmap(norm(0)));
```

```
CPU times: user 4.55 s, sys: 66 ms, total: 4.62 s
Wall time: 4.81 s
```

```
/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/matplotlib/co
  cbook._putmask(xa, xa < 0.0, -1)
```



In this case, using the `pandas.DataFrame` representation (obtained using `.df`) is faster. On the other hand, try using this approach on the atom-atom picture at the top! That will take a while.

You'll notice that these may not be pixel-perfect copies. This is because the number of pixels doesn't evenly divide into the number of residues. You can improve this by increasing the resolution (`dpi` in matplotlib) or the figure size. However, in both versions you can see the overall structure quite clearly.

CHAPTER 3

API Reference

3.1 Contact maps

contact_map.ContactCount(counter, object_f, . . .)	Return object when dealing with contacts (residue or atom).
ContactMap(frame[, query, haystack, cutoff, . . .])	Contact map (atomic and residue) for a single frame.
ContactFrequency(trajectory[, query, . . .])	Contact frequency (atomic and residue) for a trajectory.
ContactDifference(positive, negative)	Contact map comparison (atomic and residue).

3.1.1 contact_map.ContactCount

class `contact_map.`**`ContactCount`** (*counter*, *object_f*, *n_x*, *n_y*)

Return object when dealing with contacts (residue or atom).

This contains all the information about the contacts of a given type. This information can be represented several ways. One is as a list of contact pairs, each associated with the fraction of time the contact occurs. Another is as a matrix, where the rows and columns label the pair number, and the value is the fraction of time. This class provides several methods to get different representations of this data for further analysis.

In general, instances of this class shouldn't be created by a user using `__init__`; instead, they will be returned by other methods. So users will often need to use this object for analysis.

Parameters

- **counter** (`collections.Counter`) – the counter describing the count of how often the contact occurred; key is a frozenset of a pair of numbers (identifying the atoms/residues); value is the raw count of the number of times it occurred
- **object_f** (*callable*) – method to obtain the object associated with the number used in `counter`; typically `mdtraj.Topology.residue()` or `mdtraj.Topology.atom()`.
- **n_x** (*int*) – number of objects in the x direction (used in plotting)

- **n_y** (*int*) – number of objects in the y direction (used in plotting)

__init__(*counter*, *object_f*, *n_x*, *n_y*)

Methods

__init__(counter, object_f, n_x, n_y)	
most_common([obj])	Most common values (ordered) with object as keys.
most_common_idx()	Most common values (ordered) with indices as keys.
plot([cmap, vmin, vmax])	Plot contact matrix (requires matplotlib)

counter

`collections.Counter` – keys use index number; count is contact occurrences

df

`pandas.SparseDataFrame` – DataFrame representation of the contact matrix

Rows/columns correspond to indices and the values correspond to the count

most_common(*obj=None*)

Most common values (ordered) with object as keys.

This uses the objects for the contact pair (typically MDTraj `Atom` or `Residue` objects), instead of numeric indices. This is more readable and can be easily used for further manipulation.

Parameters **obj** (*MDTraj Atom or Residue*) – if given, the return value only has entries including this object (allowing one to, for example, get the most common contacts with a specific residue)

Returns the most common contacts in order. If the list is `l`, then each element `l[e]` is a tuple with two parts: `l[e][0]` is the key, which is a pair of Atom or Residue objects, and `l[e][1]` is the count of how often that contact occurred.

Return type list

See also:

most_common_idx() same thing, using index numbers as key

most_common_idx()

Most common values (ordered) with indices as keys.

Returns the most common contacts in order. The if the list is `l`, then each element `l[e]` consists of two parts: `l[e][0]` is a pair of integers, representing the indices of the objects associated with the contact, and `l[e][1]` is the count of how often that contact occurred

Return type list

See also:

most_common() same thing, using objects as key

plot(*cmap='seismic'*, *vmin=-1.0*, *vmax=1.0*)

Plot contact matrix (requires matplotlib)

Parameters

- **cmap** (*str*) – color map name, default 'seismic'

- **vmin** (*float*) – minimum value for color map interpolation; default -1.0
- **vmax** (*float*) – maximum value for color map interpolation; default 1.0

Returns

- **fig** (`matplotlib.Figure`) – matplotlib figure object for this plot
- **ax** (`matplotlib.Axes`) – matplotlib axes object for this plot

sparse_matrix

`scipy.sparse.dok.dok_matrix` – sparse matrix representation of contacts

Rows/columns correspond to indices and the values correspond to the count

3.1.2 contact_map.ContactMap

class contact_map.**ContactMap**(*frame*, *query=None*, *haystack=None*, *cutoff=0.45*, *n_neighbors_ignored=2*)

Contact map (atomic and residue) for a single frame.

__init__(*frame*, *query=None*, *haystack=None*, *cutoff=0.45*, *n_neighbors_ignored=2*)

Methods

__init__(frame[, query, haystack, cutoff, . . .])	
contact_map(trajectory, frame_number, . . .)	Returns atom and residue contact maps for the given frame.
from_file(filename)	Load this object from a given file
most_common_atoms_for_contact(contact_pair)	Most common atom contacts for a given residue contact pair
most_common_atoms_for_residue(residue)	Most common atom contact pairs for contacts with the given residue
save_to_file(filename[, mode])	Save this object to the given file.

contact_map(*trajectory*, *frame_number*, *residue_query_atom_idxs*, *residue_ignore_atom_idxs*)

Returns atom and residue contact maps for the given frame.

Parameters

- **frame** (*mdtraj.Trajectory*) – the desired frame (uses the first frame in this trajectory)
- **residue_query_atom_idxs** (*dict*) –
- **residue_ignore_atom_idxs** (*dict*) –

Returns

- **atom_contacts** (*collections.Counter*)
- **residue_contact** (*collections.Counter*)

cutoff

float – cutoff distance for contacts, in nanometers

from_file(*filename*)

Load this object from a given file

Parameters **`filename`** (*`string`*) – the file to read from

Returns the reloaded object

Return type `ContactObject`

See also:

`save_to_file()` save to a file

`haystack`
list of int – indices of atoms to include as haystack

`most_common_atoms_for_contact`(*contact_pair*)
Most common atom contacts for a given residue contact pair

Parameters **`contact_pair`** (*`length 2 list of Residue or int`*) – the residue contact pair for which the most common atom contact pairs will be calculated

Returns Atom contact pairs for the residue contact pair, in order of frequency. Referring to the list as `l`, each element of the list `l[e]` consists of two parts: `l[e][0]` is a list containing the two MDTraj Atom objects that make up the contact, and `l[e][1]` is the measure of how often the contact occurs.

Return type list

`most_common_atoms_for_residue`(*residue*)
Most common atom contact pairs for contacts with the given residue

Parameters **`residue`** (*`Residue or int`*) – the Residue object or index representing the residue for which the most common atom contact pairs will be calculated

Returns Atom contact pairs involving given residue, order of frequency. Referring to the list as `l`, each element of the list `l[e]` consists of two parts: `l[e][0]` is a list containing the two MDTraj Atom objects that make up the contact, and `l[e][1]` is the measure of how often the contact occurs.

Return type list

`n_neighbors_ignored`
int – number of neighbor residues (in same chain) to ignore

`query`
list of int – indices of atoms to include as query

`residue_ignore_atom_idxs`
dict – maps query residue index to atom indices to ignore

`residue_query_atom_idxs`
dict – maps query residue index to atom indices in query

`save_to_file`(*filename*, *mode='w'*)
Save this object to the given file.

Parameters

- **`filename`** (*`string`*) – the file to write to
- **`mode`** (*`'w' or 'a'`*) – file writing mode. Use 'w' to overwrite, 'a' to append. Note that writing by bytes ('b' flag) is automatically added.

See also:

`from_file()` load from generated file

topology

`mdtraj.Topology` – topology object for this system

The topology includes information about the atoms, how they are grouped into residues, and how the residues are grouped into chains.

3.1.3 contact_map.ContactFrequency

class `contact_map.`**ContactFrequency**(*trajectory*, *query=None*, *haystack=None*, *cutoff=0.45*, *n_neighbors_ignored=2*, *frames=None*)

Contact frequency (atomic and residue) for a trajectory.

The contact frequency is defined as fraction of the trajectory that a certain contact is made. This object calculates this quantity for all contacts with atoms in the *query* residue, with "contact" defined as being within a certain cutoff distance.

Parameters

- **trajectory** (*mdtraj.Trajectory*) – Trajectory (segment) to analyze
- **query** (*list of int*) – Indices of the atoms to be included as query. Default `None` means all atoms.
- **haystack** (*list of int*) – Indices of the atoms to be included as haystack. Default `None` means all atoms.
- **cutoff** (*float*) – Cutoff distance for contacts, in nanometers. Default 0.45.
- **n_neighbors_ignored** (*int*) – Number of neighboring residues (in the same chain) to ignore. Default 2.

__init__(*trajectory*, *query=None*, *haystack=None*, *cutoff=0.45*, *n_neighbors_ignored=2*, *frames=None*)

Methods

__init__(trajectory[, query, haystack, . . .])	
contact_map(trajectory, frame_number, . . .)	Returns atom and residue contact maps for the given frame.
from_file(filename)	Load this object from a given file
most_common_atoms_for_contact(contact_pair)	Most common atom contacts for a given residue contact pair
most_common_atoms_for_residue(residue)	Most common atom contact pairs for contacts with the given residue
save_to_file(filename[, mode])	Save this object to the given file.

atom_contacts

Atoms pairs mapped to fraction of trajectory with that contact

contact_map(*trajectory*, *frame_number*, *residue_query_atom_idxs*, *residue_ignore_atom_idxs*)

Returns atom and residue contact maps for the given frame.

Parameters

- **frame** (*mdtraj.Trajectory*) – the desired frame (uses the first frame in this trajectory)
- **residue_query_atom_idxs** (*dict*) –

- **`residue_ignore_atom_idxs`** (*`dict`*) –

Returns

- **atom_contacts** (*collections.Counter*)
- **residue_contact** (*collections.Counter*)

`cutoff`

float – cutoff distance for contacts, in nanometers

`from_file` (*filename*)

Load this object from a given file

Parameters **`filename`** (*`string`*) – the file to read from

Returns the reloaded object

Return type `ContactObject`

See also:

`save_to_file()` save to a file

`haystack`

list of int – indices of atoms to include as haystack

`most_common_atoms_for_contact` (*contact_pair*)

Most common atom contacts for a given residue contact pair

Parameters **`contact_pair`** (*`length 2 list of Residue or int`*) – the residue contact pair for which the most common atom contact pairs will be calculated

Returns Atom contact pairs for the residue contact pair, in order of frequency. Referring to the list as `l`, each element of the list `l[e]` consists of two parts: `l[e][0]` is a list containing the two MDTraj Atom objects that make up the contact, and `l[e][1]` is the measure of how often the contact occurs.

Return type list

`most_common_atoms_for_residue` (*residue*)

Most common atom contact pairs for contacts with the given residue

Parameters **`residue`** (*`Residue or int`*) – the Residue object or index representing the residue for which the most common atom contact pairs will be calculated

Returns Atom contact pairs involving given residue, order of frequency. Referring to the list as `l`, each element of the list `l[e]` consists of two parts: `l[e][0]` is a list containing the two MDTraj Atom objects that make up the contact, and `l[e][1]` is the measure of how often the contact occurs.

Return type list

`n_frames`

Number of frames in the mapped trajectory

`n_neighbors_ignored`

int – number of neighbor residues (in same chain) to ignore

`query`

list of int – indices of atoms to include as query

`residue_contacts`

Residue pairs mapped to fraction of trajectory with that contact

`residue_ignore_atom_idxs`
dict – maps query residue index to atom indices to ignore

`residue_query_atom_idxs`
dict – maps query residue index to atom indices in query

`save_to_file` (*filename*, *mode='w'*)
Save this object to the given file.

Parameters

- **`filename`** (*`string`*) – the file to write to
- **`mode`** (*`'w' or 'a'`*) – file writing mode. Use 'w' to overwrite, 'a' to append. Note that writing by bytes ('b' flag) is automatically added.

See also:

`from_file()` load from generated file

`topology`
`mdtraj.Topology` – topology object for this system

The topology includes information about the atoms, how they are grouped into residues, and how the residues are grouped into chains.

3.1.4 contact_map.ContactDifference

class `contact_map.`**`ContactDifference`** (*positive*, *negative*)
Contact map comparison (atomic and residue).

This can compare single frames or entire trajectories (or even mix the two!) While this can be directly instantiated by the user, the more common way to make this object is by using the − operator, i.e., `diff = map_1 - map_2`.

`__init__` (*positive*, *negative*)

Methods

`__init__`(positive, negative)	
`contact_map`(*args, **kwargs)	
`from_file`(filename)	Load this object from a given file
`most_common_atoms_for_contact`(contact_pair)	Most common atom contacts for a given residue contact pair
`most_common_atoms_for_residue`(residue)	Most common atom contact pairs for contacts with the given residue
`save_to_file`(filename[, mode])	Save this object to the given file.

`cutoff`
float – cutoff distance for contacts, in nanometers

`from_file` (*filename*)
Load this object from a given file

Parameters **`filename`** (*`string`*) – the file to read from

Returns the reloaded object

Return type `ContactObject`

See also:

`save_to_file()` save to a file

`haystack`
list of int – indices of atoms to include as haystack

`most_common_atoms_for_contact`(*contact_pair*)
Most common atom contacts for a given residue contact pair

Parameters **`contact_pair`** (*`length 2 list of Residue or int`*) – the residue contact pair for which the most common atom contact pairs will be calculated

Returns Atom contact pairs for the residue contact pair, in order of frequency. Referring to the list as `l`, each element of the list `l[e]` consists of two parts: `l[e][0]` is a list containing the two MDTraj Atom objects that make up the contact, and `l[e][1]` is the measure of how often the contact occurs.

Return type list

`most_common_atoms_for_residue`(*residue*)
Most common atom contact pairs for contacts with the given residue

Parameters **`residue`** (*`Residue or int`*) – the Residue object or index representing the residue for which the most common atom contact pairs will be calculated

Returns Atom contact pairs involving given residue, order of frequency. Referring to the list as `l`, each element of the list `l[e]` consists of two parts: `l[e][0]` is a list containing the two MDTraj Atom objects that make up the contact, and `l[e][1]` is the measure of how often the contact occurs.

Return type list

`n_neighbors_ignored`
int – number of neighbor residues (in same chain) to ignore

`query`
list of int – indices of atoms to include as query

`residue_ignore_atom_idxs`
dict – maps query residue index to atom indices to ignore

`residue_query_atom_idxs`
dict – maps query residue index to atom indices in query

`save_to_file`(*filename*, *mode='w'*)
Save this object to the given file.

Parameters

- **`filename`** (*`string`*) – the file to write to
- **`mode`** (*`'w' or 'a'`*) – file writing mode. Use 'w' to overwrite, 'a' to append. Note that writing by bytes ('b' flag) is automatically added.

See also:

`from_file()` load from generated file

topology

`mdtraj.Topology` – topology object for this system

The topology includes information about the atoms, how they are grouped into residues, and how the residues are grouped into chains.

3.2 Minimum Distance (and related)

MinimumDistanceCounter(trajectory, query, . . .)	Count how often each atom pair is the minimum distance.
NearestAtoms(trajectory, cutoff[, . . .])	Identify nearest atoms (within a cutoff) to an atom.

3.2.1 min_dist.MinimumDistanceCounter

class min_dist.**MinimumDistanceCounter**(*trajectory*, *query*, *haystack*)

Count how often each atom pair is the minimum distance.

Parameters

- **trajectory** (`mdtraj.Trajectory`) – trajectory to be analyzed
- **query** (*list*) – list of the (integer) atom indices to use as the query
- **haystack** (*list*) – list of the (integer) atom indices to use as the haystack

topology

`mdtraj.Topology` – the topology object associated with the class

atom_pairs

list – list of 2-tuples representing atom index pairs to use when looking for the minimum distance

minimum_distances

list – the minimum distance between query group and haystack group at each frame of the trajectory

__init__(*trajectory*, *query*, *haystack*)

Methods

__init__(trajectory, query, haystack)

atom_count

`collections.Counter` – map from atom pair to the number of times that pair is the minimum distance

atom_history

list of 2-tuples – list of atom pairs when represent the minimum distance at each frame of the trajectory

residue_count

`collections.Counter` – map from residue pair to the number of times that pair is the minimum distance

residue_history

list of 2-tuples – list of residue pairs when represent the minimum distance at each frame of the trajectory

3.2.2 min_dist.NearestAtoms

class `min_dist.`**NearestAtoms**(*trajectory*, *cutoff*, *frame_number=0*, *excluded=None*)

Identify nearest atoms (within a cutoff) to an atom.

This was primarily written to quickly look for atoms that are nearly overlapping, but should be extendable to have other uses.

Parameters

- **trajectory** (`mdtraj.Trajectory`) – trajectory to be analyzed
- **cutoff** (*float*) – cutoff distance (in nm)
- **frame_number** (*int*) – frame number within the trajectory (counting from 0), default 0
- **excluded** (*dict*) – a dict of {atom_index: [excluded_atom_indices]}, where the excluded atom indices are atoms that should not be counted when considering the atom for the key atom_index. Default is `None`, which ignores all atoms in the same residue. Passing an empty dict, `{}`, will result in all atom pairs being considered

nearest

dict – dictionary mapping atom index to the atom index of the nearest atom to this one

nearest_distance

dict – dictionary mapping atom index to the distance to the nearest atom

__init__(*trajectory*, *cutoff*, *frame_number=0*, *excluded=None*)

Methods

`__init__`(trajectory, cutoff[, frame_number, . . .])	

sorted_distances

list – 3-tuple (atom_index, nearest_atom_index, nearest_distance) for each atom, sorted by distance.

- genindex

`contact_map` is an open source project, released under the GNU LGPL, version 2.1 or (at your option) any later version. Development takes place in public at https://github.com/dwhswenson/contact_map; your contributions would be welcome!

If you have suggestions or bug reports, please raise an issue on our GitHub issues page.

Index

Symbols

__init__() (contact_map.ContactCount method), 20
__init__() (contact_map.ContactDifference method), 25
__init__() (contact_map.ContactFrequency method), 23
__init__() (contact_map.ContactMap method), 21
__init__() (min_dist.MinimumDistanceCounter method), 27
__init__() (min_dist.NearestAtoms method), 28

A

atom_contacts (contact_map.ContactFrequency attribute), 23
atom_count (min_dist.MinimumDistanceCounter attribute), 27
atom_history (min_dist.MinimumDistanceCounter attribute), 27
atom_pairs (min_dist.MinimumDistanceCounter attribute), 27

C

contact_map() (contact_map.ContactFrequency method), 23
contact_map() (contact_map.ContactMap method), 21
ContactCount (class in contact_map), 19
ContactDifference (class in contact_map), 25
ContactFrequency (class in contact_map), 23
ContactMap (class in contact_map), 21
counter (contact_map.ContactCount attribute), 20
cutoff (contact_map.ContactDifference attribute), 25
cutoff (contact_map.ContactFrequency attribute), 24
cutoff (contact_map.ContactMap attribute), 21

D

df (contact_map.ContactCount attribute), 20

F

from_file() (contact_map.ContactDifference method), 25
from_file() (contact_map.ContactFrequency method), 24
from_file() (contact_map.ContactMap method), 21

H

haystack (contact_map.ContactDifference attribute), 26
haystack (contact_map.ContactFrequency attribute), 24
haystack (contact_map.ContactMap attribute), 22

M

minimum_distances (min_dist.MinimumDistanceCounter attribute), 27
MinimumDistanceCounter (class in min_dist), 27
most_common() (contact_map.ContactCount method), 20
most_common_atoms_for_contact() (contact_map.ContactDifference method), 26
most_common_atoms_for_contact() (contact_map.ContactFrequency method), 24
most_common_atoms_for_contact() (contact_map.ContactMap method), 22
most_common_atoms_for_residue() (contact_map.ContactDifference method), 26
most_common_atoms_for_residue() (contact_map.ContactFrequency method), 24
most_common_atoms_for_residue() (contact_map.ContactMap method), 22
most_common_idx() (contact_map.ContactCount method), 20

N

n_frames (contact_map.ContactFrequency attribute), 24
n_neighbors_ignored (contact_map.ContactDifference attribute), 26
n_neighbors_ignored (contact_map.ContactFrequency attribute), 24
n_neighbors_ignored (contact_map.ContactMap attribute), 22
nearest (min_dist.NearestAtoms attribute), 28
nearest_distance (min_dist.NearestAtoms attribute), 28
NearestAtoms (class in min_dist), 28

P

plot() (contact_map.ContactCount method), 20

Q

query (contact_map.ContactDifference attribute), 26
query (contact_map.ContactFrequency attribute), 24
query (contact_map.ContactMap attribute), 22

R

residue_contacts (contact_map.ContactFrequency attribute), 24
residue_count (min_dist.MinimumDistanceCounter attribute), 27
residue_history (min_dist.MinimumDistanceCounter attribute), 27
residue_ignore_atom_idxs (contact_map.ContactDifference attribute), 26
residue_ignore_atom_idxs (contact_map.ContactFrequency attribute), 24
residue_ignore_atom_idxs (contact_map.ContactMap attribute), 22
residue_query_atom_idxs (contact_map.ContactDifference attribute), 26
residue_query_atom_idxs (contact_map.ContactFrequency attribute), 25
residue_query_atom_idxs (contact_map.ContactMap attribute), 22

S

save_to_file() (contact_map.ContactDifference method), 26
save_to_file() (contact_map.ContactFrequency method), 25
save_to_file() (contact_map.ContactMap method), 22
sorted_distances (min_dist.NearestAtoms attribute), 28
sparse_matrix (contact_map.ContactCount attribute), 21

T

topology (contact_map.ContactDifference attribute), 26
topology (contact_map.ContactFrequency attribute), 25
topology (contact_map.ContactMap attribute), 23
topology (min_dist.MinimumDistanceCounter attribute), 27