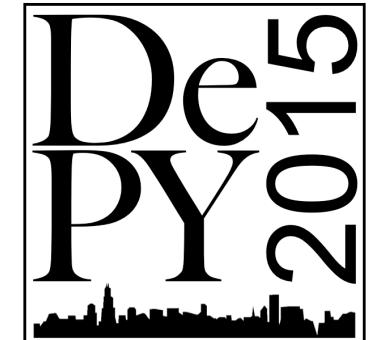
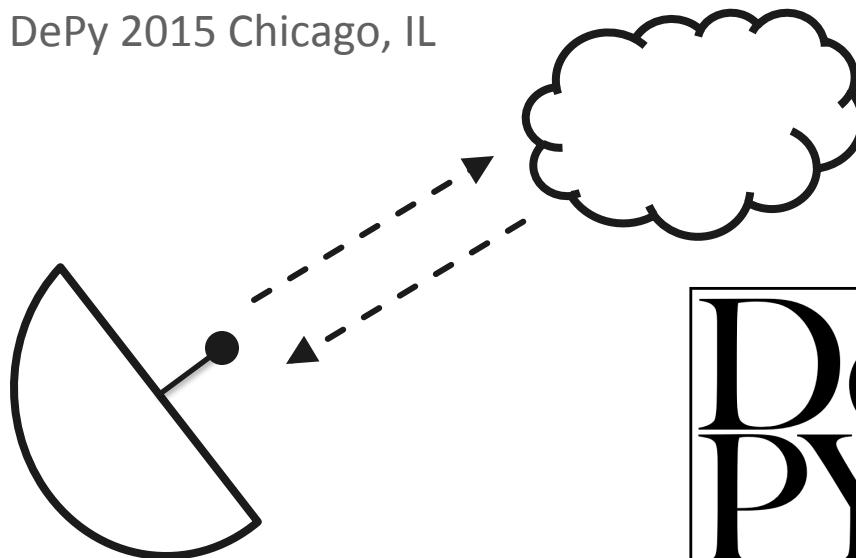


## Speeding Up Python Data Analysis Using Cython ...and stories about weather radar

Jonathan Helmus, Argonne National Laboratory

May 29<sup>th</sup>, 2015

DePy 2015 Chicago, IL



# The ARM Climate Research Facility

- The U.S. Department of Energy's Atmospheric Radiation Measurement (ARM) Climate Research facility provides *in situ* and remote sensing data with a mission to **improve climate and earth systems models**.
- The program operates a **large number of instruments** at three **fixed sites** as well as two **mobile facilities** that can be deployed in support of field campaigns around the globe.
- This instrumentation includes a number of scanning **cloud and precipitation radars** which were acquired with funding from the American Recovery Act.
- The program is the process of preparing **additional new scanning radars** for deployment in 2015.

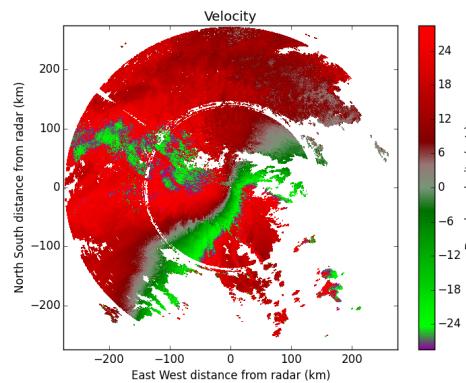
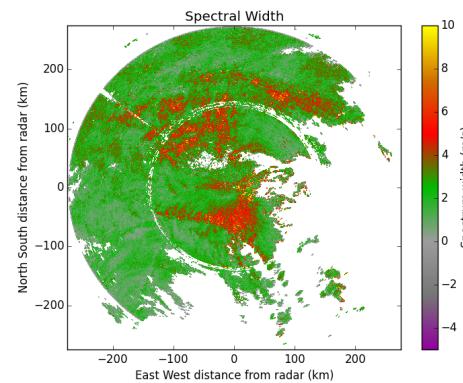
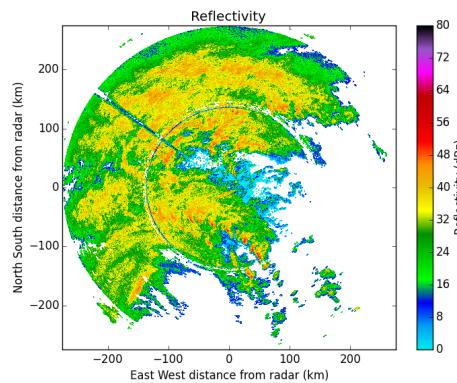
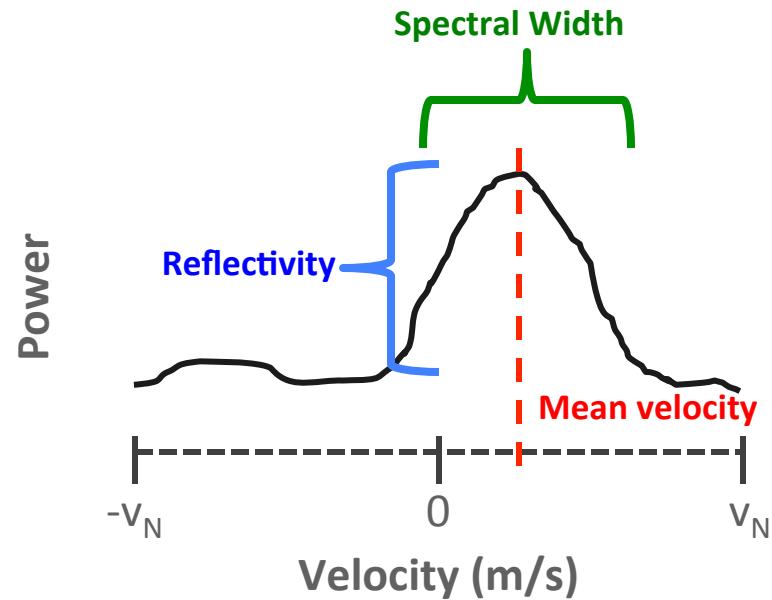
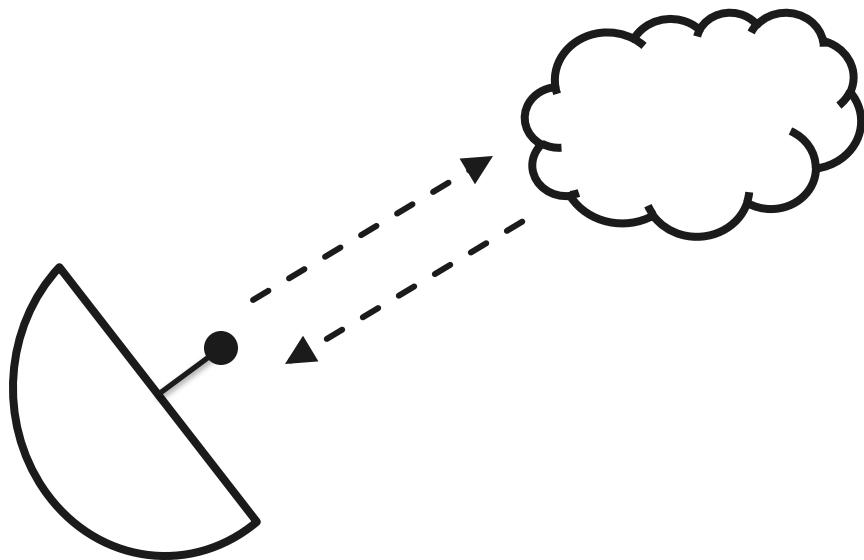


# The Python ARM Radar Toolkit: Py-ART

- Py-ART is a module for visualizing, correcting and analyzing **weather radar data** using packages from the scientific Python stack.
- Development began to address the needs of the **ARM** program with the acquisition of **multiple scanning cloud and precipitation radars**.
- The project has since been expanded to work with a **variety of weather radars**, including NEXRAD and TDWR radars, and a wide user base including radar researchers, weather enthusiasts and climate modelers.
- Available on GitHub as **open source software** under a BSD license, [arm-doe.github.io/pyart/](https://arm-doe.github.io/pyart/).
- Conda packages are available at [binstar.org/jjhelmus](https://binstar.org/jjhelmus) for Windows and OS X.



# Doppler weather radar 101

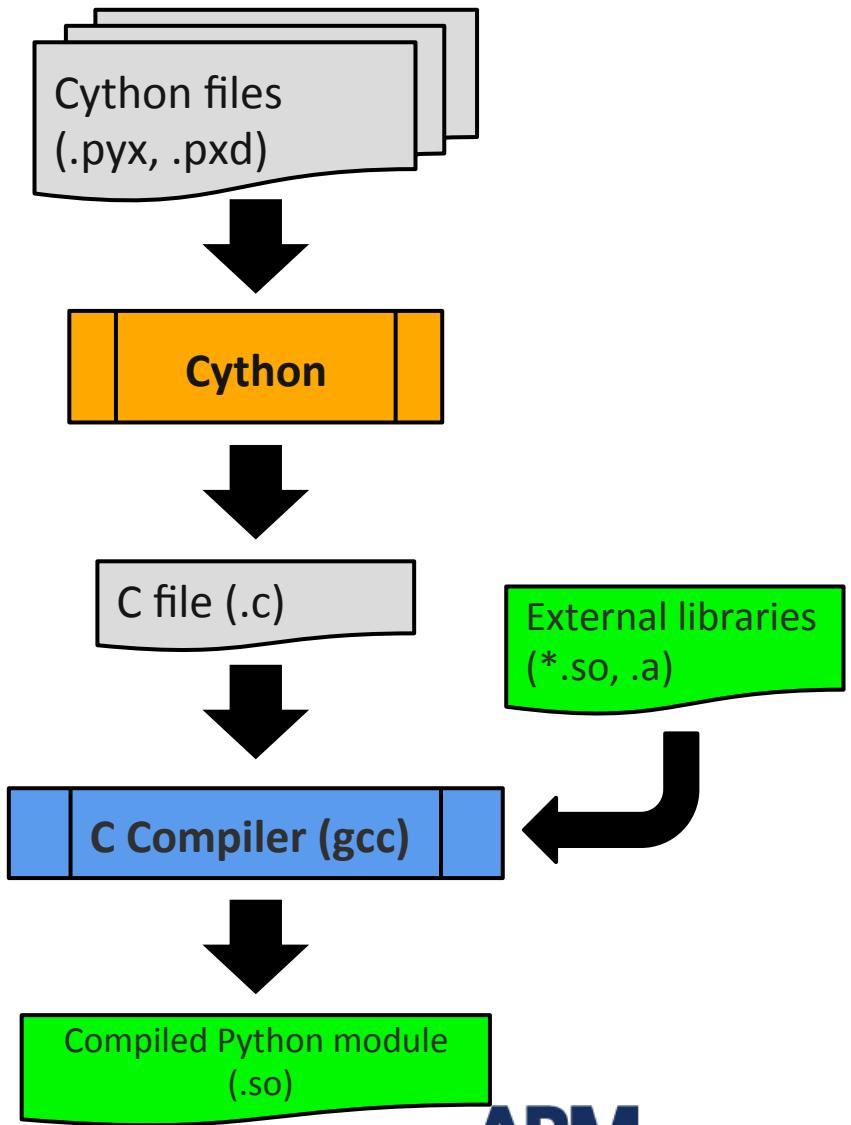


# Accessing C libraries in Python with Cython

The NASA TRMM Radar Software Library (RSL) is capable of reading in radar data in a number of formats. We used Cython to create a wrapper around this library in Py-ART.

## Cython

- Python to C code translator.
- Generates a Python extension module.
- Language additions make it easy to interact with C/C++ functions and classes
- Can also be used to speed up Python code by adding static type information.



# Wrapping the RSL library: Cython

```
cdef extern from "rsl.h":\n\n    ctypedef struct Radar:\n        Radar_header h\n        Volume **v\n\n    ctypedef struct Radar_header:\n        int month, day, year\n        int hour, minute\n        float sec\n        ...\n\n    ctypedef struct Volume:\n        Volume_header h\n        Sweep **sweep\n        ...\n    ...\n\n    Radar * RSL_anyformat_to_radar(char *infile)\n    ...\n    void RSL_free_volume(Volume *v)\n    void RSL_free_radar(Radar *r)
```

\_rsl.h.pxd

```
cimport _rsl_h\n\n    cdef class RslFile:\n        cdef _rsl_h.Radar * _Radar\n        cdef _rsl_h.Volume * _Volume\n\n        def __cinit__(self, filename):\n            self._Radar = _rsl_h.RSL_anyformat_to_radar(filename)\n            if self._Radar is NULL:\n                raise IOError('file cannot be read. ')\n\n        def __dealloc__:\n            _rsl_h.RSL_free_radar(self._Radar)\n\n        def get_volume(self, int volume_number):\n            rslvolume = _RslVolume()\n            rslvolume.load(self._Radar.v[volume_number])\n            return rslvolume\n        ...\n\n        property month:\n            def __get__(self):\n                return self._Radar.h.month\n            def __set__(self, int month):\n                self._Radar.h.month = month
```

\_rsl\_interface.pyx



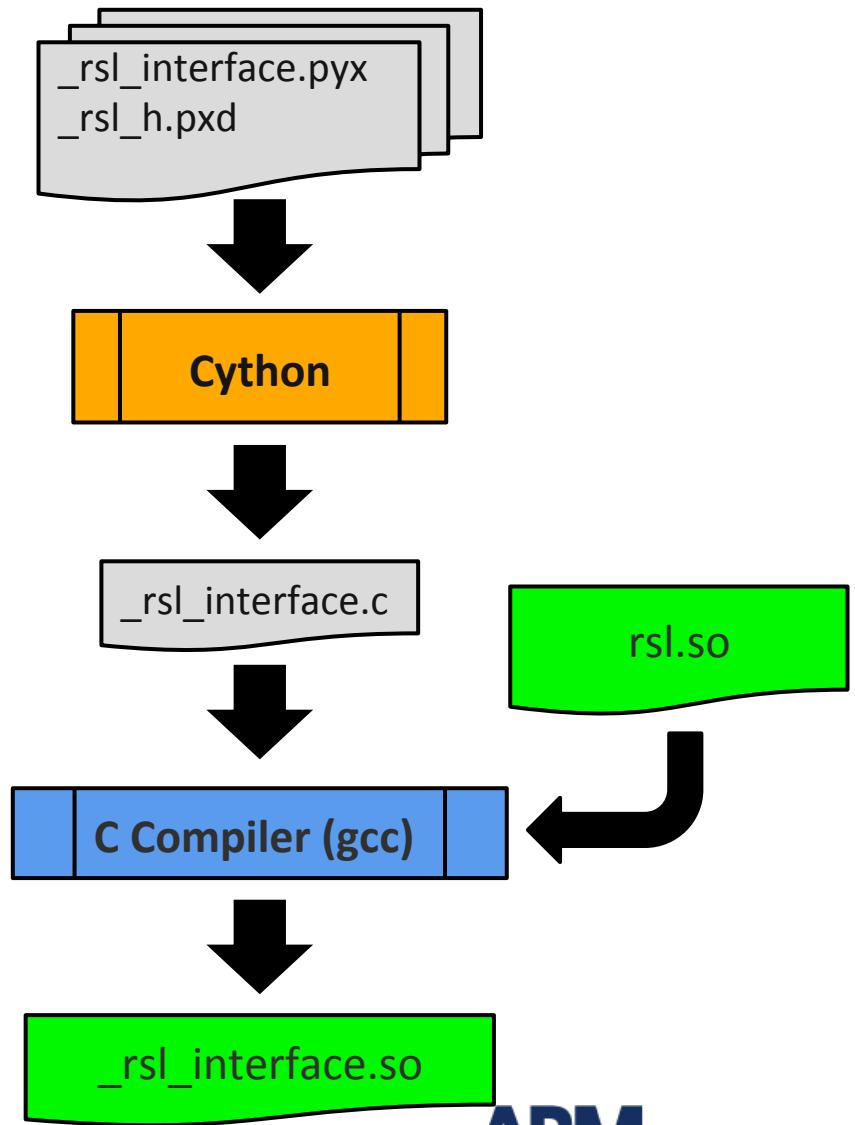
# Wrapping the RSL library: Building

```
def configuration(parent_package='', top_path=None):
    from numpy.distutils.misc_util import Configuration
    config = Configuration('io', parent_package, top_path)

    # determine and verify the at RSL location
    rsl_path = os.environ.get('RSL_PATH')
    if rsl_path is None:
        rsl_path = guess_rsl_path()
    rsl_lib_path = os.path.join(rsl_path, 'lib')
    rsl_include_path = os.path.join(rsl_path, 'include')

    # build the RSL interface if RSL is installed
    if check_rsl_path(rsl_lib_path, rsl_include_path):
        config.add_extension(
            '_rsl_interface',
            sources=['_rsl_interface.c'],
            libraries=['rsl'],
            library_dirs=[rsl_lib_path],
            include_dirs=[rsl_include_path] + [get_include()],
            runtime_library_dirs=[rsl_lib_path])
    else:
        import warnings
        warnings.warn(RSL_MISSING_WARNING % (rsl_path))
```

setup.py

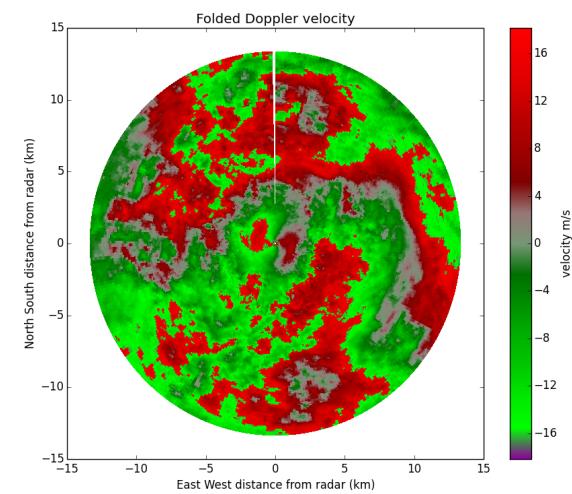
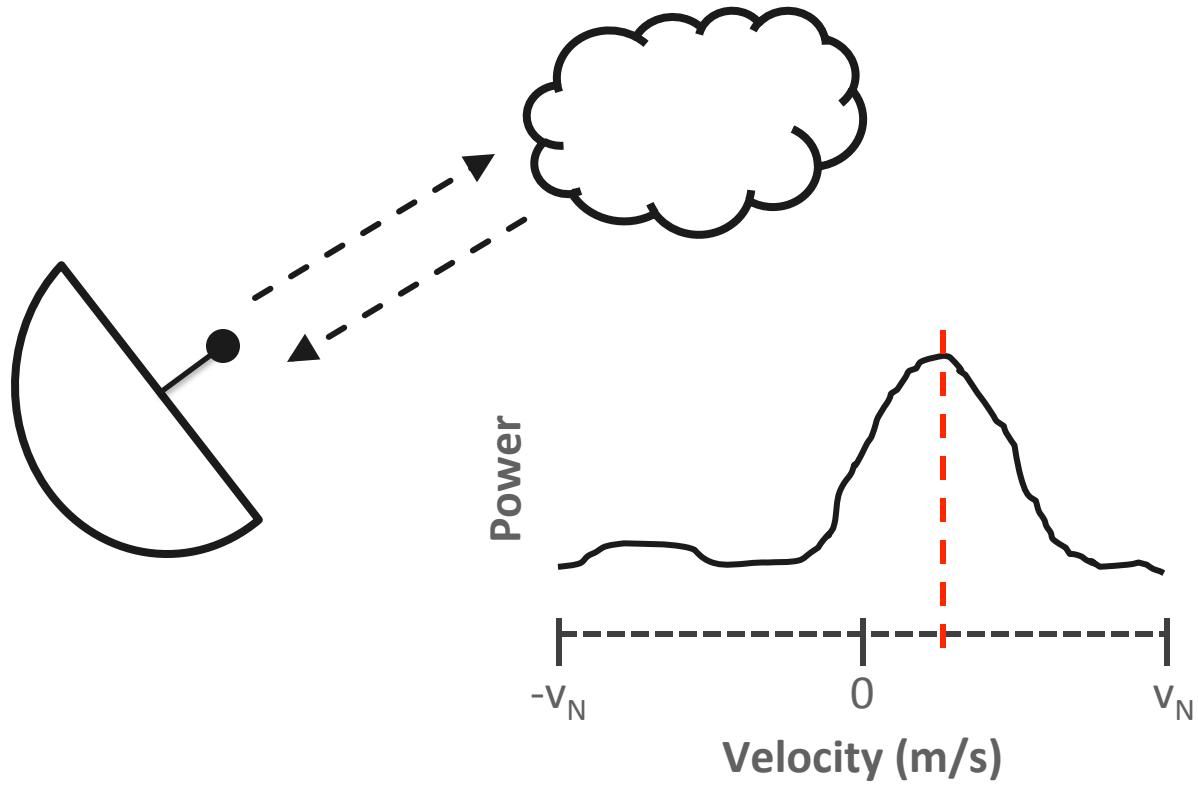


# Wrapping the RSL library: Access in Python

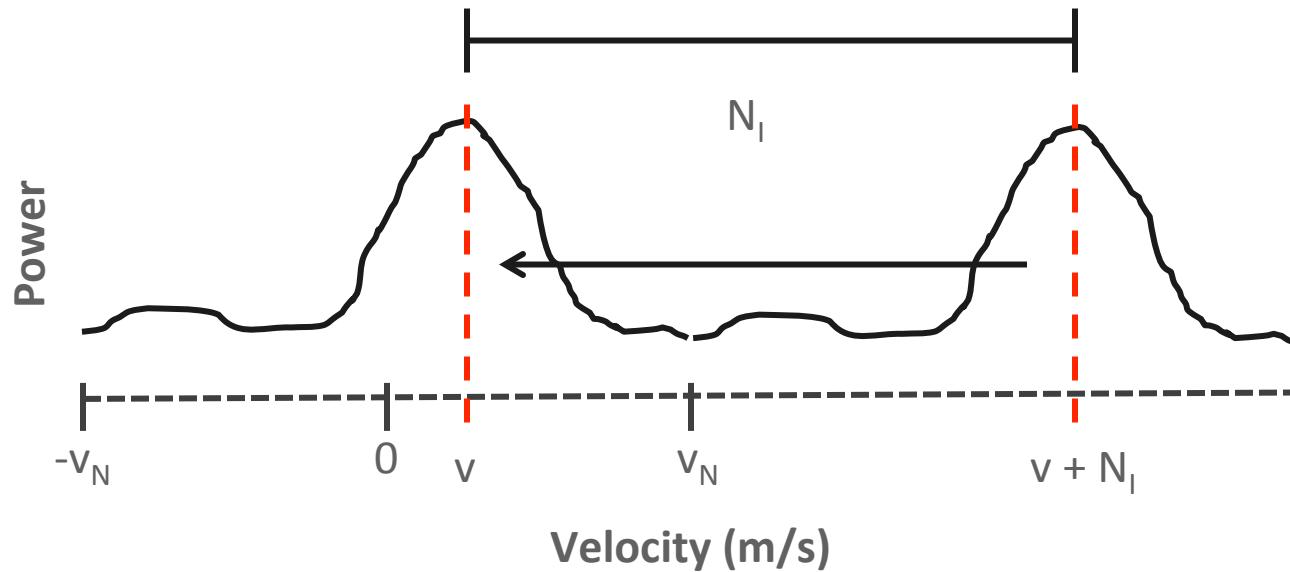
```
>>> from pyart.io import _rsl_interface
>>> rslfile = _rsl_interface.RslFile('XSW110520105408.RAW')

>>> print rslfile
<pyart.io._rsl_interface.RslFile object at 0x107112d20>
>>> print rslfile.month
5
>>> rslfile.month = 12
>>> print rslfile.month
12
>>>
>>> volume = rslfile.get_volume(1)
>>> print volume
<pyart.io._rsl_interface._RslVolume object at 0x100493760>
```

# Radial Velocities Measurements by Weather Radar



# Velocity Aliasing



$$v' = v + n \times N_I \quad \text{where } n = 0, \pm 1, \dots$$

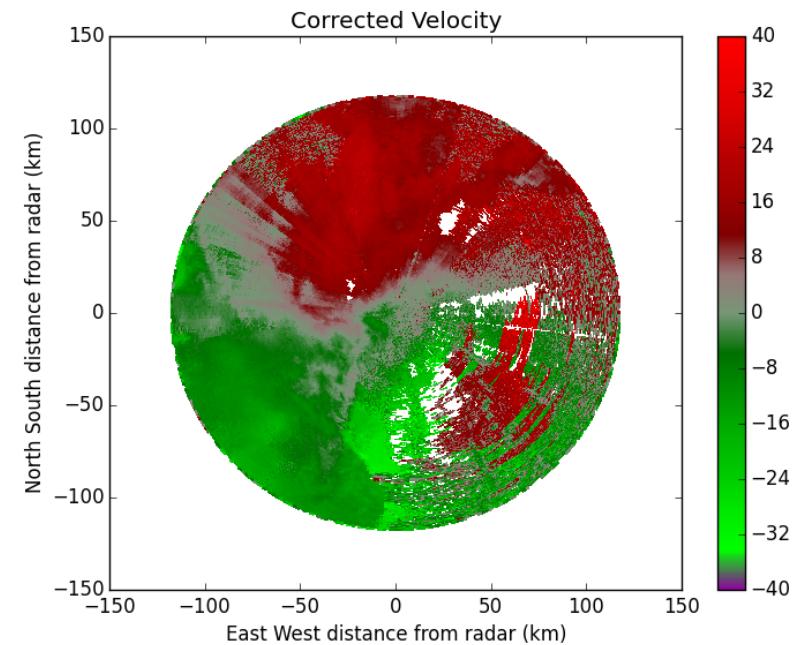
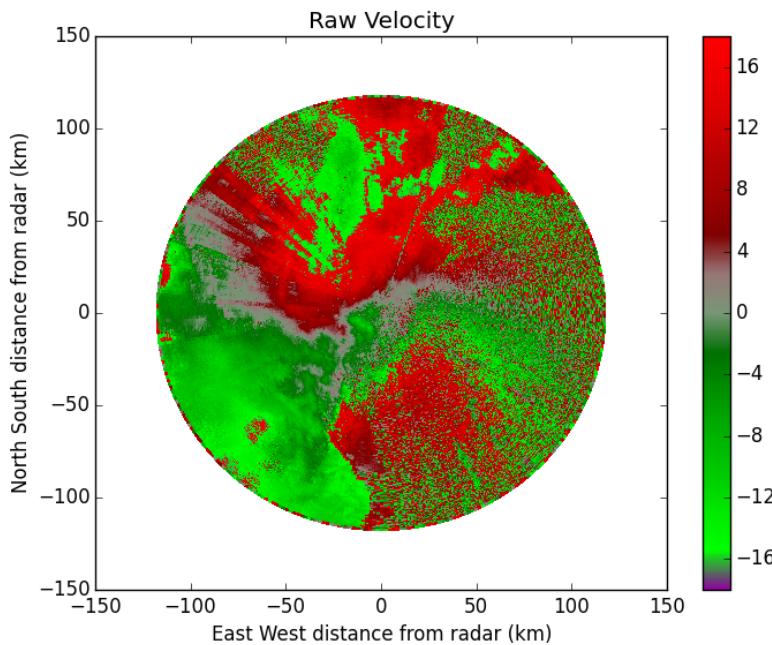
# Four-Dimensional Doppler Dealiasing

## A Real-Time Four-Dimensional Doppler Dealiasing Scheme

CURTIS N. JAMES\* AND ROBERT A. HOUZE JR.

*Department of Atmospheric Sciences, University of Washington, Seattle, Washington*

J Tech, Vol 18, 2001, pg. 1674.



# FourDD Python module: Cython

```
cimport pyart.io._rsl_h as _rsl_h

cdef extern from "dealias_fourdd.h":

    int dealias_fourdd(
        _rsl_h.Volume* rvVolume,
        _rsl_h.Volume* soundVolume,
        _rsl_h.Volume* lastVolume,
        float missingVal, float compthresh,
        float compthresh2, float thresh,
        float ckval, float stdthresh,
        float epsilon, int maxcount,
        int pass2, int rm, int proximity,
        int mingood, int filt,
        int ba_mincount, int ba_edgecount)
```

\_fourdd\_h.pxd

```
cimport _fourdd_h
from pyart.io._rsl_interface cimport _RslVolume

from ..io import _rsl_interface

cpdef fourdd_dealias(
    _RslVolume radialVelVolume, _RslVolume lastVelVolume,
    _RslVolume soundVolume, _RslVolume Dzvolume, ...):
    ...
    # copy the radial velocity data to unfoldedVolume
    unfoldedVolume = _rsl_interface.copy_volume(radialVelVolume)
    ...
    usuccess = _fourdd_h.dealias_fourdd(
        unfoldedVolume._Volume, soundVolume._Volume,
        lastVelVolume._Volume,
        MISSINGVEL, compthresh, compthresh2, thresh,
        ckval, stdthresh, epsilon,
        maxcount, pass2, rm, proximity, mingood,
        filt, ba_mincount, ba_edgecount)
    data = unfoldedVolume.get_data()
    return usuccess, data
```

\_fourdd\_interface.pyx

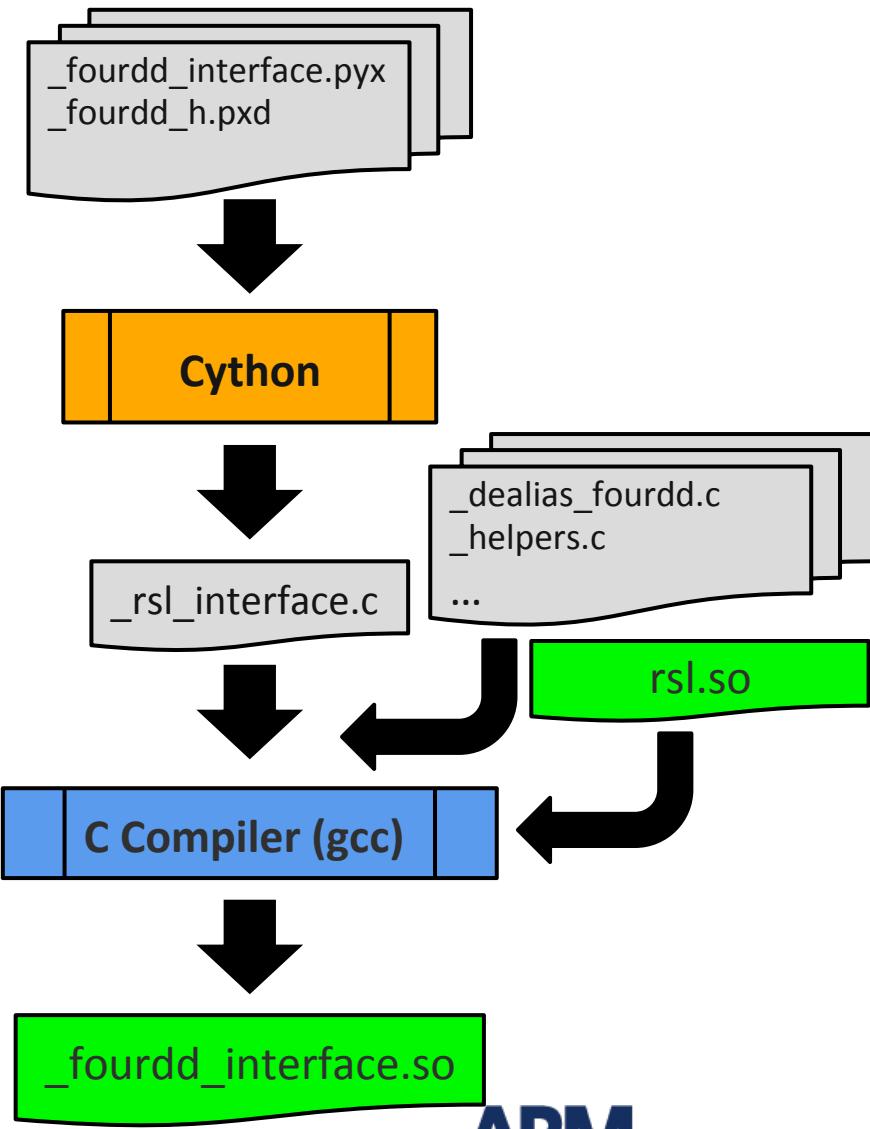


# FourDD Python module: Building

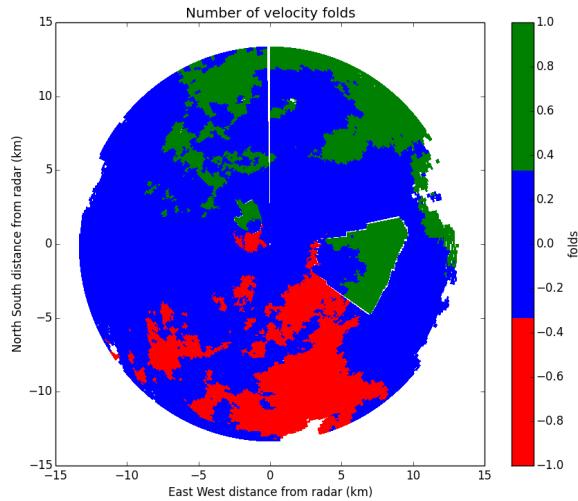
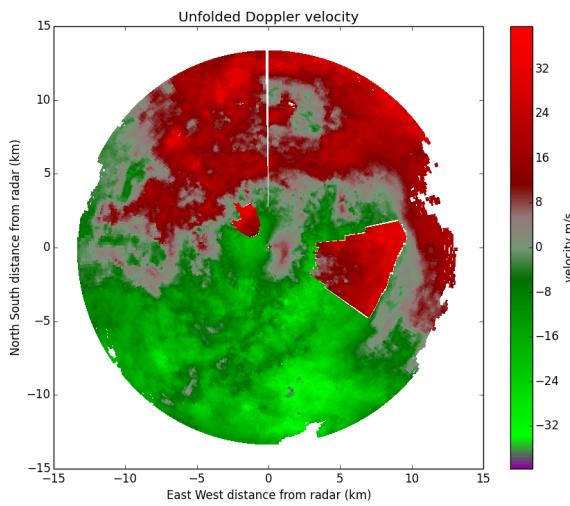
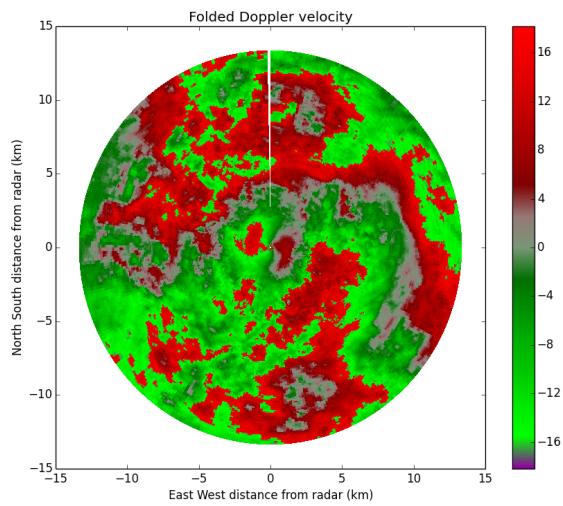
```
# Cython wrapper around FourDD
fourdd_sources = ['src/dealias_fourdd.c',
                  'src/sounding_to_volume.c',
                  'src/filter_by_reflectivity.c',
                  'src/helpers.c']

config.add_extension(
    '_fourdd_interface',
    sources=['_fourdd_interface.c'] + fourdd_sources,
    libraries=['rsl'],
    library_dirs=[rsl_lib_path],
    include_dirs=([rsl_include_path, 'src'] +
                 [get_include()]),
    runtime_library_dirs=[rsl_lib_path])
```

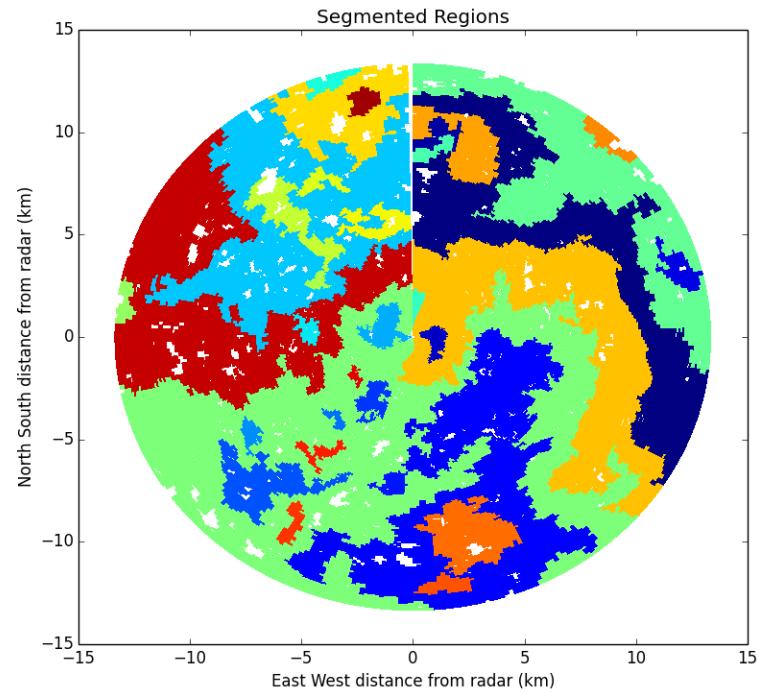
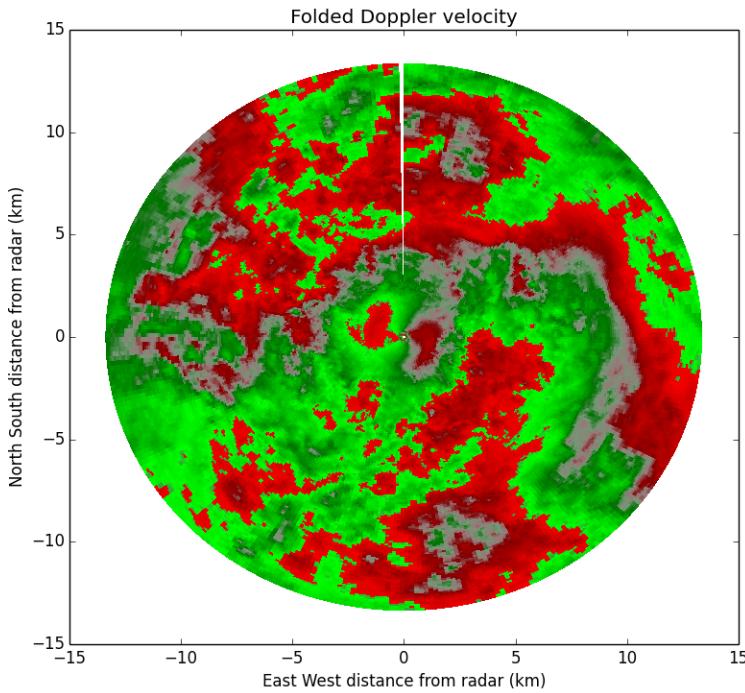
setup.py



# Four-Dimensional Doppler Dealiasing

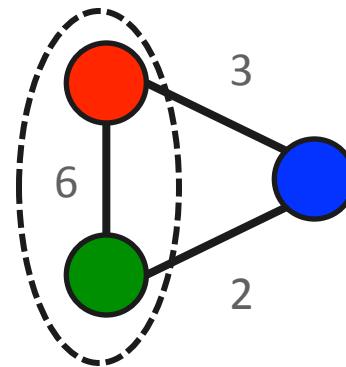
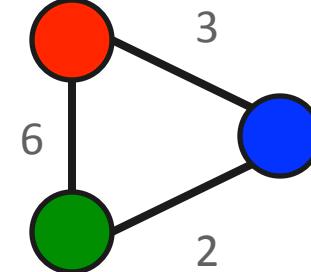
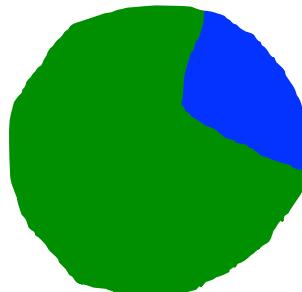
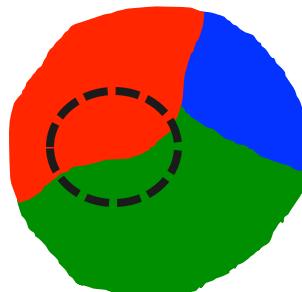
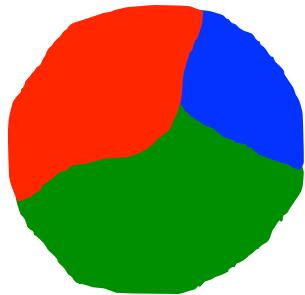


# Region Based Dealiasing : Algorithm Part I

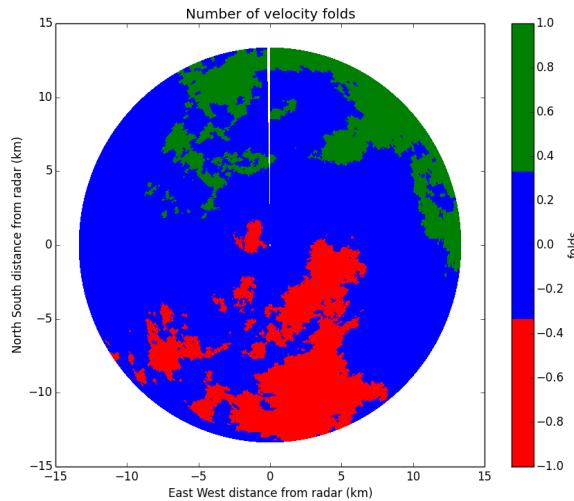
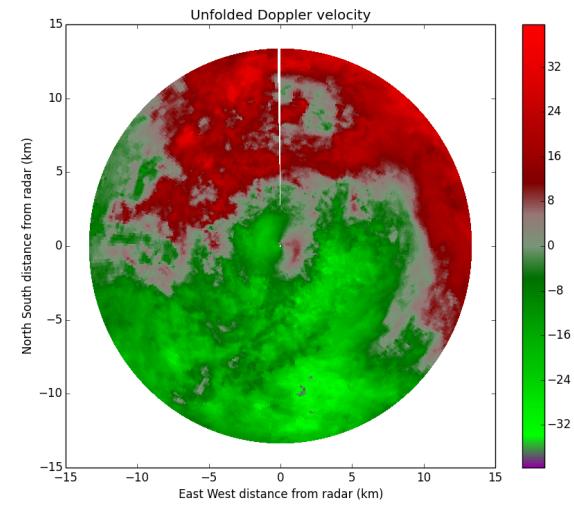
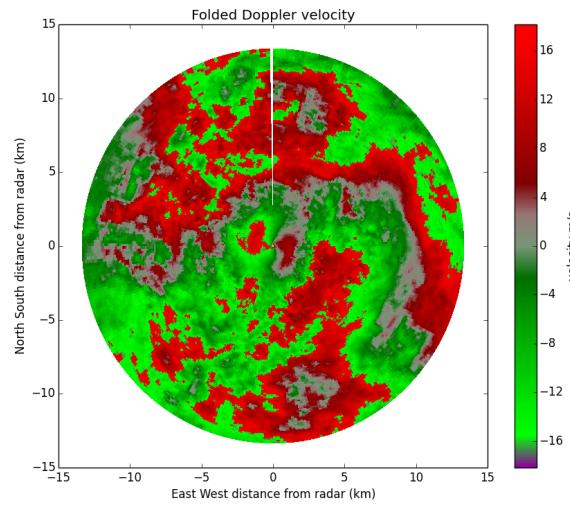


- Segmentation into regions based upon folded velocities.
- Uses `scipy.ndimage` module.

# Region Based Dealiasing : Algorithm Part II



# Region Based Dealiasing : Results on test case



# Dealiasing algorithm profiling with *line\_profiler*

[https://pypi.python.org/pypi/line\\_profiler/](https://pypi.python.org/pypi/line_profiler/)

## Source code

```
@profile
def dealias_region_based(
    radar, interval_splits=3, interval_limits=None,
    skip_between_rays=2, skip_along_ray=2, centered=True,
    nyquist_vel=None, gatefilter=None, rays_wrap_around=None,
    keep_original=True, vel_field=None, corr_vel_field=None, **kwargs):
    """
        Dealias Doppler velocities using a region based algorithm.
    ...

```

## Script

```
import pyart
radar = pyart.io.read('105235.mdv')
pyart.correct.dealias_region_based(radar)
```

# Profiling results

```
$ kernprof.py -l -v script.py
```

```
Total time: 153.644 s
```

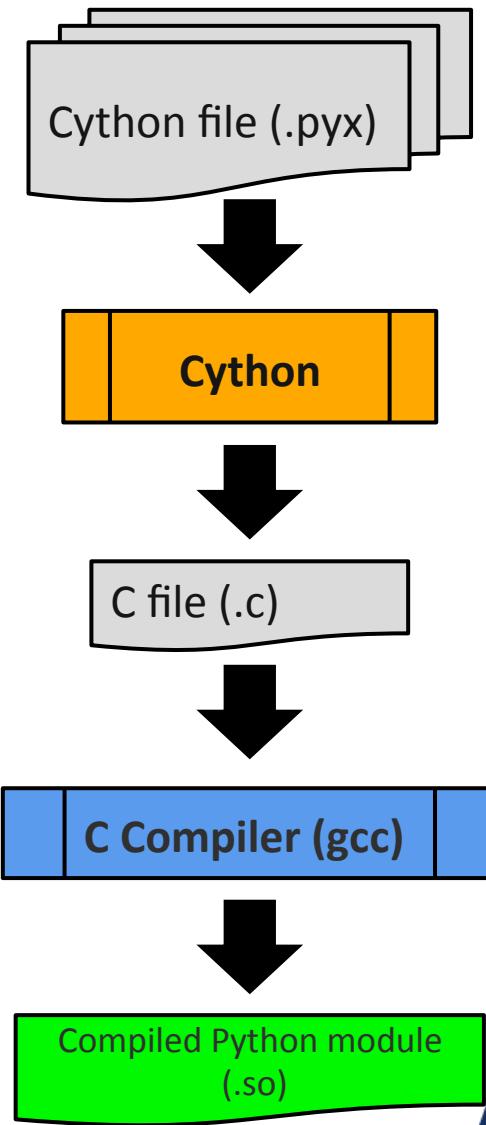
Line #	Hits	Time	Per Hit	% Time	Line Contents
=====					
159					# extract sweep data
160	17	3170	186.5	0.0	sdata = vdata[sweep_slice].copy() # is a copy needed here?
161	17	38	2.2	0.0	scorr = data[sweep_slice]
162	17	28	1.6	0.0	sfilter = gfilter[sweep_slice]
163					
164					# find regions in original data
165	17	284246	16720.4	0.2	labels, nfeatures = _find_regions(sdata, sfilter, interval_limits)
166	17	43	2.5	0.0	edge_sum, edge_count, region_sizes = _edge_sum_and_count(
167	17	19	1.1	0.0	labels, nfeatures, sdata, rays_wrap_around, skip_between_rays,
168	17	107524384	6324963.8	70.0	skip_along_ray)
169					
170					# find the number of folds in the regions
171	17	58564	3444.9	0.0	region_tracker = _RegionTracker(region_sizes)
172	17	33254618	1956154.0	21.6	edge_tracker = _EdgeTracker(edge_sum, edge_count, nyquist_interval)
173	48314	56613	1.2	0.0	while True:
174	48314	6727992	139.3	4.4	if _combine_regions(region_tracker, edge_tracker):
175	17	19	1.1	0.0	break

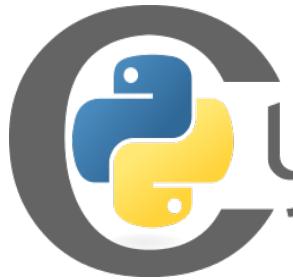
# Dealiasing algorithm optimization using Cython

Python code, especially when it contains nested loops, can be slow. One method to speed up the run time is to use Cython.

## Cython

- Python to C code translator.
- Generates a Python extension module.
- Can be used to speed up Python code by adding static type information.
- Code produced by Cython can rival the speed of compiled code yet is typically easier to understand and modify.



The Python logo icon is a dark gray circle containing a white outline of the Python logo symbol, which consists of two interlocking snakes.

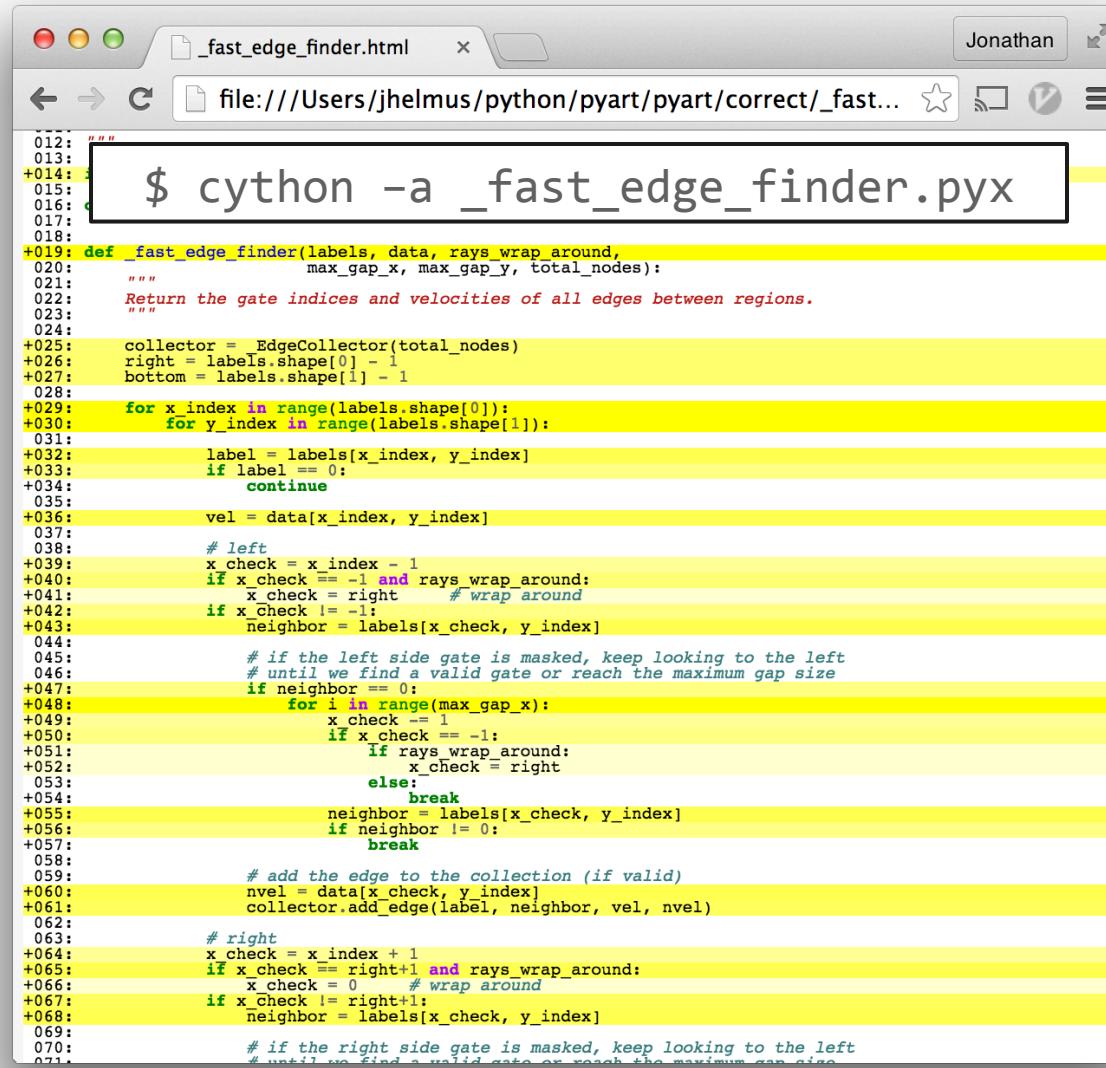
# Python demo

Adding type information to the region based dealiasing algorithm



Profiling Python code ..., Jonathan Helmus, SEA Conference 2015, Boulder, CO

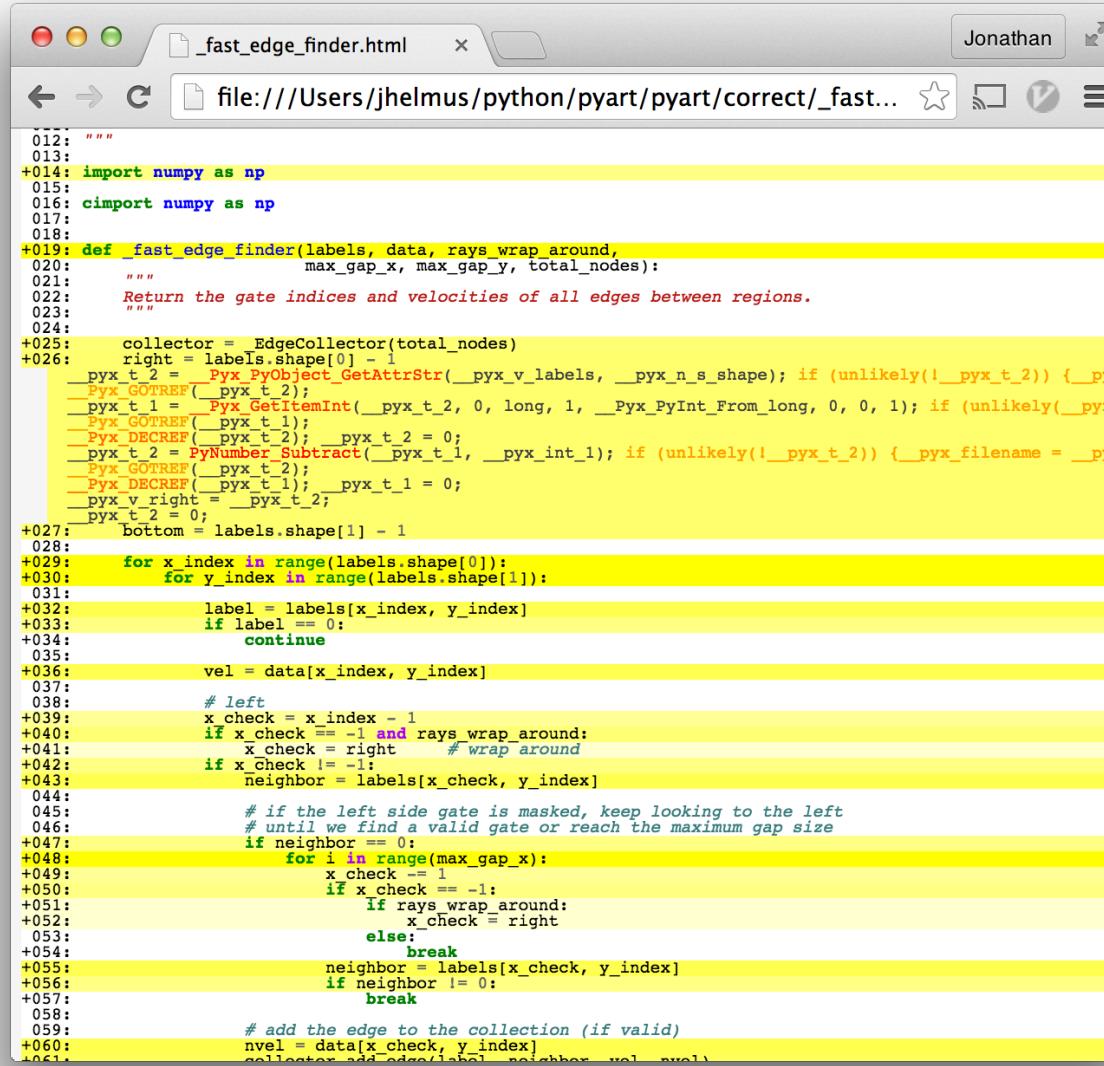
# Optimizing Cython code



\$ cython -a \_fast\_edge\_finder.pyx

```
012: """
013:
014: +
015: def _fast_edge_finder(labels, data, rays_wrap_around,
016:                       max_gap_x, max_gap_y, total_nodes):
017:     """
018:     """Return the gate indices and velocities of all edges between regions.
019:     """
020:     collector = EdgeCollector(total_nodes)
021:     right = labels.shape[0] - 1
022:     bottom = labels.shape[1] - 1
023:
024:     for x_index in range(labels.shape[0]):
025:         for y_index in range(labels.shape[1]):
026:             label = labels[x_index, y_index]
027:             if label == 0:
028:                 continue
029:             vel = data[x_index, y_index]
030:
031:             # left
032:             x_check = x_index - 1
033:             if x_check == -1 and rays_wrap_around:
034:                 x_check = right # wrap around
035:             if x_check != -1:
036:                 neighbor = labels[x_check, y_index]
037:
038:                 # if the left side gate is masked, keep looking to the left
039:                 # until we find a valid gate or reach the maximum gap size
040:                 if neighbor == 0:
041:                     for i in range(max_gap_x):
042:                         x_check -= 1
043:                         if x_check == -1:
044:                             If rays_wrap_around:
045:                                 x_check = right
046:                             else:
047:                                 break
048:                         neighbor = labels[x_check, y_index]
049:                         if neighbor != 0:
050:                             break
051:
052:                 # add the edge to the collection (if valid)
053:                 nvel = data[x_check, y_index]
054:                 collector.add_edge(label, neighbor, vel, nvel)
055:
056:             # right
057:             x_check = x_index + 1
058:             if x_check == right+1 and rays_wrap_around:
059:                 x_check = 0 # wrap around
060:             if x_check != right+1:
061:                 neighbor = labels[x_check, y_index]
062:
063:                 # if the right side gate is masked, keep looking to the left
064:                 # until we find a valid gate or reach the maximum gap size
065:                 if neighbor == 0:
```

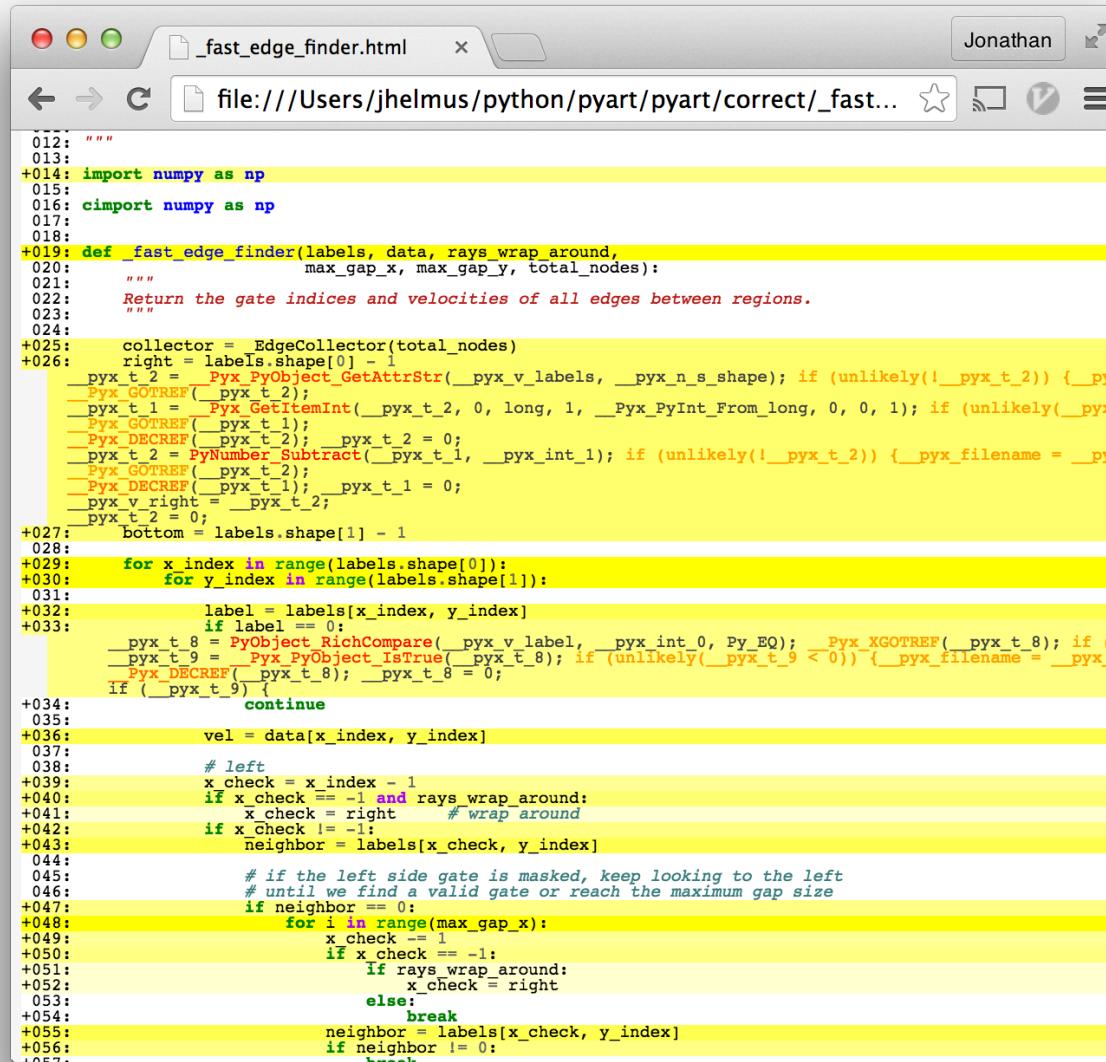
# Optimizing Cython code



The screenshot shows a web browser window with the title '\_fast\_edge\_finder.html'. The URL bar displays 'file:///Users/jhelmus/python/pyart/pyart/correct/\_fast...'. The page content is a block of Cython code. The code is annotated with numerous orange and yellow highlights, primarily around memory management and performance-critical sections. The code defines a function '\_fast\_edge\_finder' that takes 'labels', 'data', and 'rays\_wrap\_around' as parameters. It uses NumPy arrays and an EdgeCollector to find edges between regions based on labels and data. The highlighted sections include Pyrex object creation, memory deallocation, and loop iterations.

```
012: """
013:
014: import numpy as np
015:
016: cimport numpy as np
017:
018:
019: def _fast_edge_finder(labels, data, rays_wrap_around,
020:                         max_gap_x, max_gap_y, total_nodes):
021:     """
022:         Return the gate indices and velocities of all edges between regions.
023:
024:
025:         collector = EdgeCollector(total_nodes)
026:         right = labels.shape[0] - 1
027:         pyc_t_2 = __Pyx_PyObject_GetAttrStr(__pyx_v_labels, __pyx_n_s_shape); if (unlikely(!__pyx_t_2)) {__pyx_t_2 = __Pyx_GOTREF(__pyx_t_2);
028:             Pyx_DECREF(__pyx_t_2);
029:             pyc_t_1 = __Pyx_GetItemInt(__pyx_t_2, 0, long, 1, __Pyx_PyInt_From_long, 0, 0, 1); if (unlikely(!__pyx_t_1)) {__pyx_t_1 = __Pyx_GOTREF(__pyx_t_1);
030:                 Pyx_DECREF(__pyx_t_2);
031:                 pyc_t_2 = PyNumber_Subtract(__pyx_t_1, __pyx_int_1); if (unlikely(!__pyx_t_2)) {__pyx_filename = __pyx_t_1;
032:                     __Pyx_DECREF(__pyx_t_2);
033:                     Pyx_DECREF(__pyx_t_1);
034:                     __pyx_t_1 = 0;
035:                     __pyx_v_right = __pyx_t_2;
036:                     __pyx_t_2 = 0;
037:                     bottom = labels.shape[1] - 1
038:                     for x_index in range(labels.shape[0]):
039:                         for y_index in range(labels.shape[1]):
040:                             label = labels[x_index, y_index]
041:                             if label == 0:
042:                                 continue
043:                             vel = data[x_index, y_index]
044:                             # left
045:                             x_check = x_index - 1
046:                             if x_check == -1 and rays_wrap_around:
047:                                 x_check = right #wrap around
048:                             if x_check != -1:
049:                                 neighbor = labels[x_check, y_index]
050:                                 for i in range(max_gap_x):
051:                                     x_check -= 1
052:                                     if x_check == -1:
053:                                         If rays_wrap_around:
054:                                             x_check = right
055:                                         else:
056:                                             break
057:                                         neighbor = labels[x_check, y_index]
058:                                         if neighbor != 0:
059:                                             break
060:                                         # add the edge to the collection (if valid)
061:                                         nvel = data[x_check, y_index]
```

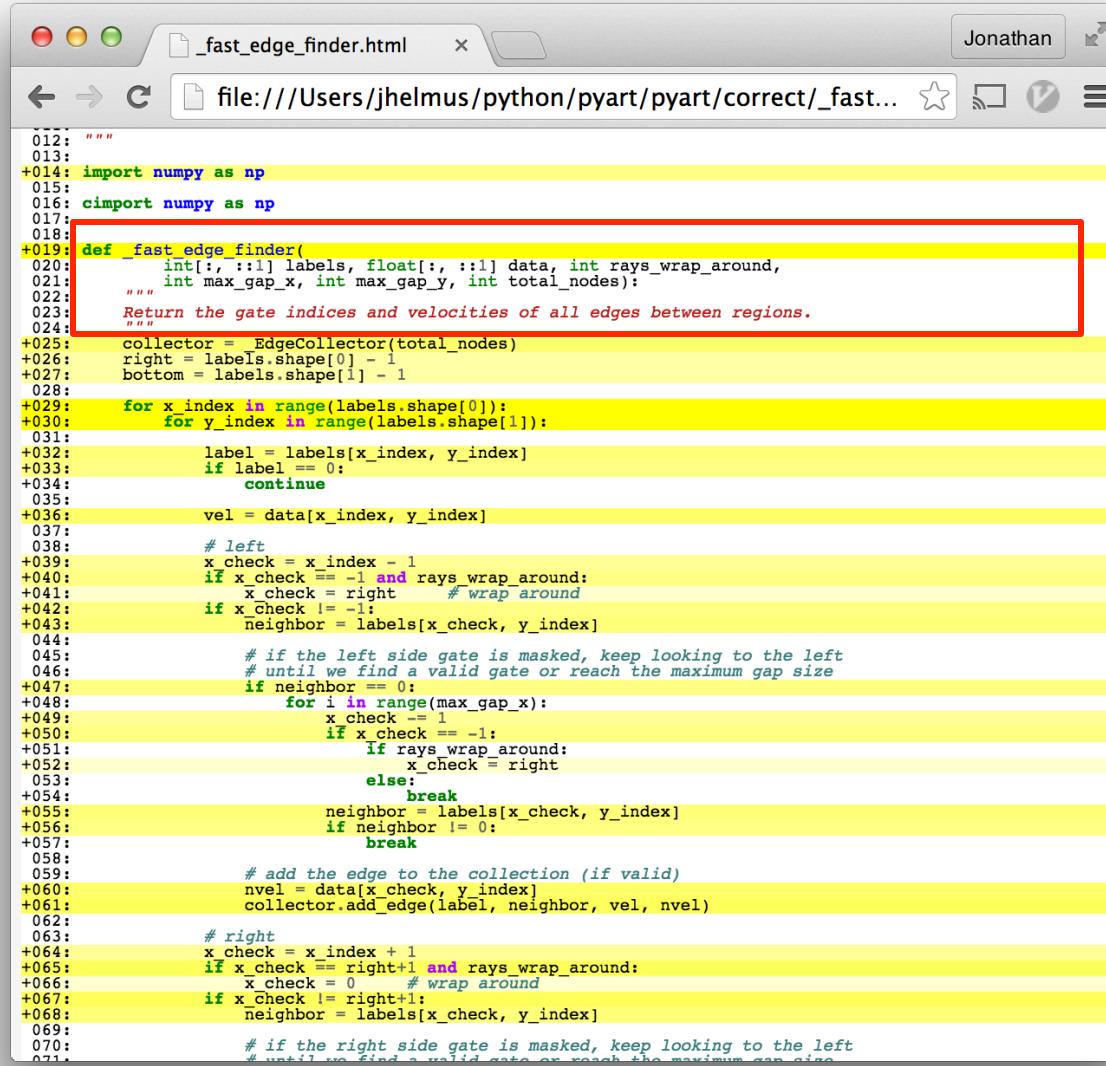
# Optimizing Cython code



The screenshot shows a web browser window with the title '\_fast\_edge\_finder.html'. The URL bar displays 'file:///Users/jhelmus/python/pyart/pyart/correct/\_fast...'. The page content is a block of Cython code. The code is annotated with numerous orange and yellow annotations, likely from a static analysis tool like Cython's own optimizer or a third-party tool like Cythonify. These annotations highlight various performance-critical sections, such as memory allocations, deallocations, and comparisons. The code itself is a function named '\_fast\_edge\_finder' that takes 'labels', 'data', and 'rays\_wrap\_around' as parameters, returning edge indices and velocities.

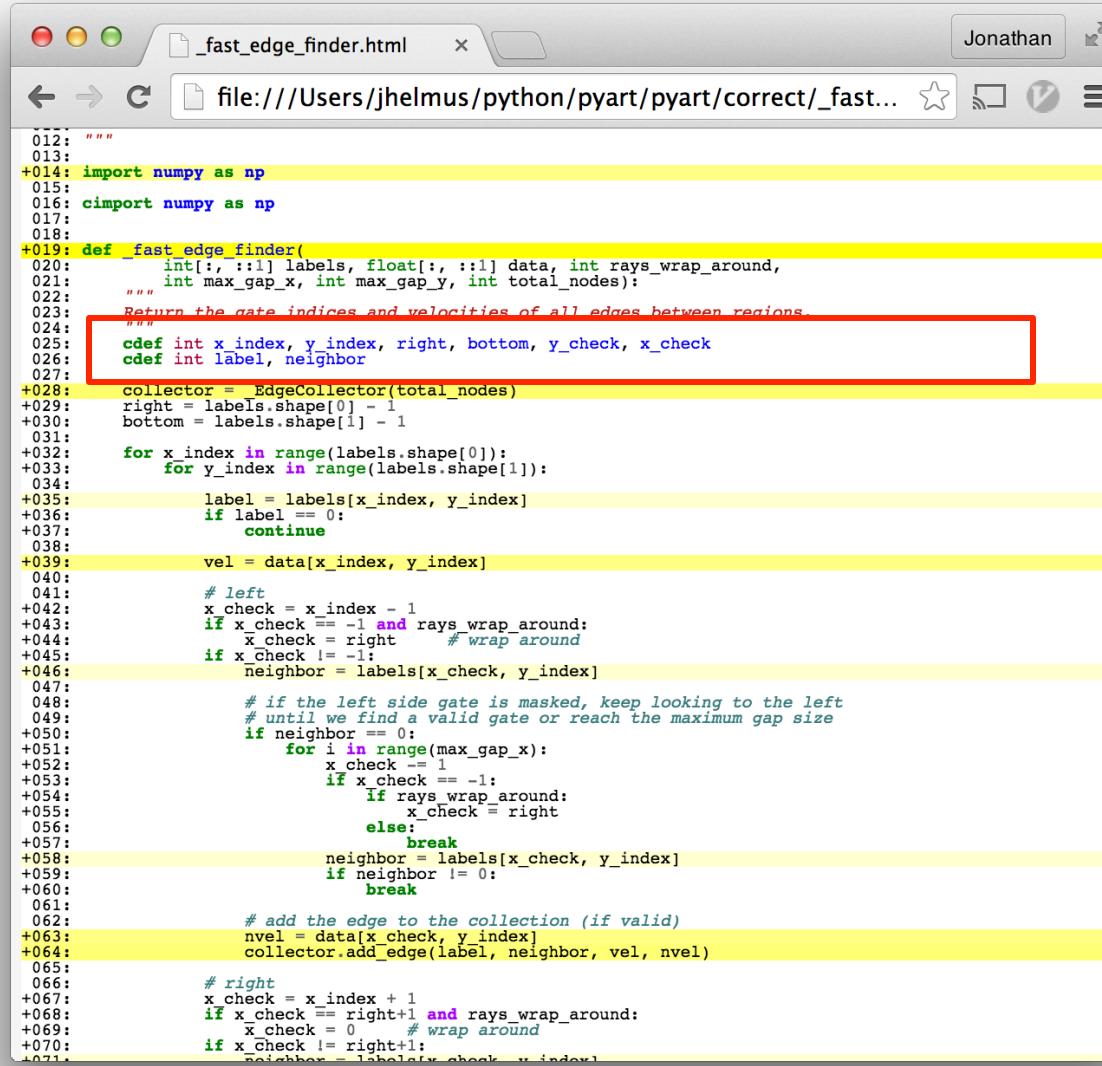
```
012: """
013:
+014: import numpy as np
015:
016: cimport numpy as np
017:
018:
+019: def _fast_edge_finder(labels, data, rays_wrap_around,
020:                         max_gap_x, max_gap_y, total_nodes):
021:     """
022:         Return the gate indices and velocities of all edges between regions.
023:
024:
+025:     collector = EdgeCollector(total_nodes)
026:     right = labels.shape[0] - 1
+027:     Pyx_t_2 = Pyx_PyObject_GetAttrStr(_pyx_v_labels, _pyx_n_s_shape); if (unlikely(!_pyx_t_2)) {__pyx_t_2 = Pyx_GOTREF(_pyx_t_2);
+028:     Pyx_t_1 = Pyx_GetItemInt(_pyx_t_2, 0, long, 1, __Pyx_PyInt_From_long, 0, 0, 1); if (unlikely(_pyx_t_1 == Pyx_DECREF(_pyx_t_2)); __pyx_t_2 = 0;
+029:     Pyx_number_subtract(_pyx_t_1, __pyx_int_1); if (unlikely(!_pyx_t_2)) {__pyx_filename = __pyx_t_2 = Pyx_GOTREF(_pyx_t_2);
+030:     Pyx_DECREF(_pyx_t_1); __pyx_t_1 = 0;
+031:     Pyx_v_right = __pyx_t_2;
+032:     __pyx_t_2 = 0;
+033:     bottom = labels.shape[1] - 1
+034:     for x_index in range(labels.shape[0]):
+035:         for y_index in range(labels.shape[1]):
+036:             label = labels[x_index, y_index]
+037:             if label == 0:
+038:                 continue
+039:             vel = data[x_index, y_index]
+040:             # left
+041:             x_check = x_index - 1
+042:             if x_check == -1 and rays_wrap_around:
+043:                 x_check = right # wrap around
+044:             if x_check != -1:
+045:                 neighbor = labels[x_check, y_index]
+046:                 # if the left side gate is masked, keep looking to the left
+047:                 # until we find a valid gate or reach the maximum gap size
+048:                 if neighbor == 0:
+049:                     for i in range(max_gap_x):
+050:                         x_check -= 1
+051:                         if x_check == -1:
+052:                             if rays_wrap_around:
+053:                                 x_check = right
+054:                             else:
+055:                                 break
+056:                         neighbor = labels[x_check, y_index]
+057:                         if neighbor != 0:
```

# Optimizing Cython code



```
012: """
013:
+014: import numpy as np
015:
016: cimport numpy as np
017:
018:
+019: def _fast_edge_finder(
020:     int[:, ::1] labels, float[:, ::1] data, int rays_wrap_around,
021:     int max_gap_x, int max_gap_y, int total_nodes):
022: """
023: Return the gate indices and velocities of all edges between regions.
024:
025:     collector = EdgeCollector(total_nodes)
026:     right = labels.shape[0] - 1
027:     bottom = labels.shape[1] - 1
028:
029:     for x_index in range(labels.shape[0]):
030:         for y_index in range(labels.shape[1]):
031:
032:             label = labels[x_index, y_index]
033:             if label == 0:
034:                 continue
035:
036:             vel = data[x_index, y_index]
037:
038:             # left
039:             x_check = x_index - 1
040:             if x_check == -1 and rays_wrap_around:
041:                 x_check = right # wrap around
042:             if x_check != -1:
043:                 neighbor = labels[x_check, y_index]
044:
045:                 # if the left side gate is masked, keep looking to the left
046:                 # until we find a valid gate or reach the maximum gap size
047:             if neighbor == 0:
048:                 for i in range(max_gap_x):
049:                     x_check -= 1
050:                     if x_check == -1:
051:                         If rays_wrap_around:
052:                             x_check = right
053:                         else:
054:                             break
055:                     neighbor = labels[x_check, y_index]
056:                     if neighbor != 0:
057:                         break
058:
059:                 # add the edge to the collection (if valid)
060:                 nvel = data[x_check, y_index]
061:                 collector.add_edge(label, neighbor, vel, nvel)
062:
063:             # right
064:             x_check = x_index + 1
065:             if x_check == right+1 and rays_wrap_around:
066:                 x_check = 0 # wrap around
067:             if x_check != right+1:
068:                 neighbor = labels[x_check, y_index]
069:
070:                 # if the right side gate is masked, keep looking to the left
071:                 # until we find a valid gate or reach the maximum gap size
```

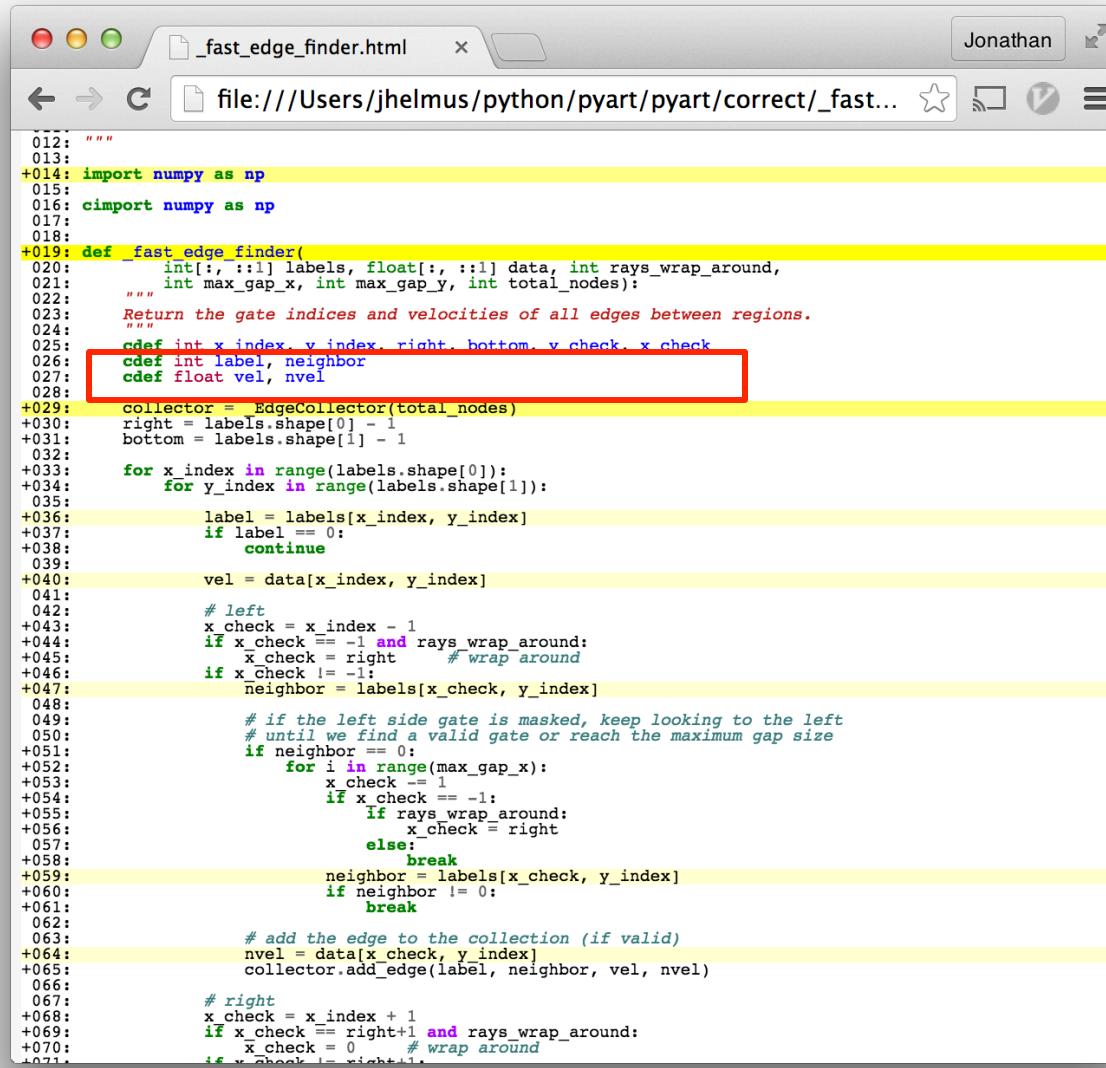
# Optimizing Cython code



The screenshot shows a web browser window with the title '\_fast\_edge\_finder.html'. The URL in the address bar is 'file:///Users/jhelmus/python/pyart/pyart/correct/\_fast...'. The page content displays a block of Python code with syntax highlighting. A red rectangular box highlights a specific section of the code, which includes several C-style type annotations ('cdef') and variable declarations ('label', 'neighbor'). The code is a Cython implementation of an edge finder function, likely for a scientific application involving grid data.

```
012: """
013:
+014: import numpy as np
015:
016: cimport numpy as np
017:
018:
+019: def _fast_edge_finder(
020:     int[:, ::1] labels, float[:, ::1] data, int rays_wrap_around,
021:     int max_gap_x, int max_gap_y, int total_nodes):
022:
023:     """
024:     Return the gate indices and velocities of all edges between regions.
025:
026:     cdef int x_index, y_index, right, bottom, y_check, x_check
027:
+028:     collector = EdgeCollector(total_nodes)
+029:     right = labels.shape[0] - 1
+030:     bottom = labels.shape[1] - 1
031:
+032:     for x_index in range(labels.shape[0]):
+033:         for y_index in range(labels.shape[1]):
034:
+035:             label = labels[x_index, y_index]
036:             if label == 0:
037:                 continue
038:
+039:             vel = data[x_index, y_index]
040:
041:             # left
+042:             x_check = x_index - 1
+043:             if x_check == -1 and rays_wrap_around:
+044:                 x_check = right    # wrap around
+045:             if x_check != -1:
+046:                 neighbor = labels[x_check, y_index]
047:
048:                 # if the left side gate is masked, keep looking to the left
049:                 # until we find a valid gate or reach the maximum gap size
+050:                 if neighbor == 0:
+051:                     for i in range(max_gap_x):
+052:                         x_check = 1
+053:                         if x_check == -1:
+054:                             if rays_wrap_around:
+055:                                 x_check = right
+056:                             else:
+057:                                 break
+058:                         neighbor = labels[x_check, y_index]
+059:                         if neighbor != 0:
+060:                             break
061:
062:                 # add the edge to the collection (if valid)
+063:                 nvel = data[x_check, y_index]
+064:                 collector.add_edge(label, neighbor, vel, nvel)
065:
066:             # right
+067:             x_check = x_index + 1
+068:             if x_check == right+1 and rays_wrap_around:
+069:                 x_check = 0    # wrap around
+070:             if x_check != right+1:
+071:                 neighbor = labels[x_check, y_index]
```

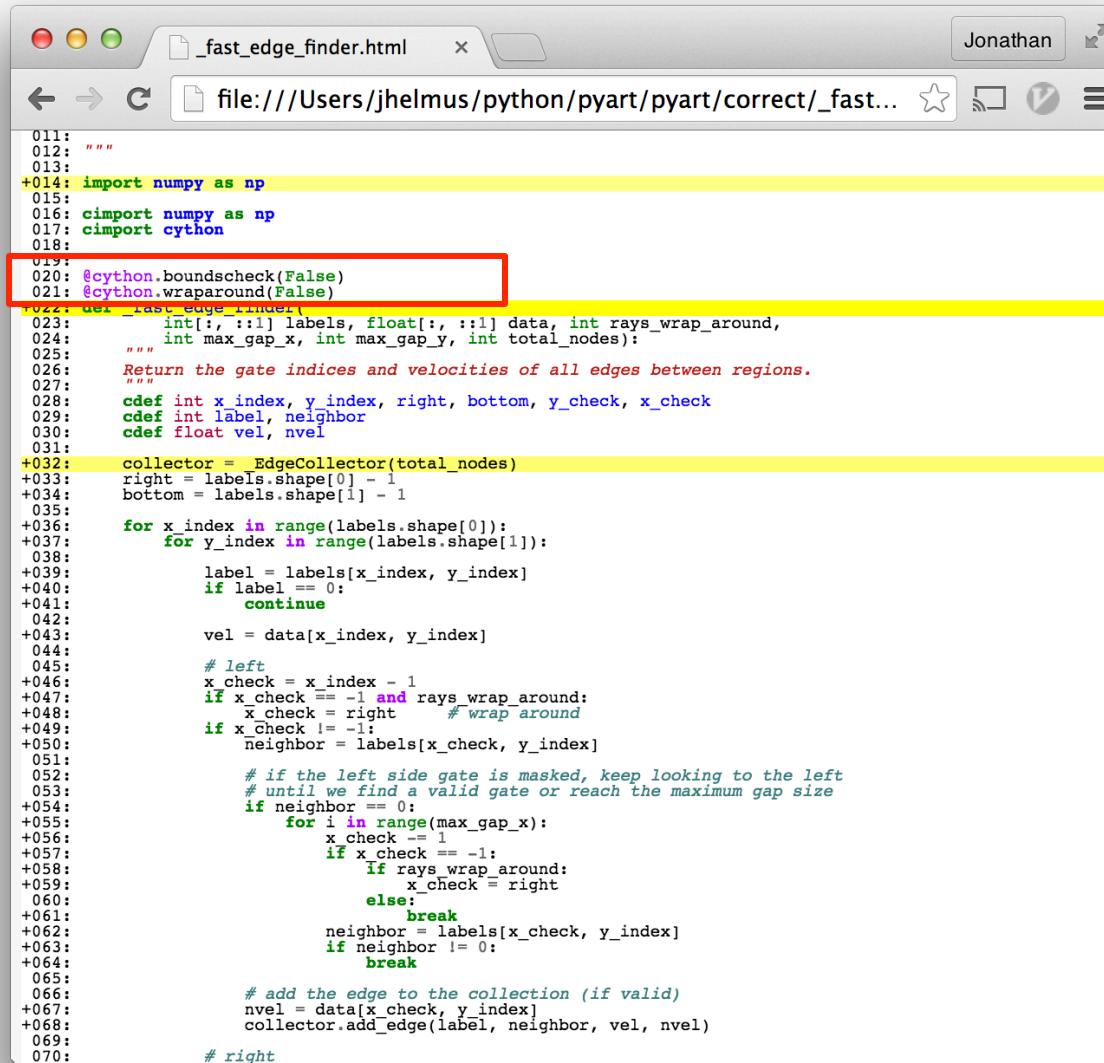
# Optimizing Cython code



The screenshot shows a web browser window with the title '\_fast\_edge\_finder.html'. The URL in the address bar is 'file:///Users/jhelmus/python/pyart/pyart/correct/\_fast...'. The page content displays a block of Python code with Cython annotations. A red box highlights a specific section of the code:

```
+012: """
+013:
+014: import numpy as np
+015:
+016: cimport numpy as np
+017:
+018:
+019: def _fast_edge_finder(
+020:     int[:, ::1] labels, float[:, ::1] data, int rays_wrap_around,
+021:     int max_gap_x, int max_gap_y, int total_nodes):
+022: """
+023:     Return the gate indices and velocities of all edges between regions.
+024:
+025:     cdef int x_index, y_index, right, bottom, v_check, x_check
+026:     cdef int label, neighbor
+027:     cdef float vel, nvel
+028:
+029:     collector = EdgeCollector(total_nodes)
+030:     right = labels.shape[0] - 1
+031:     bottom = labels.shape[1] - 1
+032:
+033:     for x_index in range(labels.shape[0]):
+034:         for y_index in range(labels.shape[1]):
+035:
+036:             label = labels[x_index, y_index]
+037:             if label == 0:
+038:                 continue
+039:
+040:             vel = data[x_index, y_index]
+041:
+042:             # left
+043:             x_check = x_index - 1
+044:             if x_check == -1 and rays_wrap_around:
+045:                 x_check = right      # wrap around
+046:             if x_check != -1:
+047:                 neighbor = labels[x_check, y_index]
+048:
+049:                 # if the left side gate is masked, keep looking to the left
+050:                 # until we find a valid gate or reach the maximum gap size
+051:                 if neighbor == 0:
+052:                     for i in range(max_gap_x):
+053:                         x_check -= 1
+054:                         if x_check == -1:
+055:                             if rays_wrap_around:
+056:                                 x_check = right
+057:                             else:
+058:                                 break
+059:                         neighbor = labels[x_check, y_index]
+060:                         if neighbor != 0:
+061:                             break
+062:
+063:                         # add the edge to the collection (if valid)
+064:                         nvel = data[x_check, y_index]
+065:                         collector.add_edge(label, neighbor, vel, nvel)
+066:
+067:             # right
+068:             x_check = x_index + 1
+069:             if x_check == right+1 and rays_wrap_around:
+070:                 x_check = 0      # wrap around
+071:                 if x_check == right+1:
```

# Optimizing Cython code



The screenshot shows a web browser window displaying a file:///Users/jhelmus/python/pyart/pyart/correct/\_fast... page. The code is written in Cython and is annotated with line numbers. A red box highlights the first few lines of the code:

```
011: """
012: """
013:
+014: import numpy as np
015:
016: cimport numpy as np
017: cimport cython
018:
019: #cython.boundscheck(False)
020: #cython.wraparound(False)
021:
022: def __fast_edge_finder(
023:     int[:, ::1] labels, float[:, ::1] data, int rays_wrap_around,
024:     int max_gap_x, int max_gap_y, int total_nodes):
025:     """
026:     Return the gate indices and velocities of all edges between regions.
027:     """
028:     cdef int x_index, y_index, right, bottom, y_check, x_check
029:     cdef int label, neighbor
030:     cdef float vel, nvel
031:
+032:     collector = EdgeCollector(total_nodes)
+033:     right = labels.shape[0] - 1
+034:     bottom = labels.shape[1] - 1
+035:
+036:     for x_index in range(labels.shape[0]):
+037:         for y_index in range(labels.shape[1]):
+038:
+039:             label = labels[x_index, y_index]
+040:             if label == 0:
+041:                 continue
+042:
+043:             vel = data[x_index, y_index]
+044:
+045:             # left
+046:             x_check = x_index - 1
+047:             if x_check == -1 and rays_wrap_around:
+048:                 x_check = right      # wrap around
+049:             if x_check != -1:
+050:                 neighbor = labels[x_check, y_index]
+051:
+052:                 # if the left side gate is masked, keep looking to the left
+053:                 # until we find a valid gate or reach the maximum gap size
+054:                 if neighbor == 0:
+055:                     for i in range(max_gap_x):
+056:                         x_check -= 1
+057:                         if x_check == -1:
+058:                             if rays_wrap_around:
+059:                                 x_check = right
+060:                             else:
+061:                                 break
+062:                         neighbor = labels[x_check, y_index]
+063:                         if neighbor != 0:
+064:                             break
+065:
+066:                 # add the edge to the collection (if valid)
+067:                 nvel = data[x_check, y_index]
+068:                 collector.add_edge(label, neighbor, vel, nvel)
+069:
+070:             # right
```

# \_fast\_edge\_finder.pyx source code : \_fast\_edge\_finder

```
@cython.boundscheck(False)
@cython.wraparound(False)
def _fast_edge_finder(
    int[:, ::1] labels, float[:, ::1] data, int rays_wrap_around,
    int max_gap_x, int max_gap_y, int total_nodes):

    cdef int x_index, right, x_check
    cdef int label, neighbor
    cdef float vel, nvel

    collector = _EdgeCollector(total_nodes)
    right = labels.shape[0] - 1

    for x_index in range(labels.shape[0]):
        for y_index in range(labels.shape[1]):
            label = labels[x_index, y_index]
            if label == 0:
                continue
            vel = data[x_index, y_index]

            # Left
            x_check = x_index - 1
            if x_check == -1 and rays_wrap_around:
                x_check = right      # wrap around
            if x_check != -1:
                neighbor = labels[x_check, y_index]
                nvel = data[x_check, y_index]
            ...
            # add the edge to the collection (if valid)
            collector.add_edge(label, neighbor, vel, nvel)
```

\_fast\_edge\_finder.pyx

# \_fast\_edge\_finder.pyx source code : \_EdgeCollector

```
# Cython implementation inspired by coo_entries in scipy/spatial/ckdtree.pyx
cdef class _EdgeCollector:

    cdef np.ndarray l_index, n_index, l_velo, n_velo
    cdef np.int32_t *l_data
    cdef np.int32_t *n_data
    cdef np.float64_t *lv_data
    cdef np.float64_t *nv_data
    cdef int idx

    def __init__(self, total_nodes):
        """ initialize.
        """
        self.l_index = np.zeros(total_nodes * 4, dtype=np.int32)
        ...
        self.l_data = <np.int32_t *>np.PyArray_DATA(self.l_index)
        ...
        self.idx = 0

    cdef int add_edge(_EdgeCollector self, int label, int neighbor,
                      float vel, float nvel):
        """ Add an edge.
        """
        if neighbor == label or neighbor == 0:
            return 0
        self.l_data[self.idx] = label
        self.n_data[self.idx] = neighbor
        self.lv_data[self.idx] = vel
        self.nv_data[self.idx] = nvel
        self.idx += 1
        return 1
```

\_fast\_edge\_finder.pyx

# Optimization algorithm in Cython

```
$ kernprof.py -l -v script.py
```

Total time: 155.869 s

Line #	Hits	Time	Per Hit	% Time	Line Contents
--------	------	------	---------	--------	---------------

=====

```
...
164                                # find regions in original data
165    17      347781  20457.7      0.2
166    17        46      2.7      0.0
167    17        20      1.2      0.0
168    17  107326070  6313298.2     68.9
```

Total time: 49.5179 s

Line #	Hits	Time	Per Hit	% Time	Line Contents
--------	------	------	---------	--------	---------------

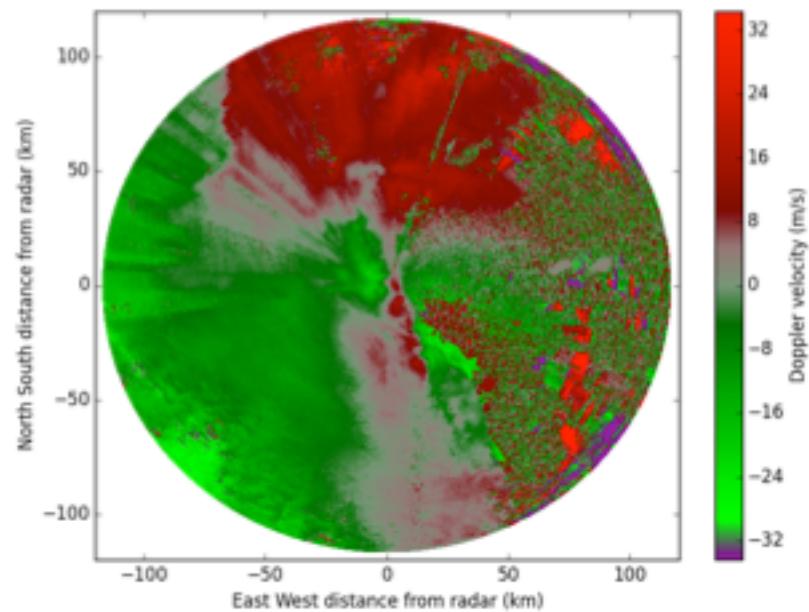
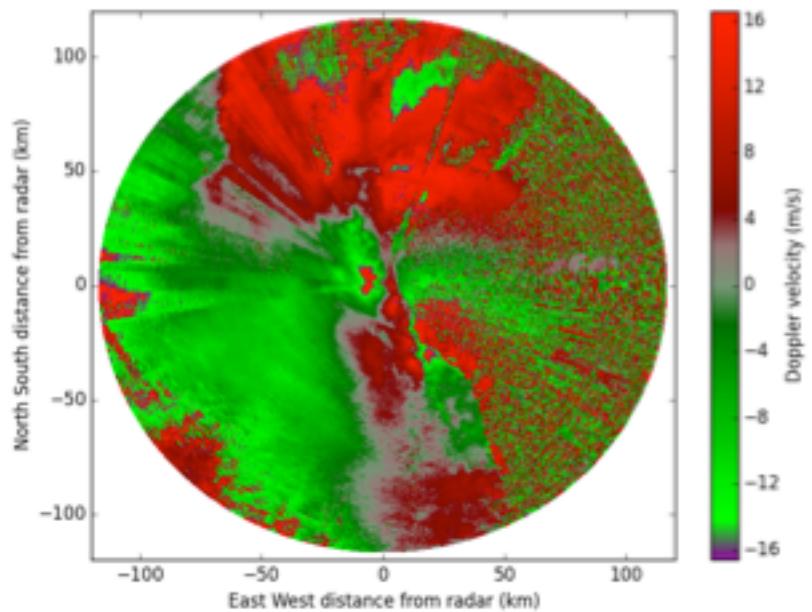
=====

```
...
160                                # find regions in original data
161    17      348933  20525.5      0.7
162    17        46      2.7      0.0
163    17        22      1.3      0.0
164    17  234857  13815.1      0.5
```

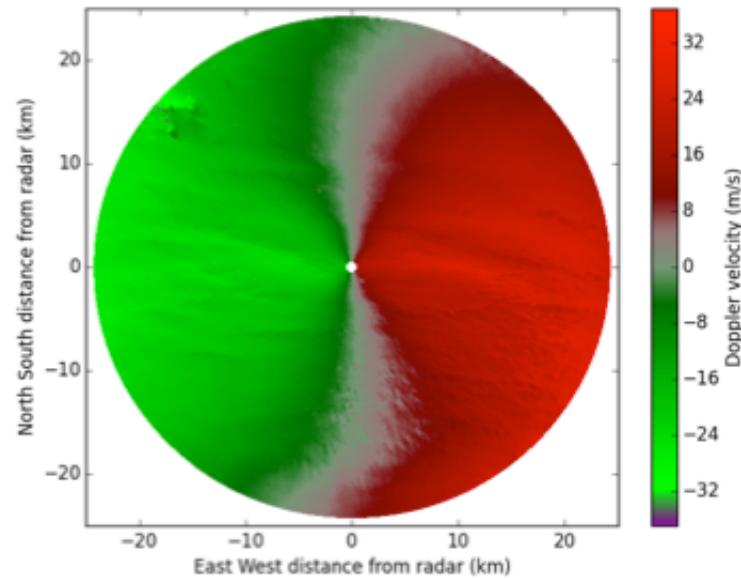
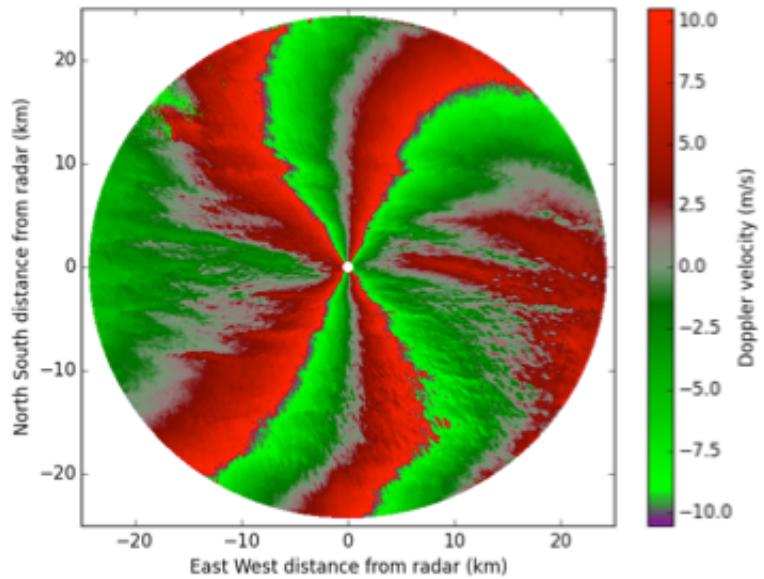
107 seconds vs. 0.234 seconds, x450 performance improvement



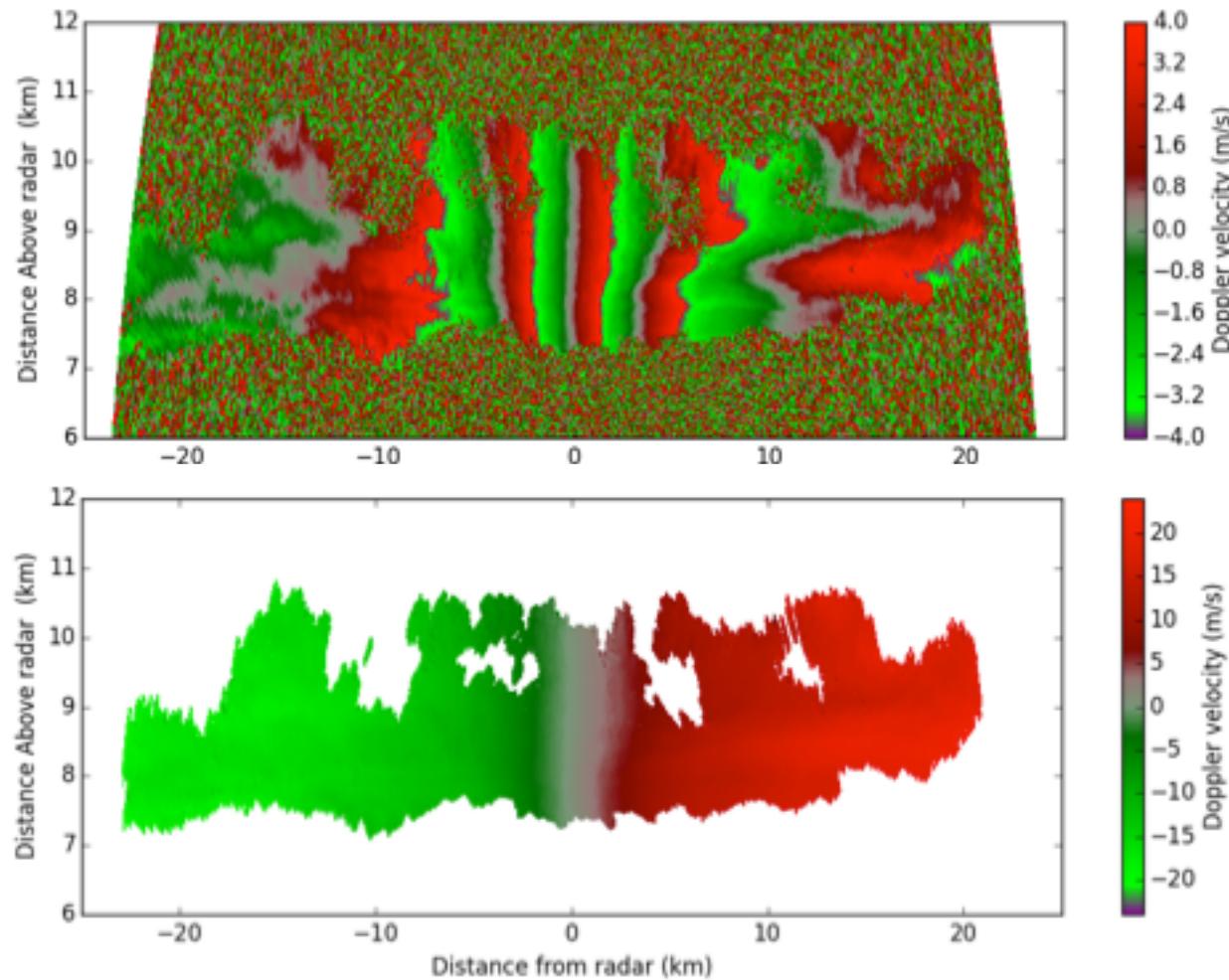
# Sample results: ARM CSAPR



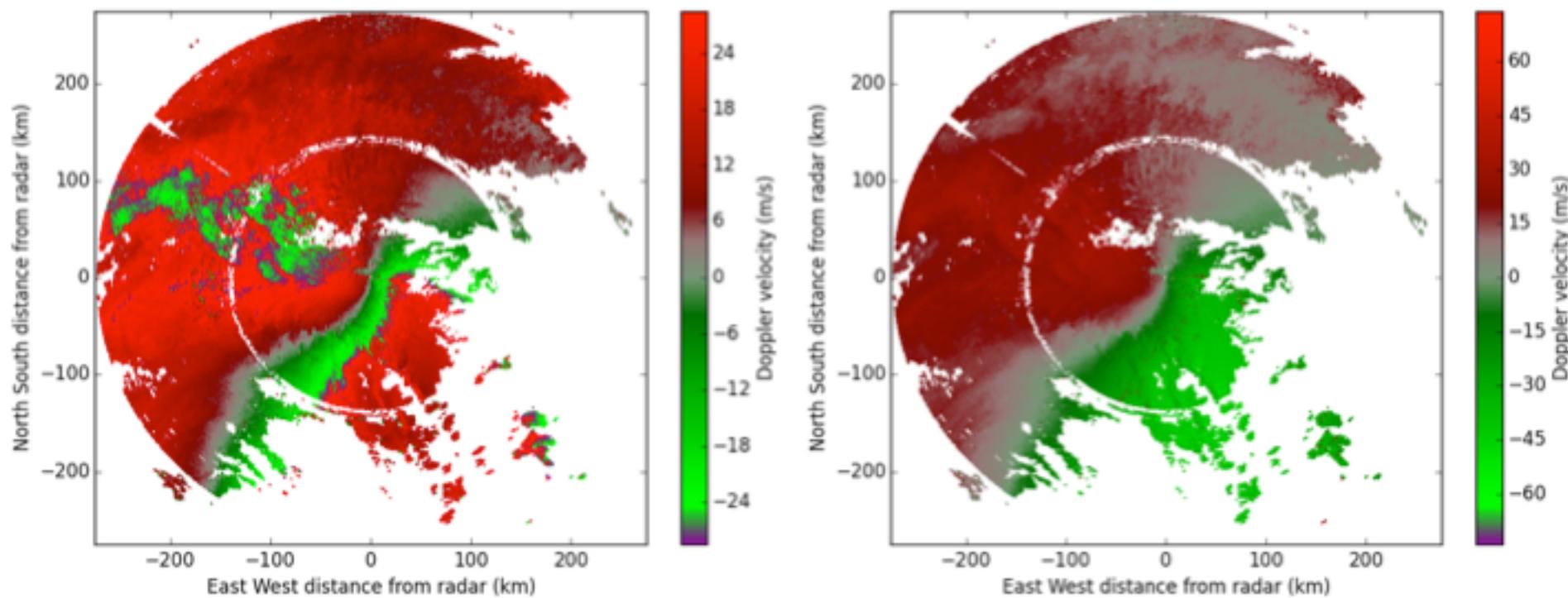
# Sample results: ARM KaSACR



# Sample results: ARM WSACR



# Sample results: NEXRAD



# Conclusions

- *Cython* is a Python to C translator which generates compiled Python modules.
- *Cython* can create **Python wrappers** around shared C/C++ libraries.
- *Cython* can create compiled modules **which incorporate C and C++ functions and classes** which are included in the package directly.
- *Cython* that can be used **to speed up the execution time** of Python code by adding **static type** information.

