TEXAS TECH UNIVERSITY™

Streamed Clustering of Lightning Mapping Data in Python Using scikit-learn

Eric Bruning
TTU Department of Geosciences
Atmospheric Science Group
@deeplycloudy

SciPy 2012
Wed, 26 June 2012
6:00-6:15 pm
Austin, TX

West Texas Lightning Mapping Array

OK

Thursday, June 27, 2013

West Texas Lightning Mapping Array

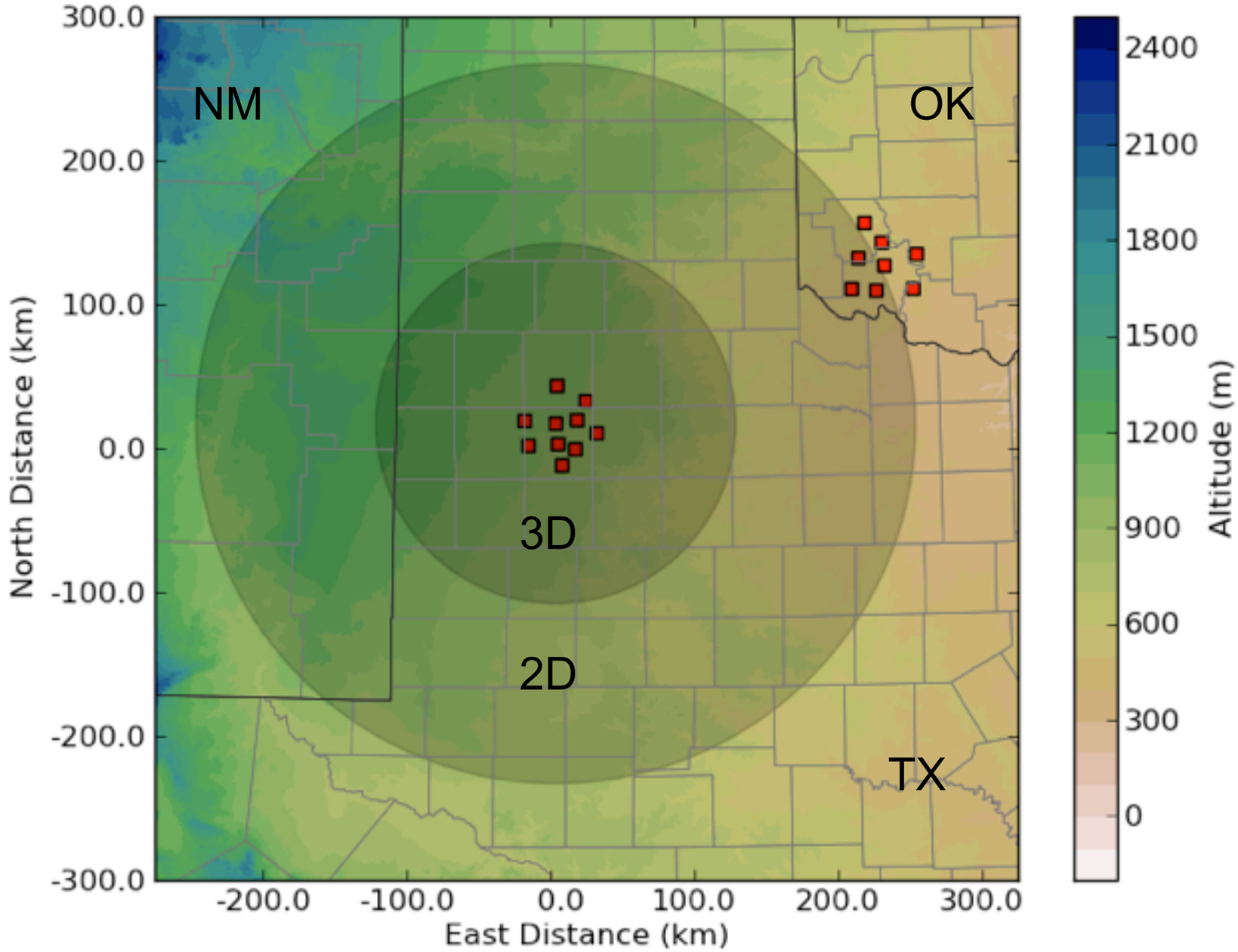
- "Triangulates" VHF sferics (think AM radio noise) produced by lightning channel steps

West Texas Lightning Mapping Array

- "Triangulates" VHF sferics (think AM radio noise) produced by lightning channel steps
- 10 stations

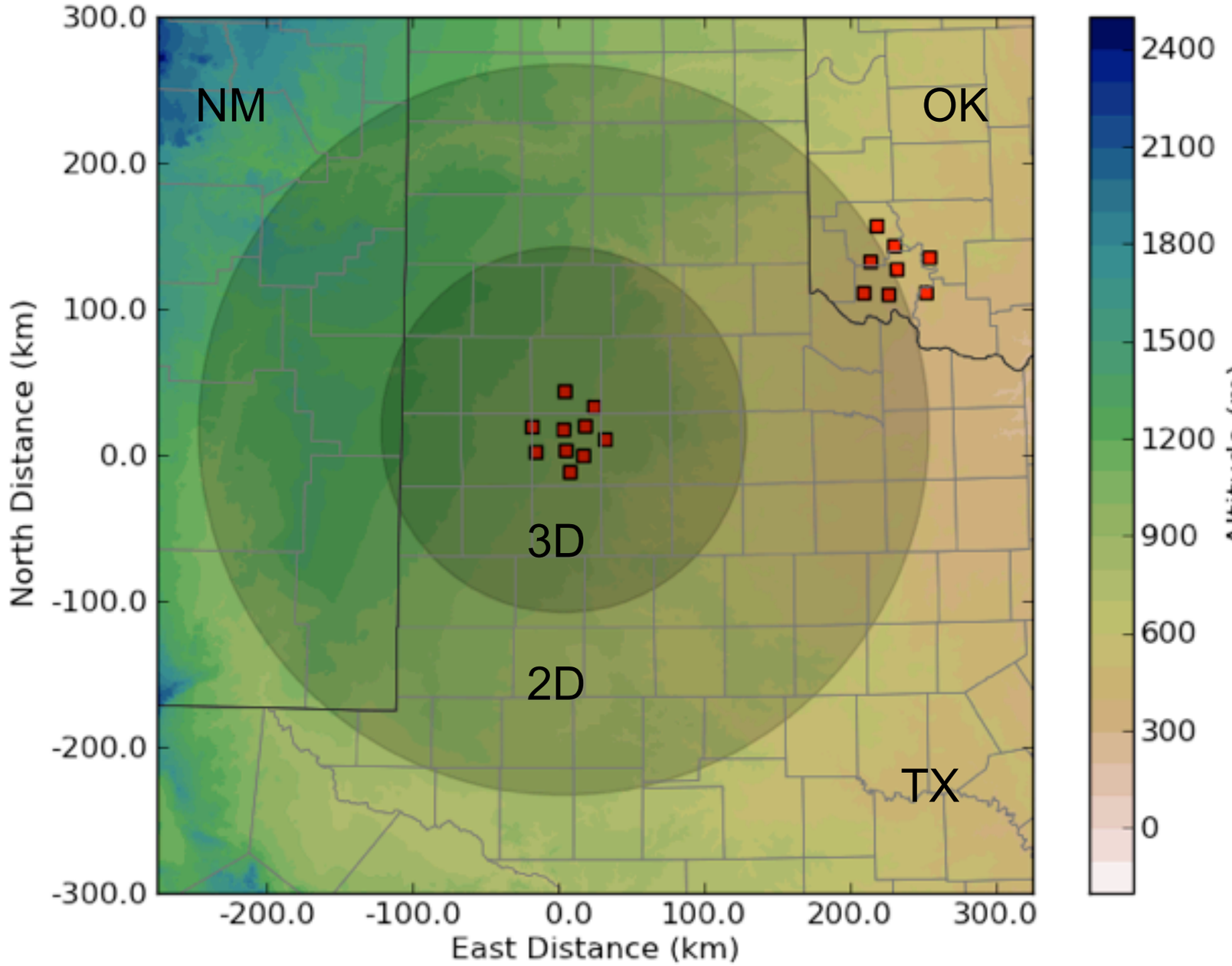
West Texas Lightning Mapping Array

- "Triangulates" VHF sferics (think AM radio noise) produced by lightning channel steps
- 10 stations



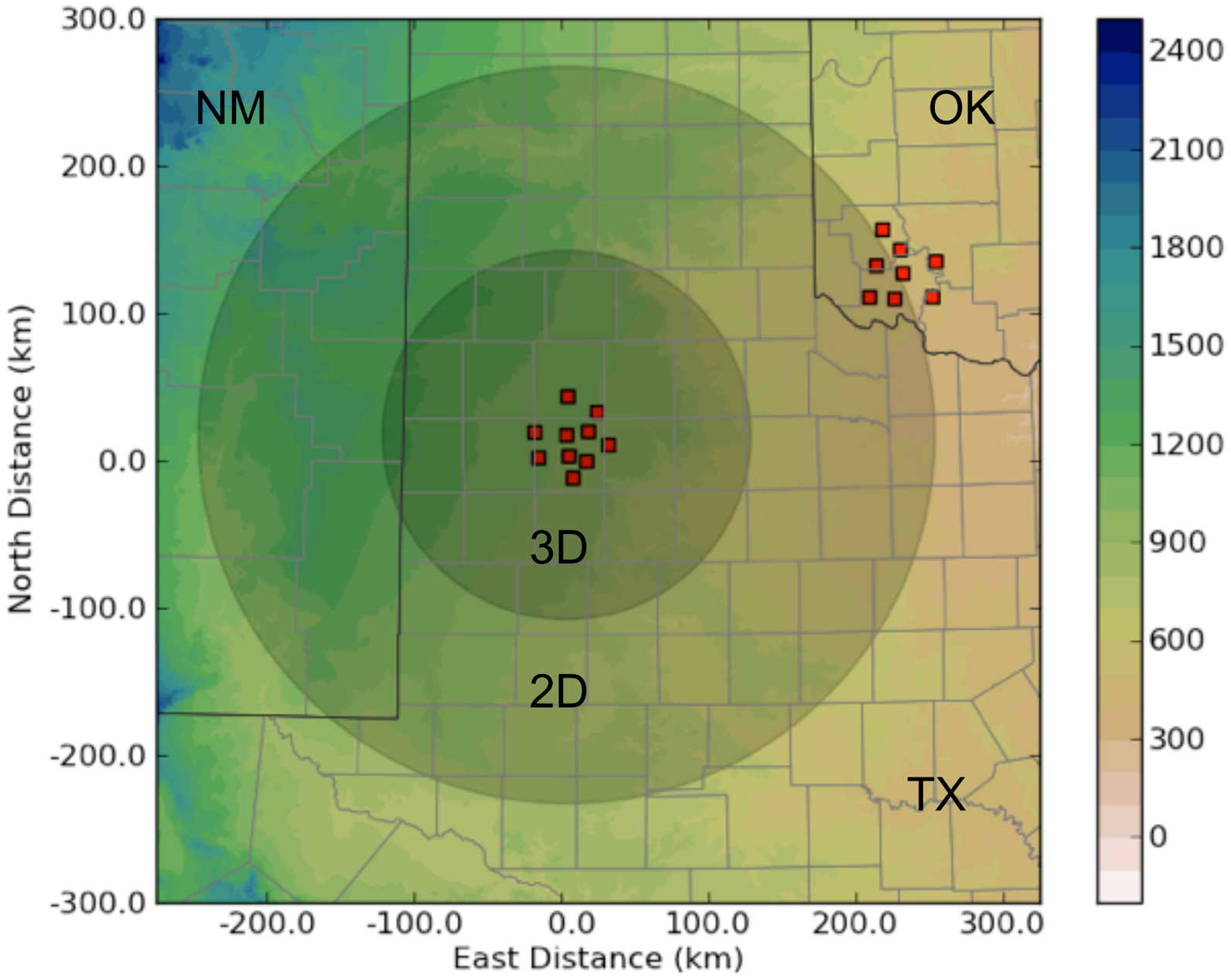
West Texas Lightning Mapping Array

- "Triangulates" VHF sferics (think AM radio noise) produced by lightning channel steps
- 10 stations
- Each station records peak pulse every 80 µs

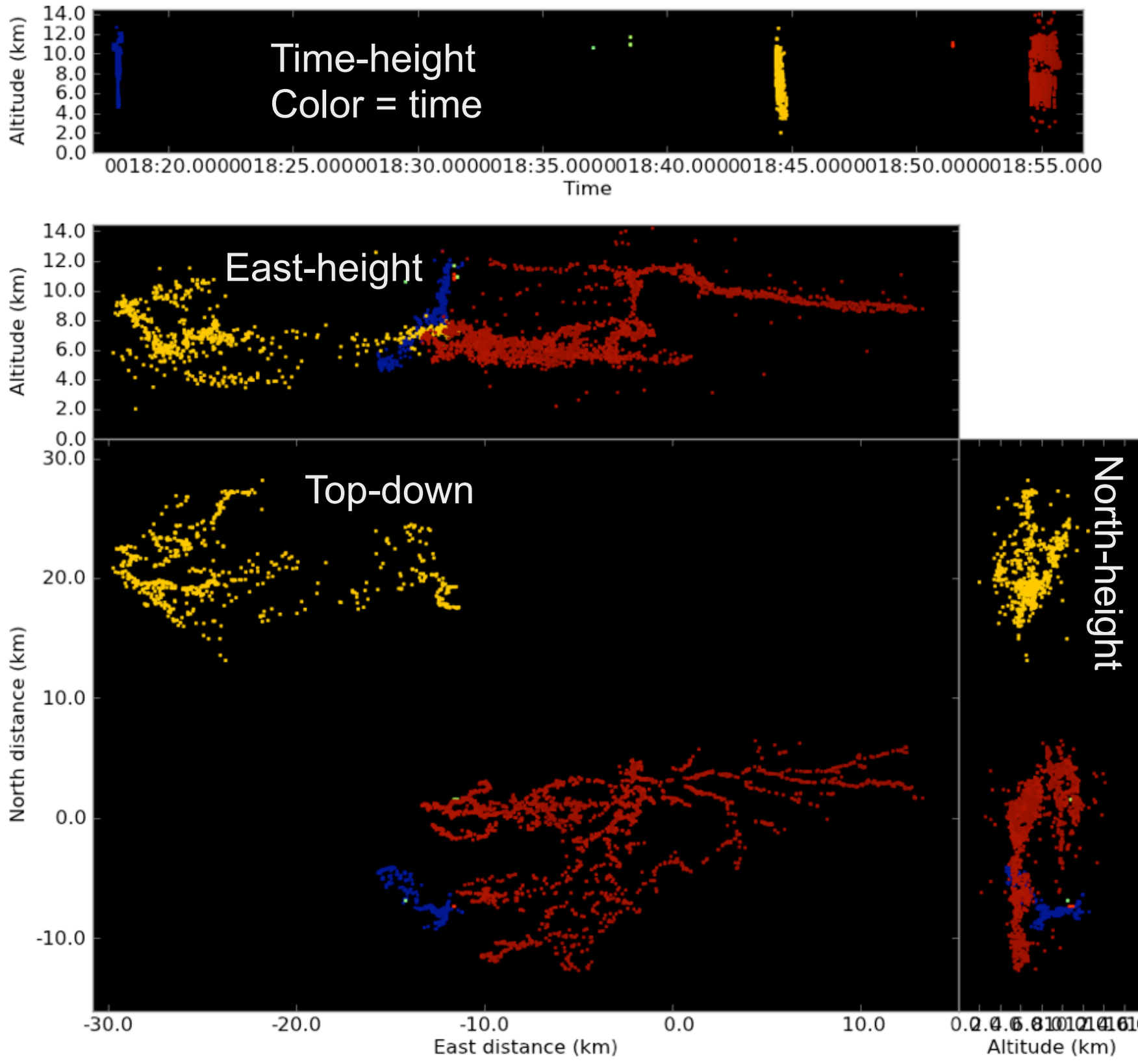


West Texas Lightning Mapping Array

- "Triangulates" VHF sferics (think AM radio noise) produced by lightning channel steps
- 10 stations
- Each station records peak pulse every 80 μs
- Typical lightning flash: 100-1000 sources

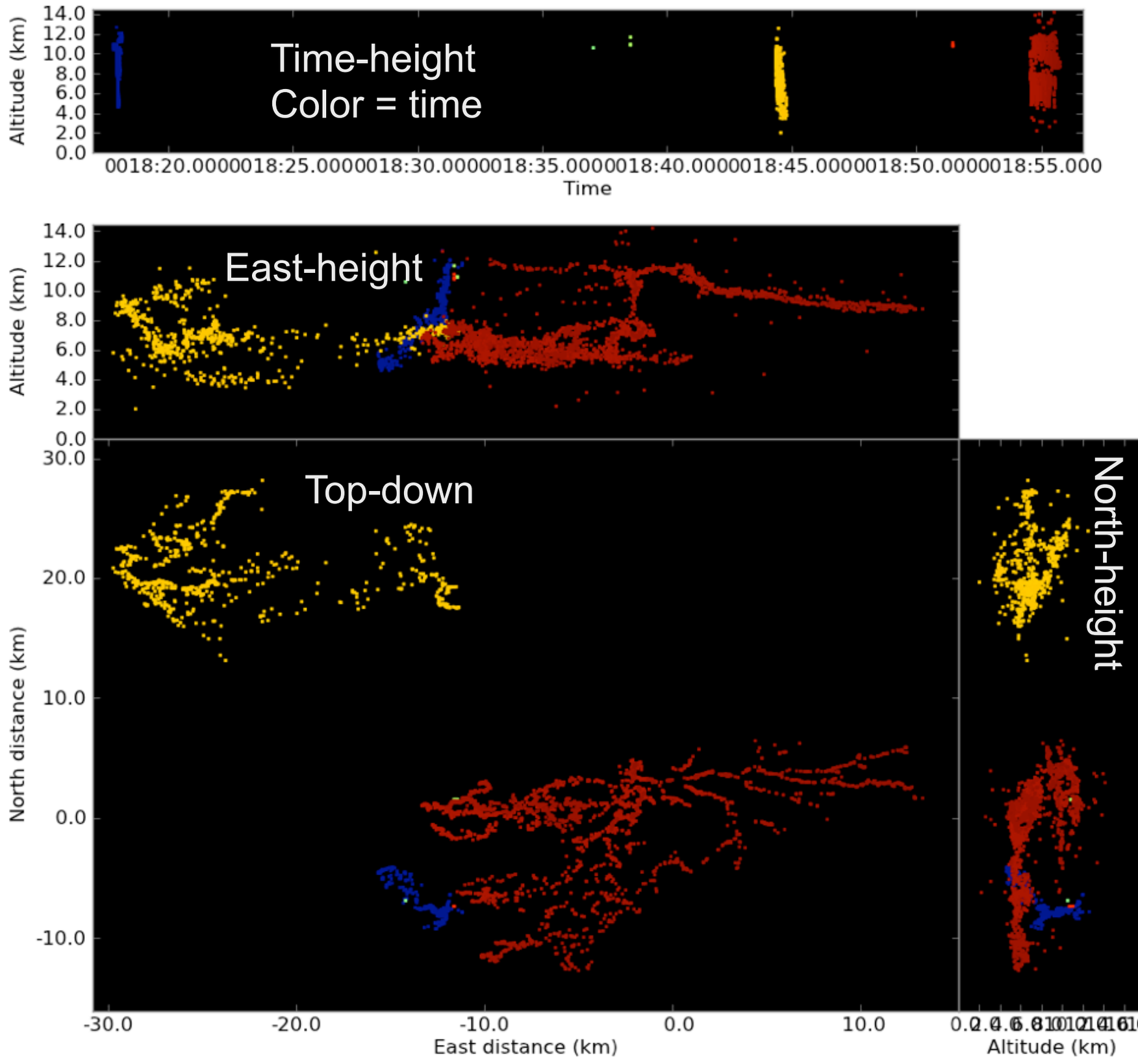


NEED: SORT VHF SOURCE LOCATIONS INTO FLASHES



NEED: SORT VHF SOURCE LOCATIONS INTO FLASHES

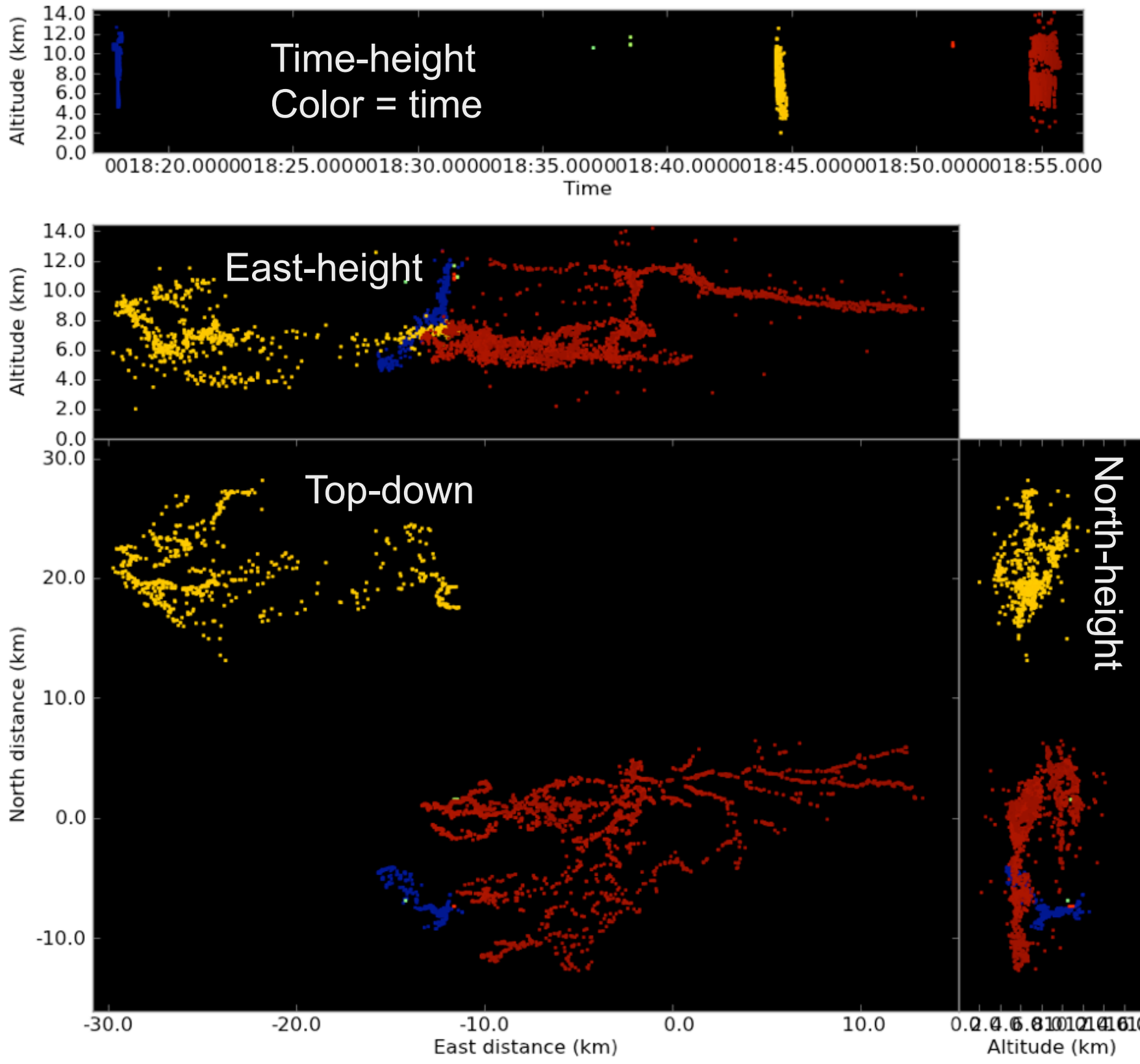
Simple case with four flashes



NEED: SORT VHF SOURCE LOCATIONS INTO FLASHES

Simple case with four flashes

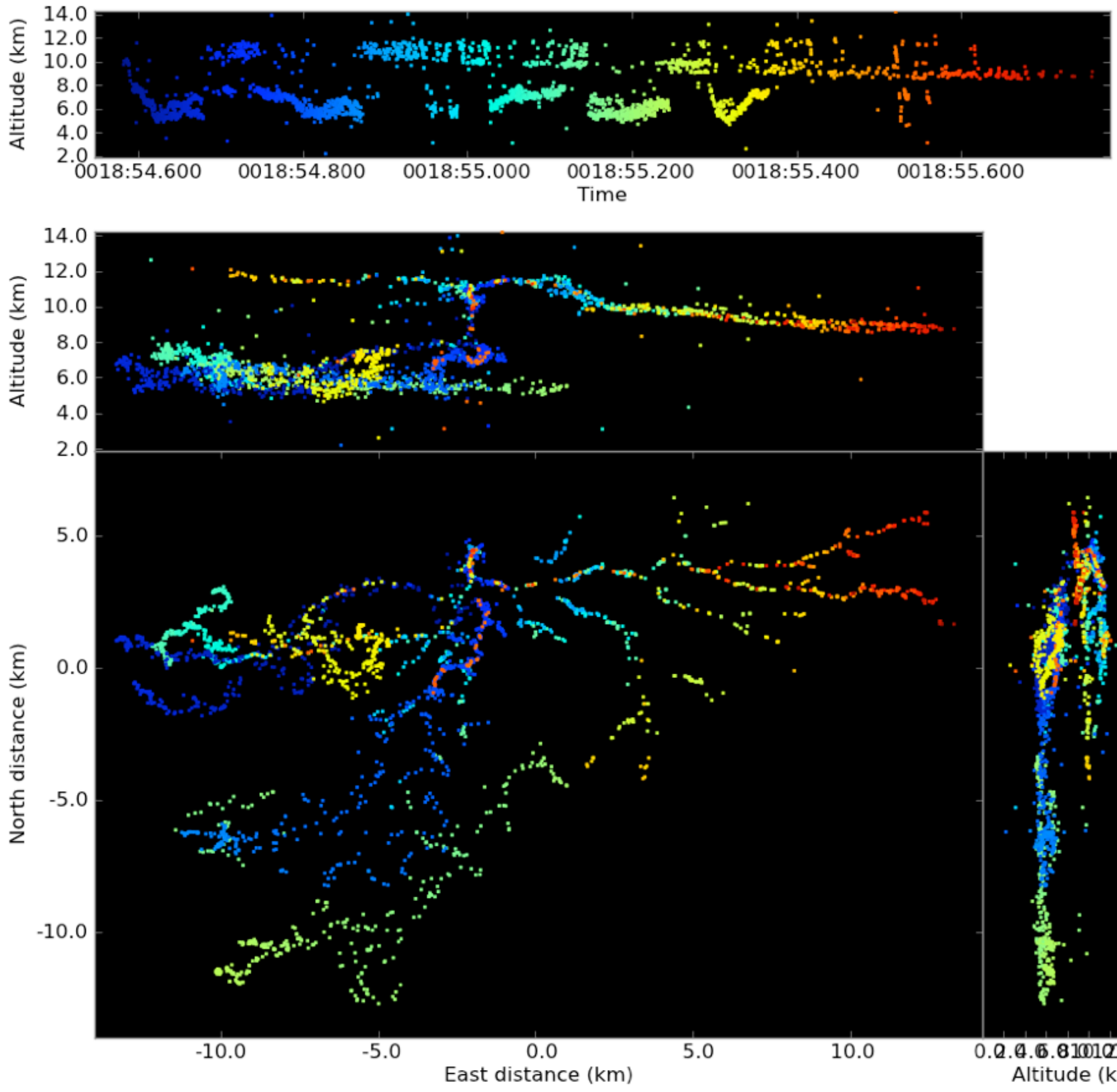
- 35 s total time



NEED: SORT VHF SOURCE LOCATIONS INTO FLASHES

Simple case with four flashes

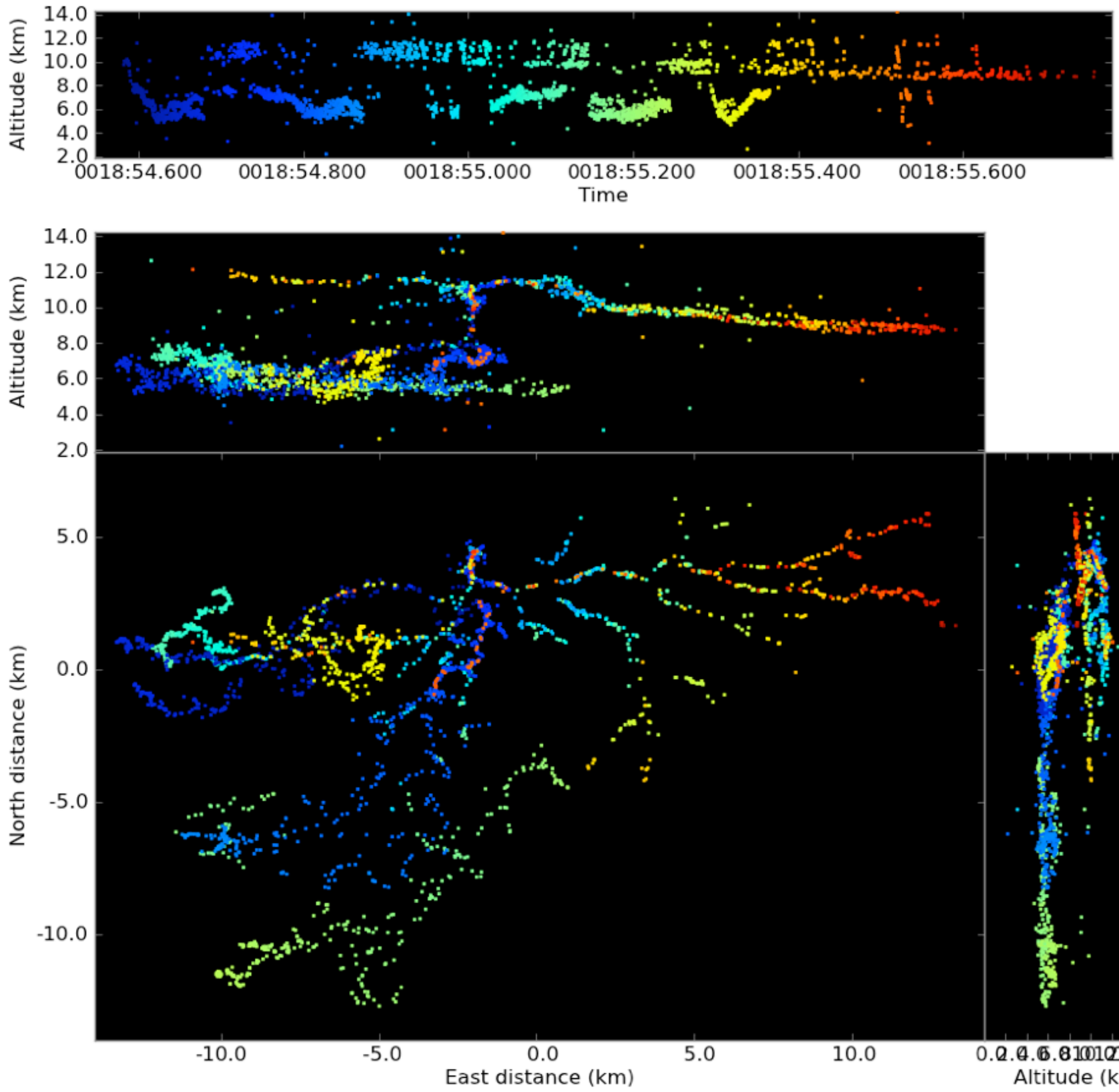
- 35 s total time
- Zoom in on one flash *(1 s long)*



NEED: SORT VHF SOURCE LOCATIONS INTO FLASHES

Simple case with four flashes

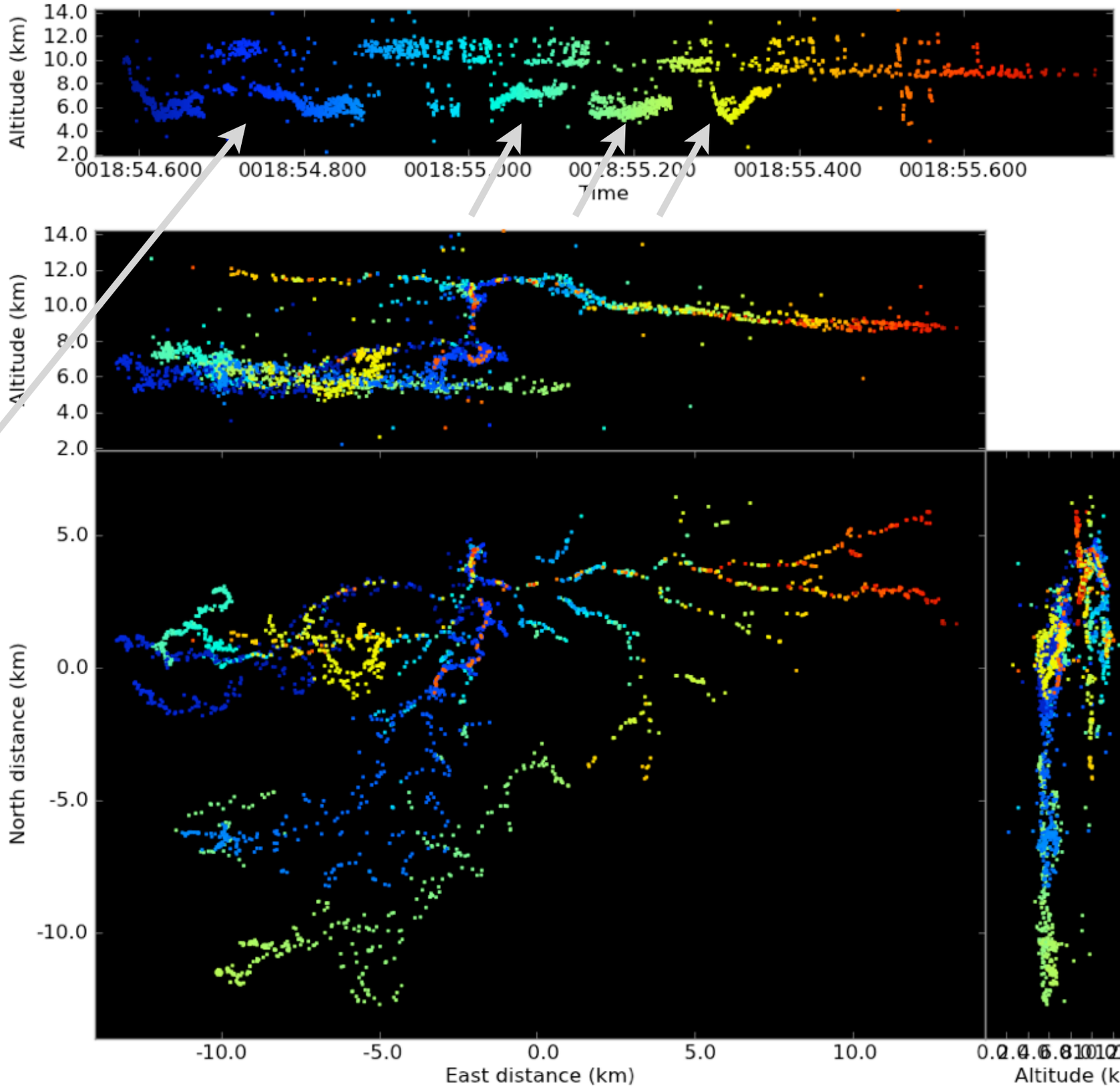
- 35 s total time
- Zoom in on one flash *(1 s long)*
- Physics evident



NEED: SORT VHF SOURCE LOCATIONS INTO FLASHES

Simple case with four flashes

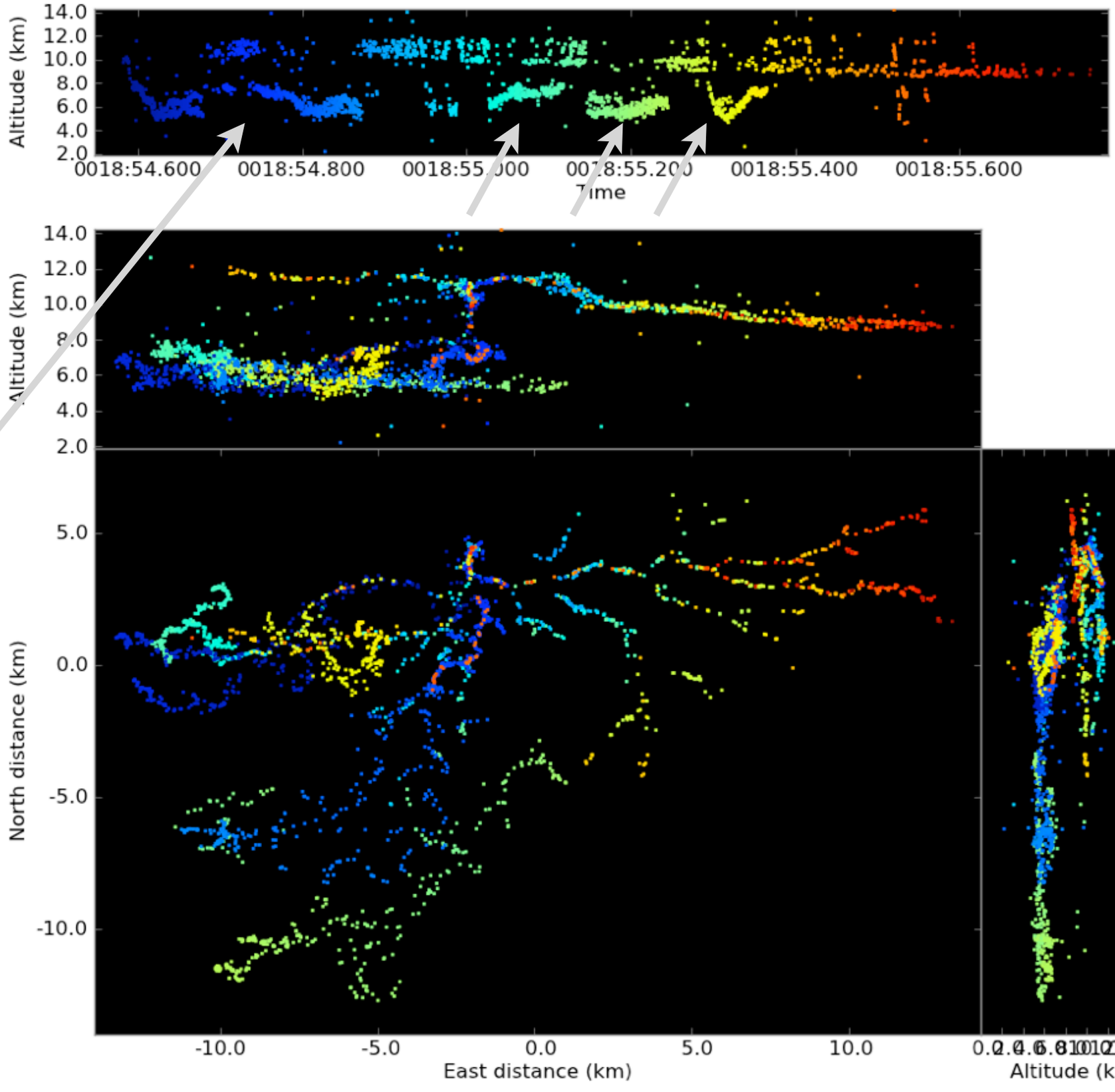
- 35 s total time
- Zoom in on one flash *(1 s long)*
- Physics evident
  - *Episodic extension of lower-level negative channel*



NEED: SORT VHF SOURCE LOCATIONS INTO FLASHES

Simple case with four flashes

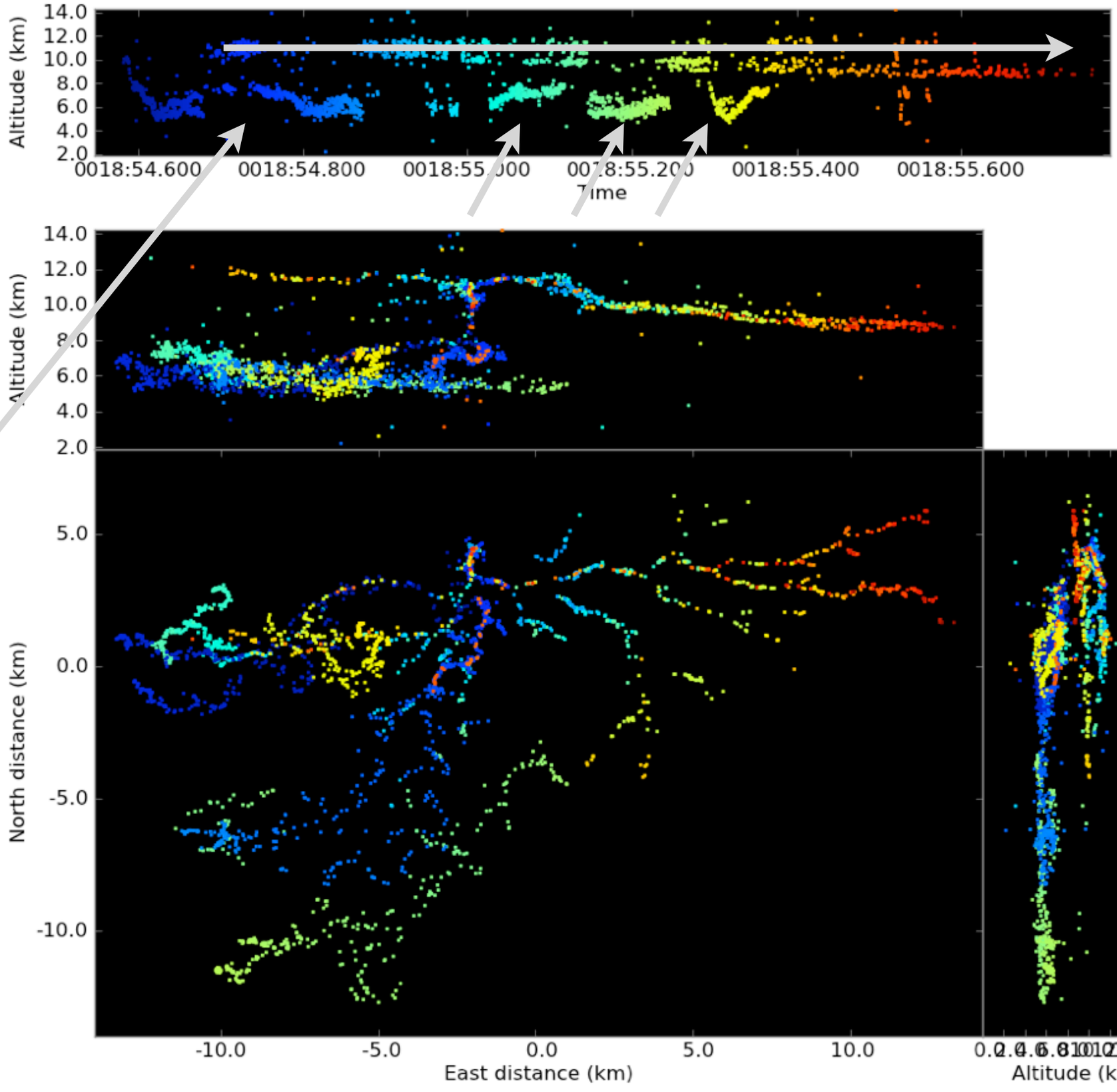
- 35 s total time
- Zoom in on one flash *(1 s long)*
- Physics evident
  - *Episodic extension of lower-level negative channel*
  - *Continuous extension of upper positive channel*



NEED: SORT VHF SOURCE LOCATIONS INTO FLASHES

Simple case with four flashes

- 35 s total time
- Zoom in on one flash *(1 s long)*
- Physics evident
  - *Episodic extension of lower-level negative channel*
  - *Continuous extension of upper positive channel*
  - *Pauses cause clustering trouble*



Regarding GIS

Regarding GIS

- GIS tools often assume variation in some interesting property across (x, y)

Regarding GIS

- GIS tools often assume variation in some interesting property across (x, y)
- Data processing and display for LMA data are fundamentally 4-dimensional

Regarding GIS

- GIS tools often assume variation in some interesting property across (x, y)
- Data processing and display for LMA data are fundamentally 4-dimensional
  - *And timing is better than 1 μs precision - standard datetime approach is limited*

REGARDING GIS

- GIS tools often assume variation in some interesting property across (x, y)
- Data processing and display for LMA data are fundamentally 4-dimensional
  - *And timing is better than 1 μs precision - standard datetime approach is limited*
    - float seconds since a reference datetime

Regarding GIS

- GIS tools often assume variation in some interesting property across (x, y)
- Data processing and display for LMA data are fundamentally 4-dimensional
  - *And timing is better than 1 µs precision - standard datetime approach is limited*
    - float seconds since a reference datetime
- I use some low-level wrappers around pyproj

Regarding GIS

- GIS tools often assume variation in some interesting property across (x, y)
- Data processing and display for LMA data are fundamentally 4-dimensional
  - *And timing is better than 1 µs precision - standard datetime approach is limited*
    - float seconds since a reference datetime
- I use some low-level wrappers around pyproj
  - *ProjType.toECEF, ProjType.fromECEF*

Regarding GIS

- GIS tools often assume variation in some interesting property across (x, y)
- Data processing and display for LMA data are fundamentally 4-dimensional
  - *And timing is better than 1 µs precision - standard datetime approach is limited*
    - float seconds since a reference datetime
- I use some low-level wrappers around pyproj
  - *ProjType.toECEF, ProjType.fromECEF*
  - *Maintains vertical coordinate*

Regarding GIS

- GIS tools often assume variation in some interesting property across (x, y)
- Data processing and display for LMA data are fundamentally 4-dimensional
  - *And timing is better than 1 µs precision - standard datetime approach is limited*
    - float seconds since a reference datetime
- I use some low-level wrappers around pyproj
  - *ProjType.toECEF, ProjType.fromECEF*
  - *Maintains vertical coordinate*
  - *Project geographic data to/from (polar) weather radar coordinates*

FLASH SORTING ALGORITHMS

FLASH SORTING ALGORITHMS

Would like to automatically separate VHF sources into flashes

FLASH SORTING ALGORITHMS

Would like to automatically separate VHF sources into flashes

Several existing implementations

FLASH SORTING ALGORITHMS

Would like to automatically separate VHF sources into flashes

Several existing implementations

- Citable

FLASH SORTING ALGORITHMS

Would like to automatically separate VHF sources into flashes

Several existing implementations

- Citable
  - *e.g., Wiens et al. (2005), MacGorman et al. (2008), McCaul et al. (2009)*

FLASH SORTING ALGORITHMS

Would like to automatically separate VHF sources into flashes

Several existing implementations

- Citable
  - *e.g., Wiens et al. (2005), MacGorman et al. (2008), McCaul et al. (2009)*
- Code is quasi-private

FLASH SORTING ALGORITHMS

Would like to automatically separate VHF sources into flashes

Several existing implementations

- Citable
  - *e.g., Wiens et al. (2005), MacGorman et al. (2008), McCaul et al. (2009)*
- Code is quasi-private
- All use some variation on grouping by time/space thresholds

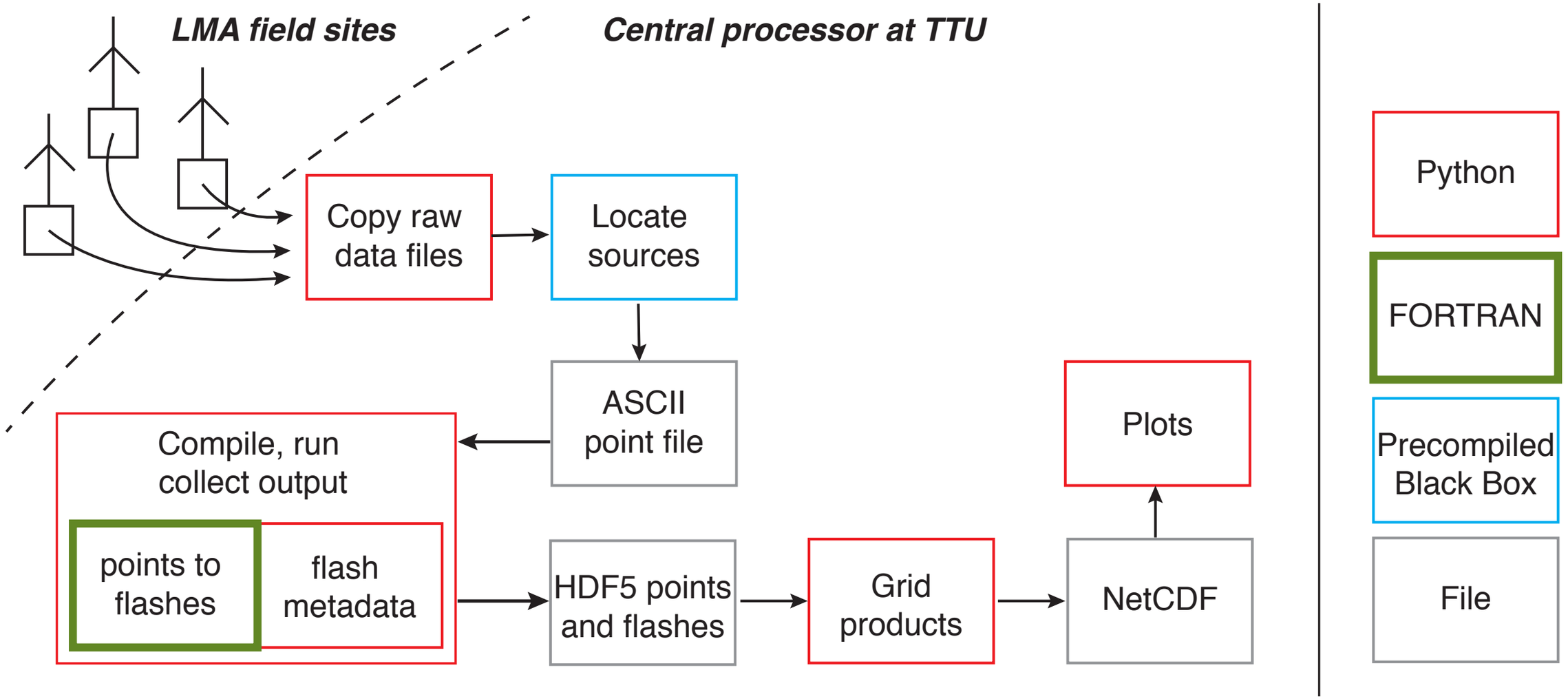
FLASH SORTING ALGORITHMS

Would like to automatically separate VHF sources into flashes

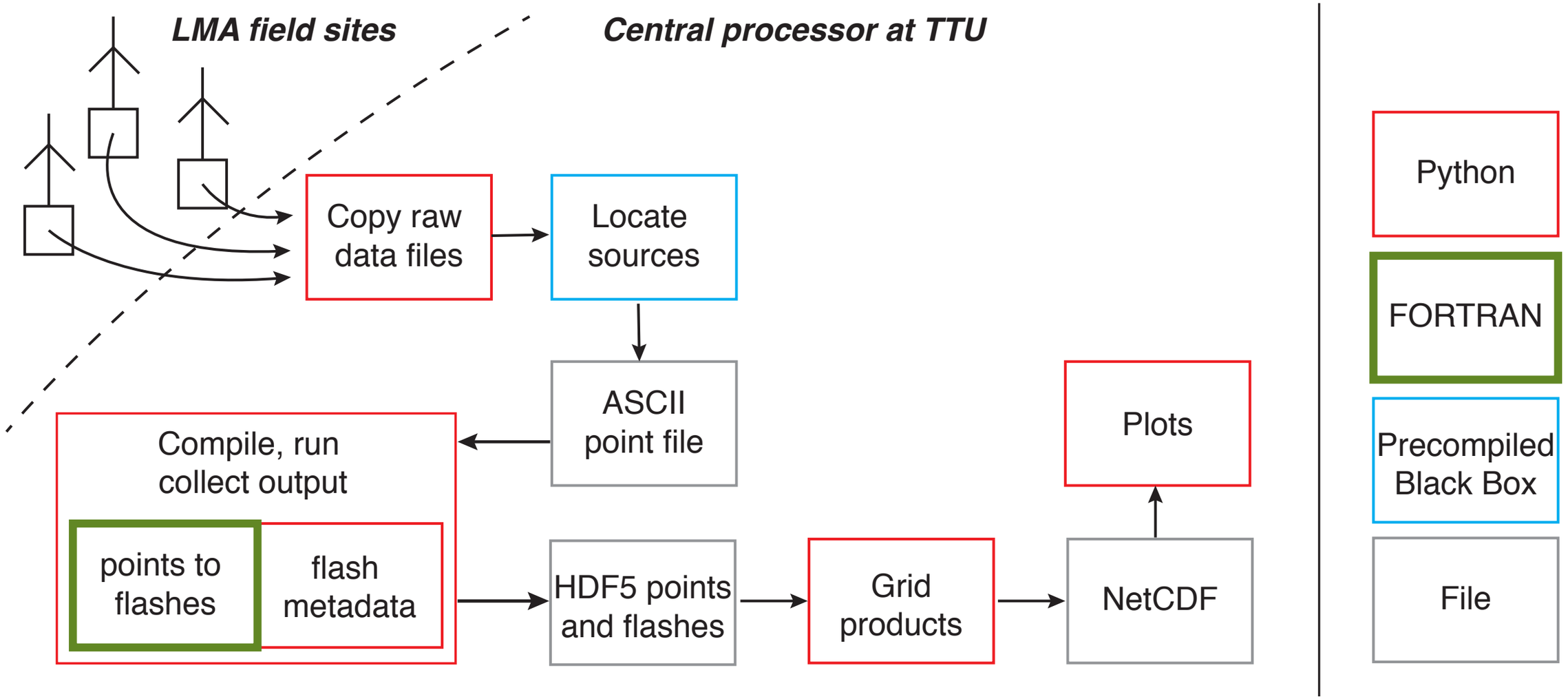
Several existing implementations

- Citable
  - *e.g., Wiens et al. (2005), MacGorman et al. (2008), McCaul et al. (2009)*
- Code is quasi-private
- All use some variation on grouping by time/space thresholds
  - *Typical: 0.15 seconds, 3 km*

CURRENT LMA PROCESSING TOOLCHAIN

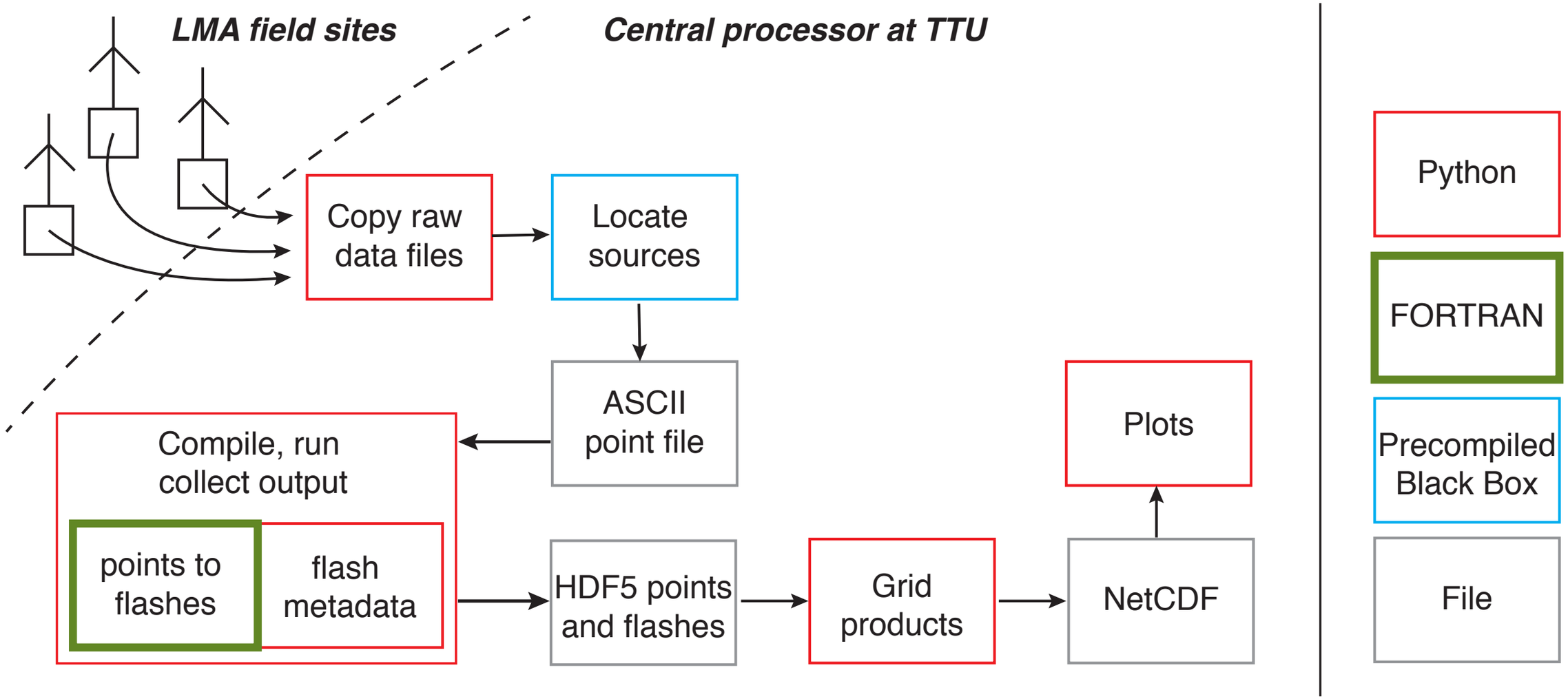


CURRENT LMA PROCESSING TOOLCHAIN



FORTRAN algorithm (McCaul et al., 2009)

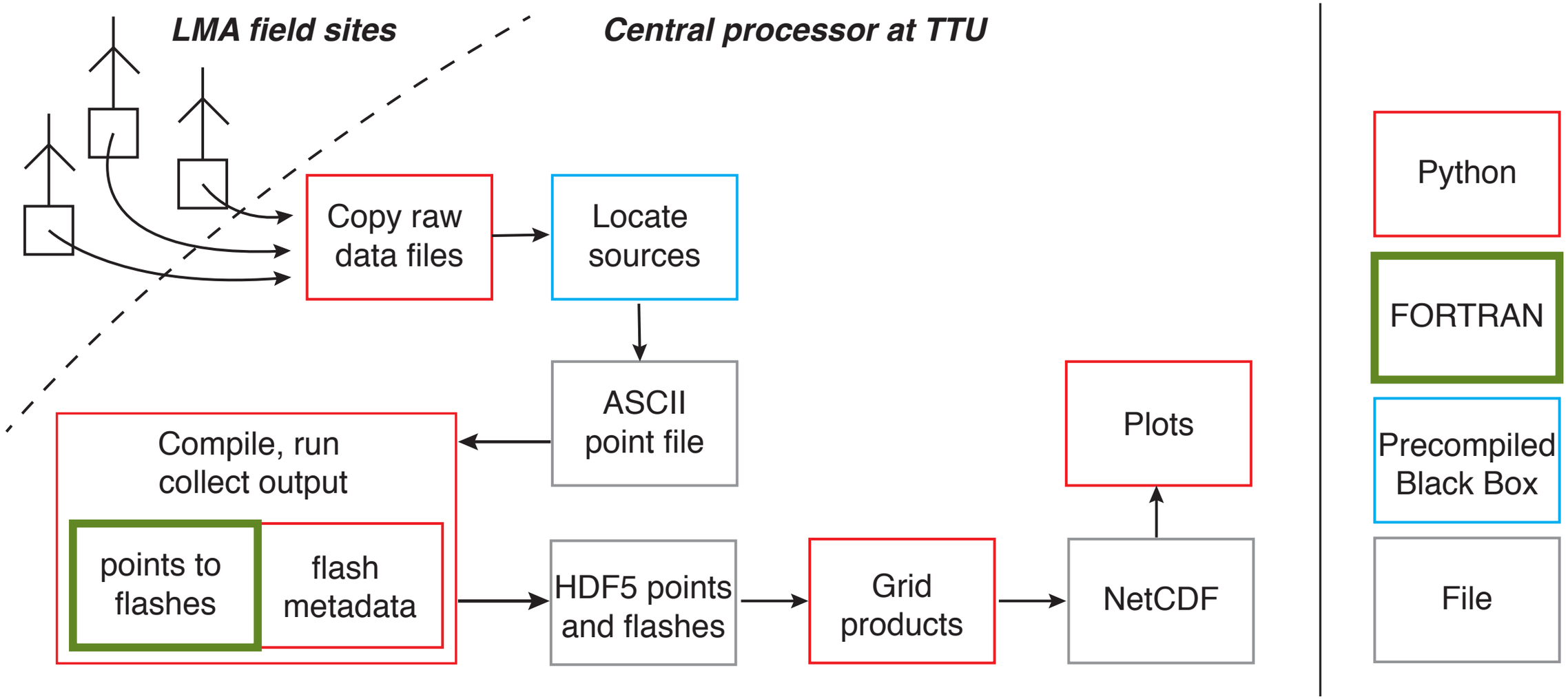
CURRENT LMA PROCESSING TOOLCHAIN



FORTRAN algorithm (McCaul et al., 2009)

- Single script with interwoven I/O and processing

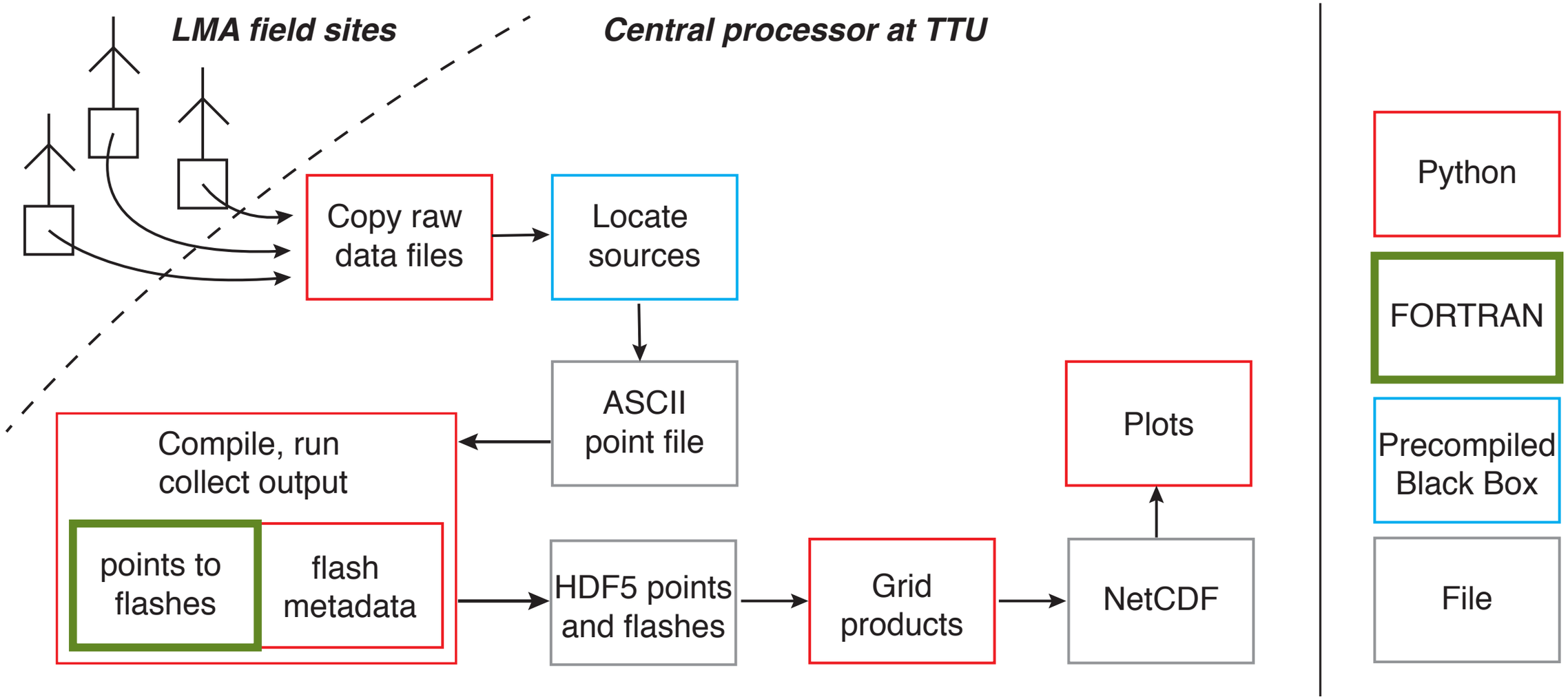
CURRENT LMA PROCESSING TOOLCHAIN



FORTRAN algorithm (McCaul et al., 2009)

- Single script with interwoven I/O and processing
- Triggered by Python. Reread FORTRAN's ASCII output format.

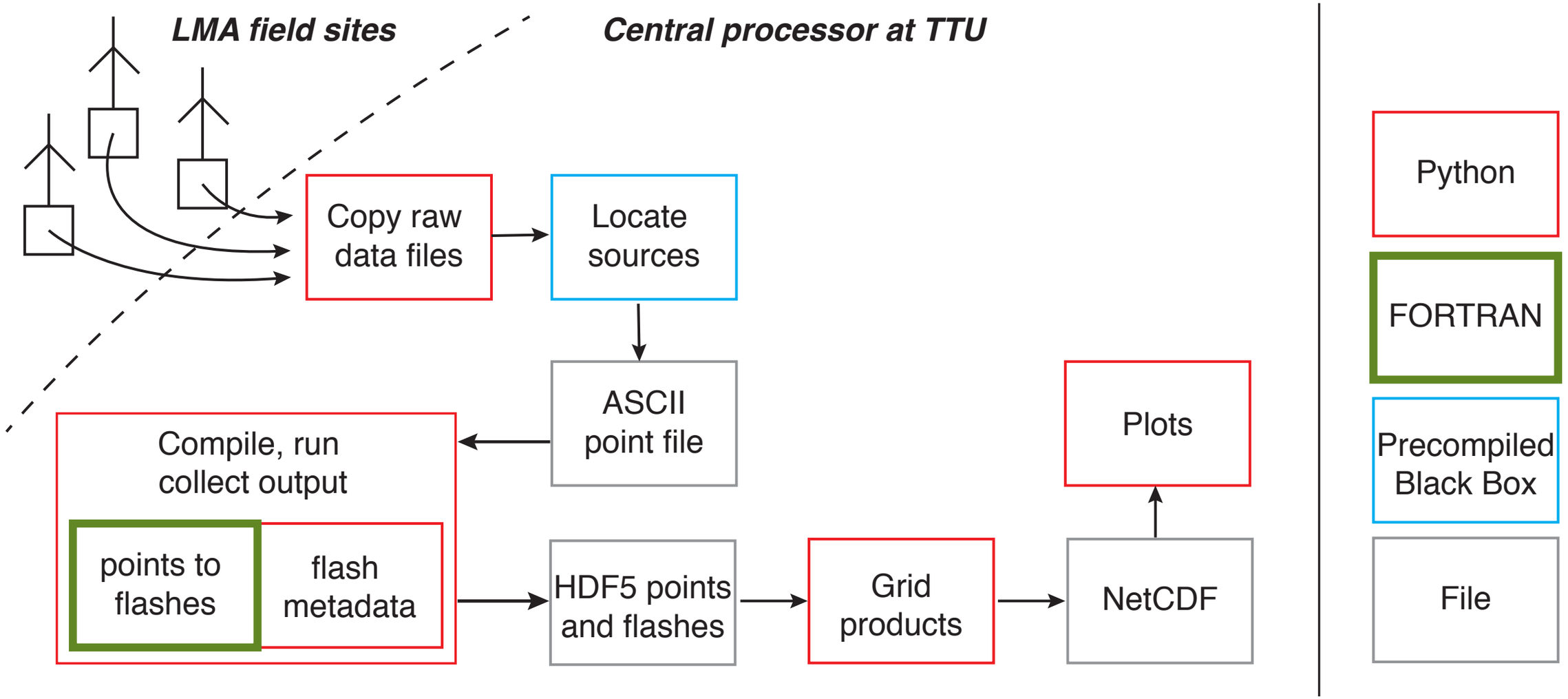
CURRENT LMA PROCESSING TOOLCHAIN



FORTRAN algorithm (McCaul et al., 2009)

- Single script with interwoven I/O and processing
- Triggered by Python. Reread FORTRAN's ASCII output format.
- Sensitive to file format changes

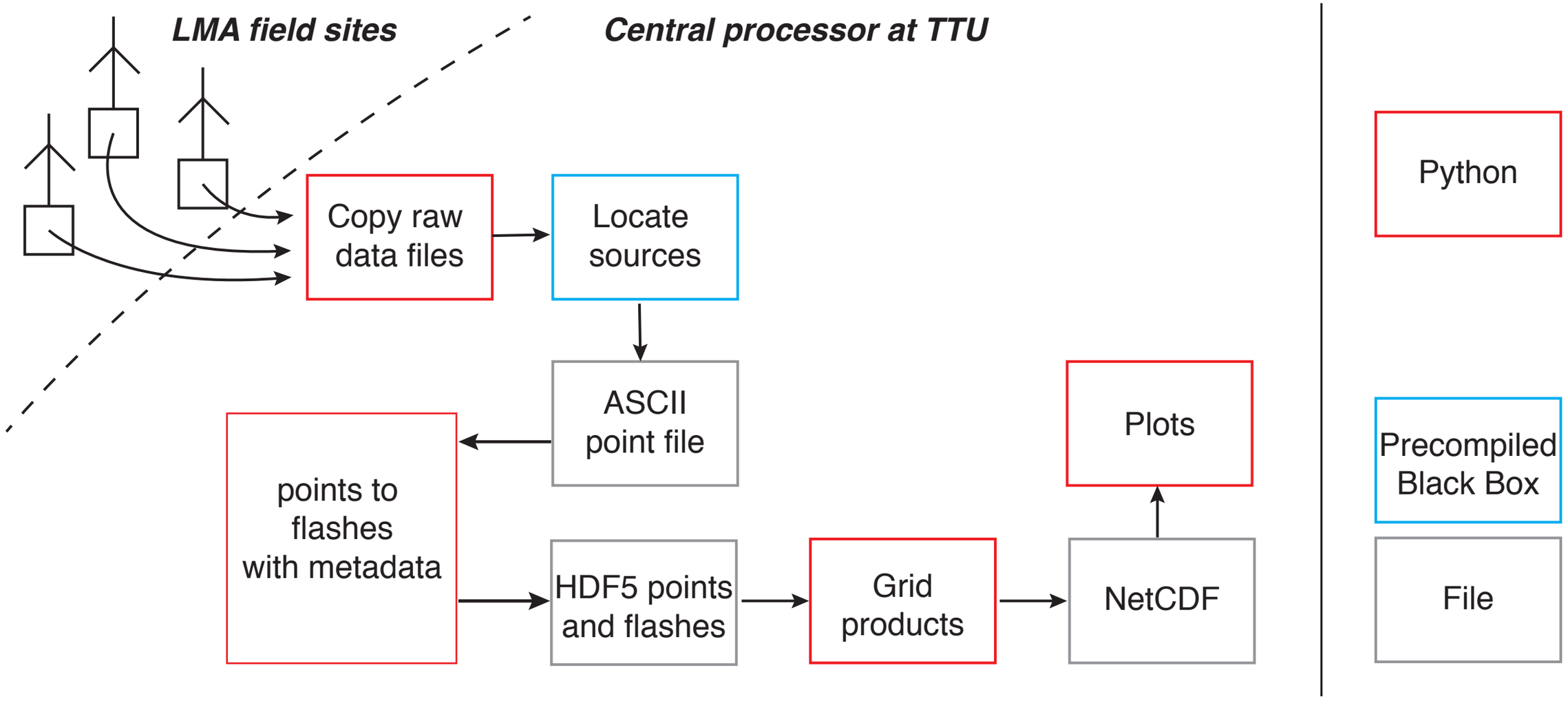
CURRENT LMA PROCESSING TOOLCHAIN



FORTRAN algorithm (McCaul et al., 2009)

- Single script with interwoven I/O and processing
- Triggered by Python. Reread FORTRAN's ASCII output format.
- Sensitive to file format changes
- Hard to maintain

CURRENT LMA PROCESSING TOOLCHAIN



FORTRAN algorithm (McCaul et al., 2009)

- Single script with interwoven I/O and processing
- Triggered by Python. Reread FORTRAN's ASCII output format.
- Sensitive to file format changes
- Hard to maintain

FLASH CLUSTERING WITH PYTHON: SCIKIT-LEARN

Replace the old FORTRAN algorithm

FLASH CLUSTERING WITH PYTHON: SCIKIT-LEARN

Replace the old FORTRAN algorithm

Adopt DBSCAN (density-based) algorithm

FLASH CLUSTERING WITH PYTHON: SCIKIT-LEARN

Replace the old FORTRAN algorithm

Adopt DBSCAN (density-based) algorithm

- scikit-learn implementation is $O(N^2)$

FLASH CLUSTERING WITH PYTHON: SCIKIT-LEARN

Replace the old FORTRAN algorithm

Adopt DBSCAN (density-based) algorithm

- scikit-learn implementation is $O(N^2)$
- time/space thresholds for flash sorting correspond to density

FLASH CLUSTERING WITH PYTHON: SCIKIT-LEARN

Replace the old FORTRAN algorithm

Adopt DBSCAN (density-based) algorithm

- scikit-learn implementation is $O(N^2)$
- time/space thresholds for flash sorting correspond to density

LMA data

FLASH CLUSTERING WITH PYTHON: SCIKIT-LEARN

Replace the old FORTRAN algorithm

Adopt DBSCAN (density-based) algorithm

- scikit-learn implementation is $O(N^2)$
- time/space thresholds for flash sorting correspond to density

LMA data

- $N=10^5$ points per minute of real-time data

FLASH CLUSTERING WITH PYTHON: SCIKIT-LEARN

Replace the old FORTRAN algorithm

Adopt DBSCAN (density-based) algorithm

- scikit-learn implementation is $O(N^2)$
- time/space thresholds for flash sorting correspond to density

LMA data

- $N=10^5$ points per minute of real-time data
  - *Clustering of whole minute not computationally feasible.*

FLASH CLUSTERING WITH PYTHON: SCIKIT-LEARN

Replace the old FORTRAN algorithm

Adopt DBSCAN (density-based) algorithm

- scikit-learn implementation is $O(N^2)$
- time/space thresholds for flash sorting correspond to density

LMA data

- $N=10^5$ points per minute of real-time data
  - *Clustering of whole minute not computationally feasible.*
- Exploit max flash duration (a few seconds) to do streamed, chunked processing

CLUSTERING WITH CHUNKS OF VHF SOURCE POINTS

Cycle 1

Cycle 2

Cycle 3

...

CLUSTERING WITH CHUNKS OF VHF SOURCE POINTS

- *Process VHF source buffer with length=twice max possible flash duration*

Cycle 1

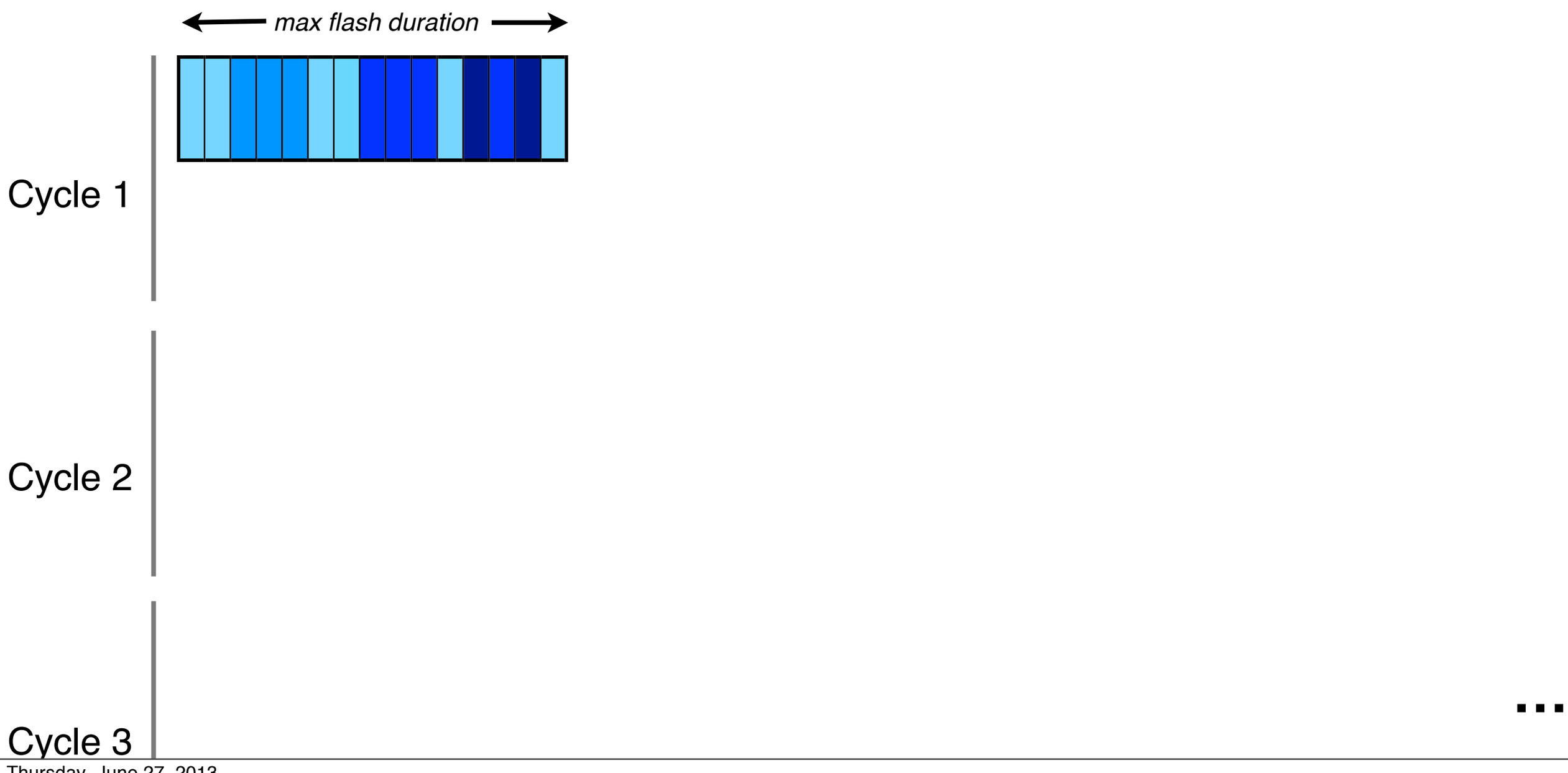
Cycle 2

Cycle 3

...

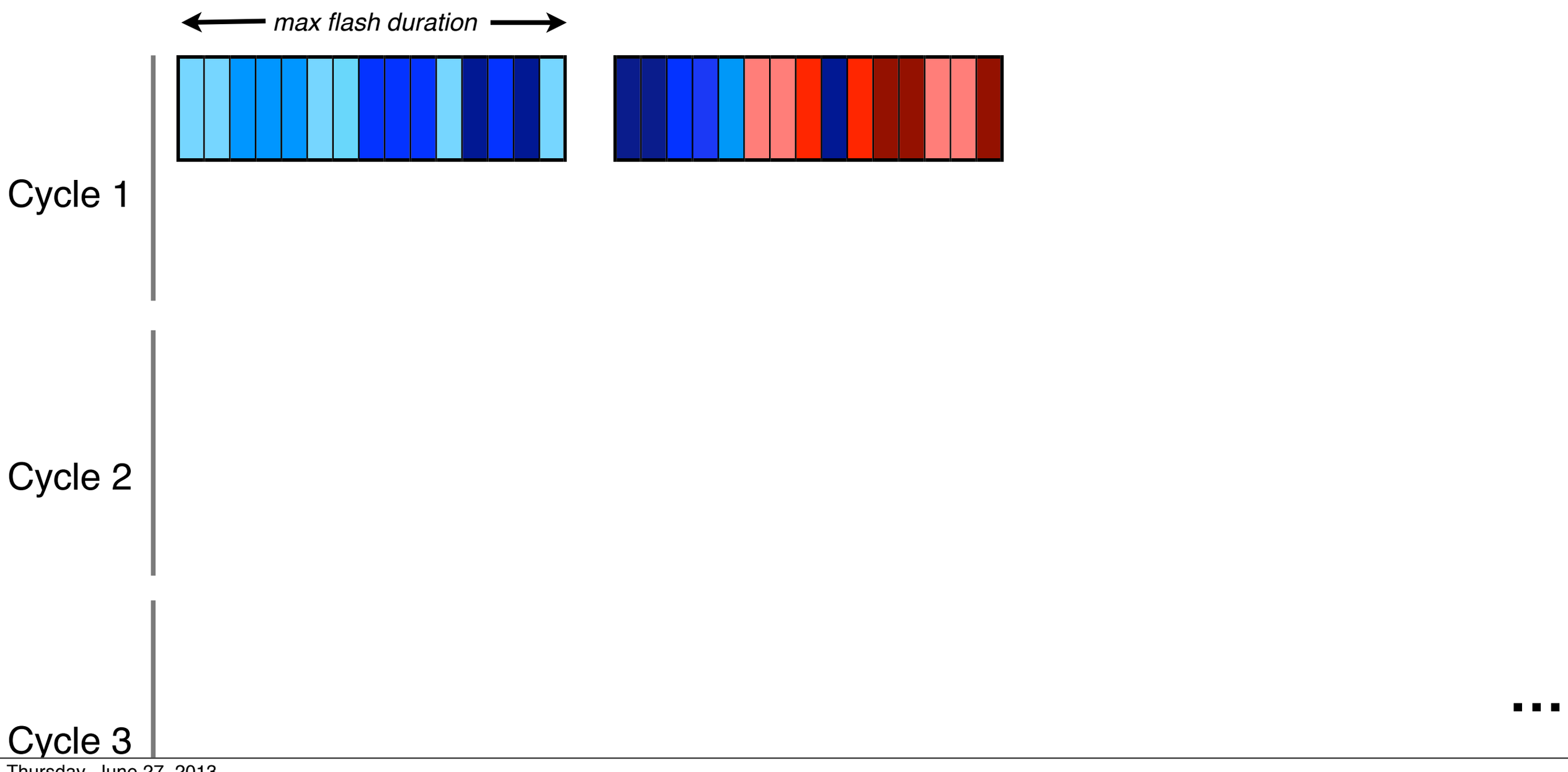
CLUSTERING WITH CHUNKS OF VHF SOURCE POINTS

- *Process VHF source buffer with length=twice max possible flash duration*



CLUSTERING WITH CHUNKS OF VHF SOURCE POINTS

- *Process VHF source buffer with length=twice max possible flash duration*



CLUSTERING WITH CHUNKS OF VHF SOURCE POINTS

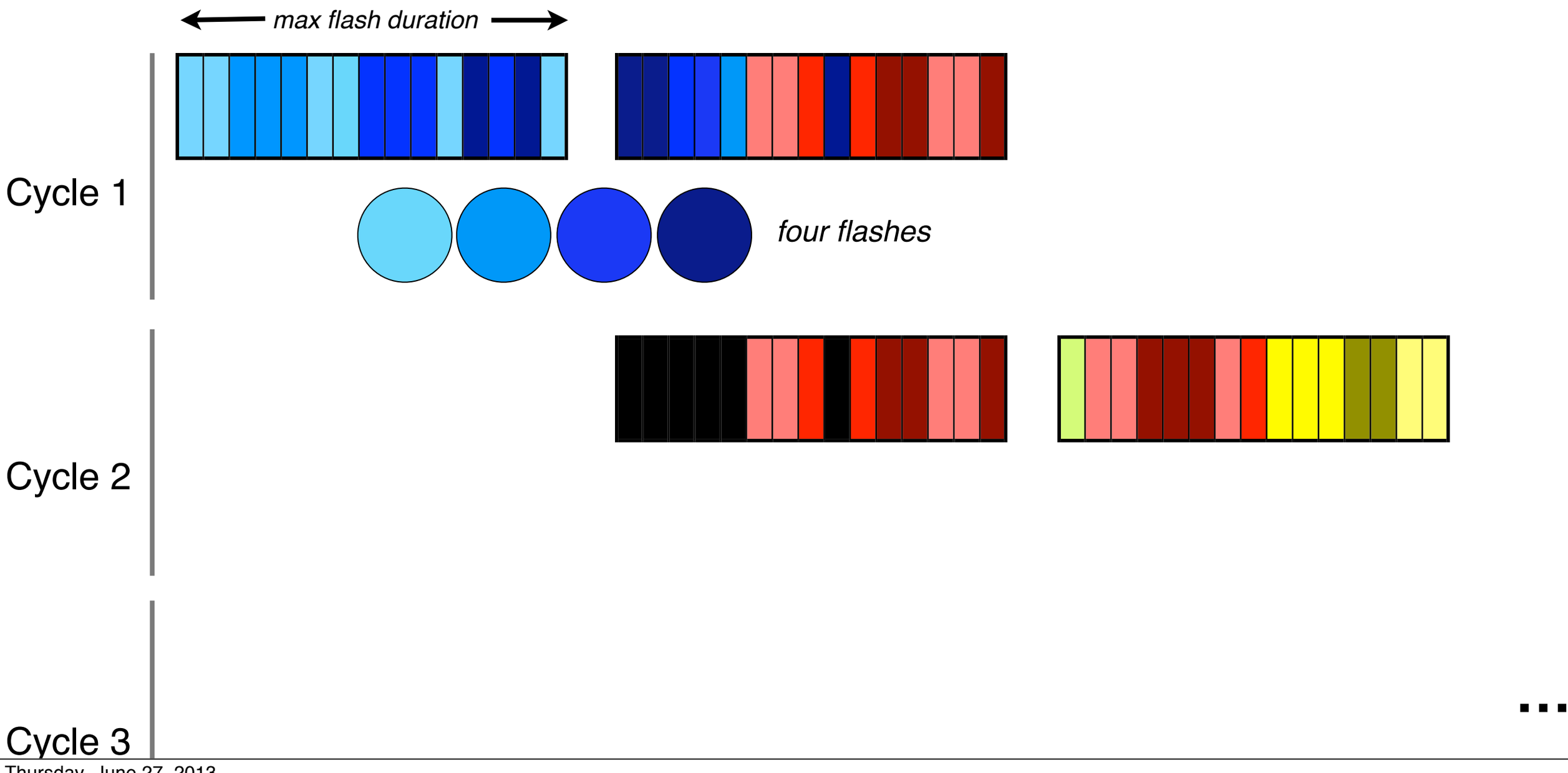
- *Process VHF source buffer with length=twice max possible flash duration*
  - All clusters starting in first half of buffer guaranteed to be complete because of cutoff

CLUSTERING WITH CHUNKS OF VHF SOURCE POINTS

- *Process VHF source buffer with length=twice max possible flash duration*
  - All clusters starting in first half of buffer guaranteed to be complete because of cutoff
- *Prune out all points in first half and their attached clusters in second half*

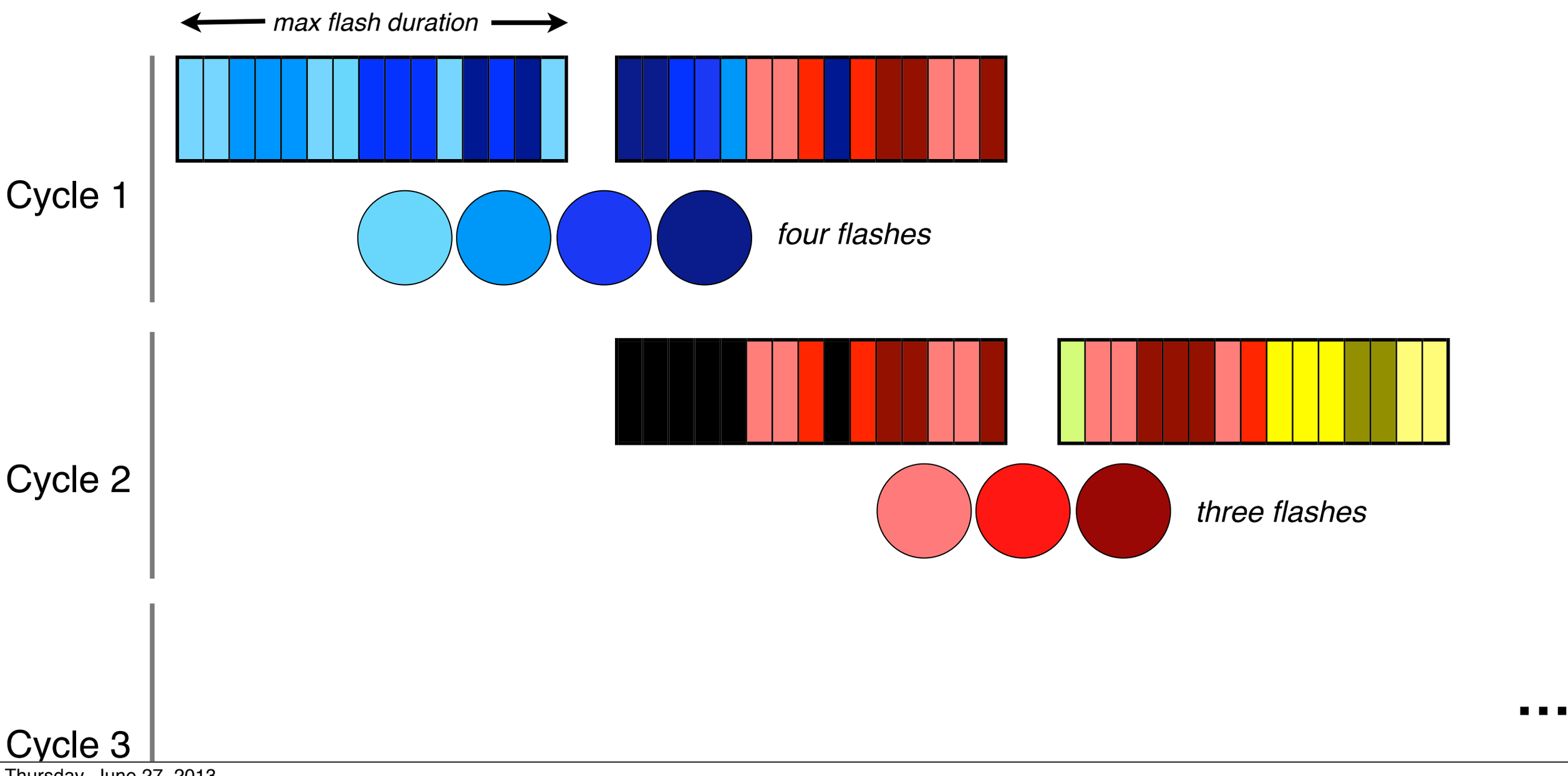
CLUSTERING WITH CHUNKS OF VHF SOURCE POINTS

- *Process VHF source buffer with length=twice max possible flash duration*
  - All clusters starting in first half of buffer guaranteed to be complete because of cutoff
- *Prune out all points in first half and their attached clusters in second half*
- *Process next chunk*



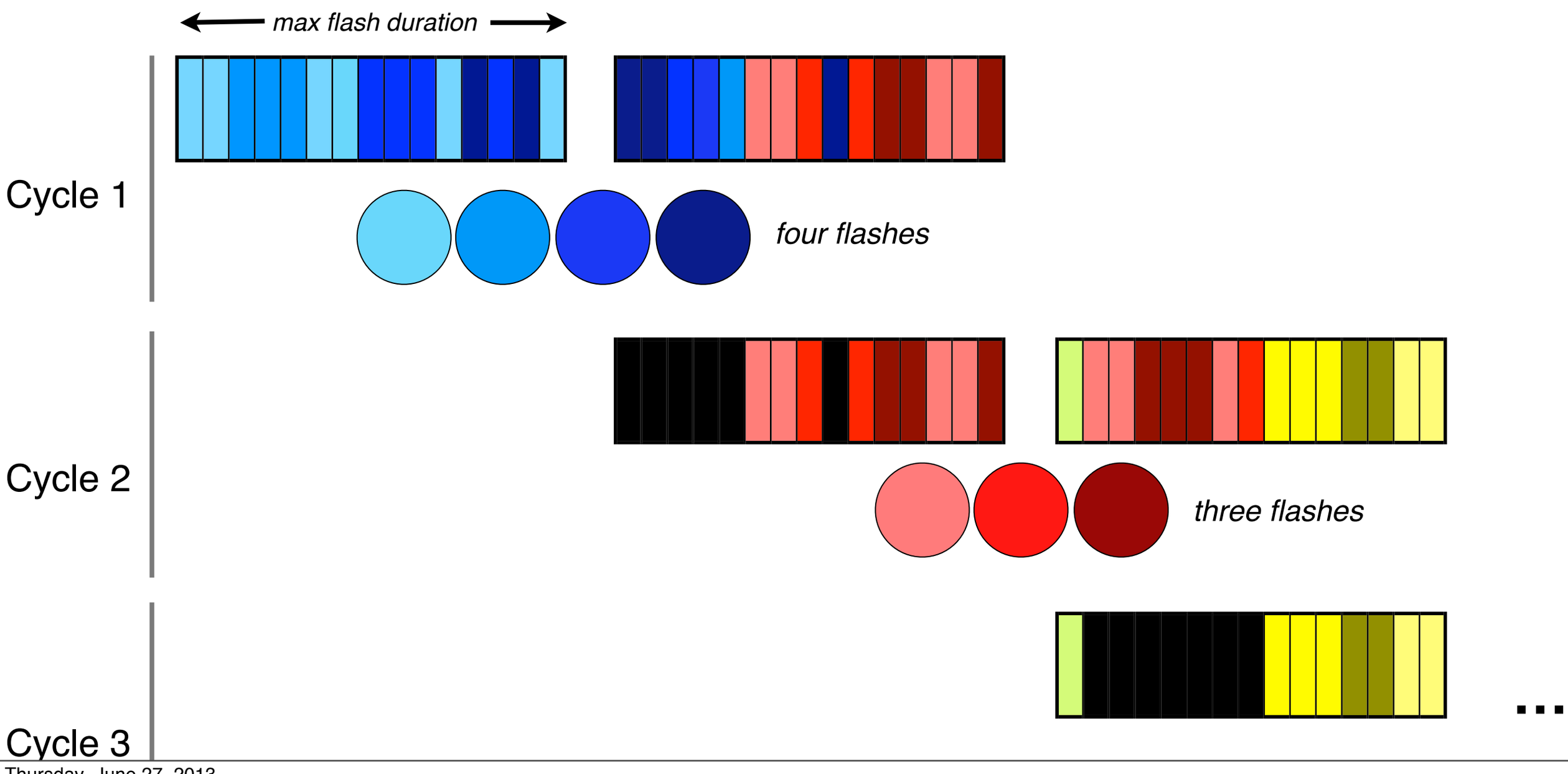
CLUSTERING WITH CHUNKS OF VHF SOURCE POINTS

- *Process VHF source buffer with length=twice max possible flash duration*
  - All clusters starting in first half of buffer guaranteed to be complete because of cutoff
- *Prune out all points in first half and their attached clusters in second half*
- *Process next chunk*



CLUSTERING WITH CHUNKS OF VHF SOURCE POINTS

- *Process VHF source buffer with length=twice max possible flash duration*
  - All clusters starting in first half of buffer guaranteed to be complete because of cutoff
- *Prune out all points in first half and their attached clusters in second half*
- *Process next chunk*



Pipeline based on coroutines: start with a stream of vectors

```
def stream(vec, target):
    for v in vec:
        target.send(v)
    target.close()

@coroutine
def chunk(start_time, max_duration, target, t_idx=-1):
    next_time = start_time + max_duration
    v_buffer = []
    try:
        while True:
            v = (yield)
            v_buffer.append(v)
            t = v[t_idx]
            if t >= next_time:
                target.send(np.asarray(v_buffer))
                v_buffer = []
                next_time = t+max_duration
    except GeneratorExit:
        target.send(np.asarray(v_buffer))
```

loop over array, send vectors

chunk **coroutine is the** target

setup

new (x,y,z,t) vector from stream

max possible duration?
send to the next step (clustering)

deal with leftovers

THE CLUSTERER

```
    @coroutine
def cluster_chunk_pairs(clustered_output_target, min_points=10):
    db = DBSCAN(eps=1.0, min_samples=min_points, metric='euclidean')
    chunk1 = (yield)
    try:
        while True:
            chunk2 = (yield)
            len1, len2 = chunk1.shape[0], chunk2.shape[0]

            # do stuff with chunk 1 and 2
            clusters = db.fit(np.vstack((chunk1, chunk2)))
            labels = clusters.labels_.astype(int)
            clustered_output_target.send((chunk1, labels[:len1]))

            # pull data out of chunk2 that was clustered as part of chunk 1
            chunk1_labelset = set(labels[:len1])
            if -1 in chunk1_labelset:
                chunk1_labelset.remove(-1) # remove the singleton cluster ID - retain these from chunk 2.
            label_iter = ( True if label in chunk1_labelset else False for i,label in enumerate(labels[len1:]) )
            clustered_in_chunk2 = np.fromiter( label_iter , dtype=bool)
            clustered_output_target.send((chunk2[clustered_in_chunk2], labels[len1:][clustered_in_chunk2]))
            residuals = chunk2[clustered_in_chunk2==False]

            # prepare for another chunk
            if len(residuals) == 0:
                residuals = chunk1[0:0,:] # empty array; preserves the number of dimensions in the data vector
            chunk1 = np.asarray(residuals)
    except GeneratorExit:
        clusters = db.fit(chunk1)
        labels = clusters.labels_.astype(int)
        clustered_output_target.send((chunk1, labels))
```

setup DBSCAN
get first chunk

get another chunk

do the clustering

figure out first chunk
for next round

process the leftovers

Final pipeline setup

```
def cluster(a_file, output_path, outfile, params, logger, min_points=1, **kwargs):

    ... preprocessing ...

    X, Y, Z, T = from_data

    D_max, t_max = 3.0e3, 0.15 # m, s

    X_vector = np.hstack((X[:,None],Y[:,None],Z[:,None])) / D_max
    T_vector = data['time'][:,None] / t_max
    XYZT = np.hstack((X_vector, T_vector-T_vector.min()))
```

preprocessing to get T; cartesian X,Y,Z from geographic data

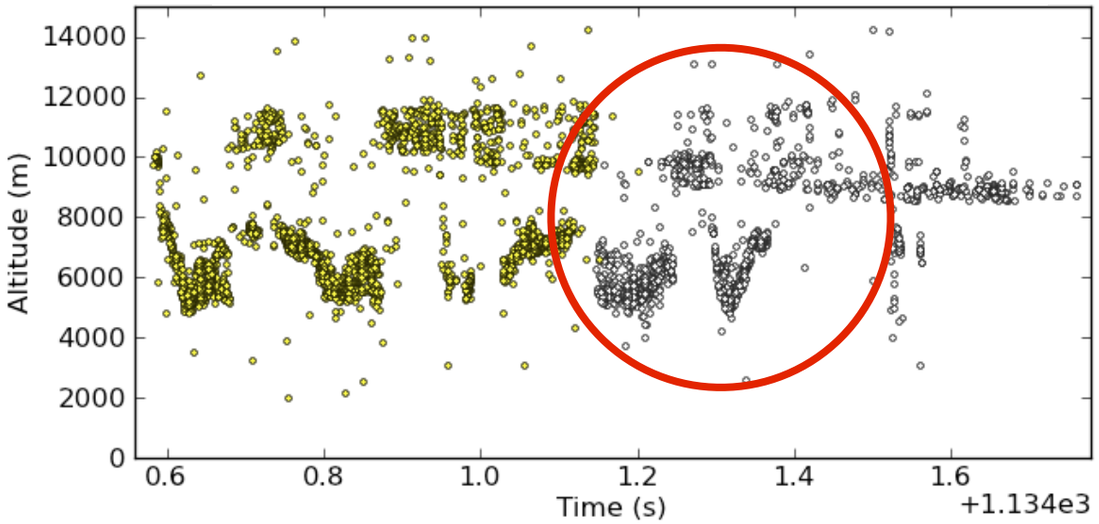
create a normalized vector

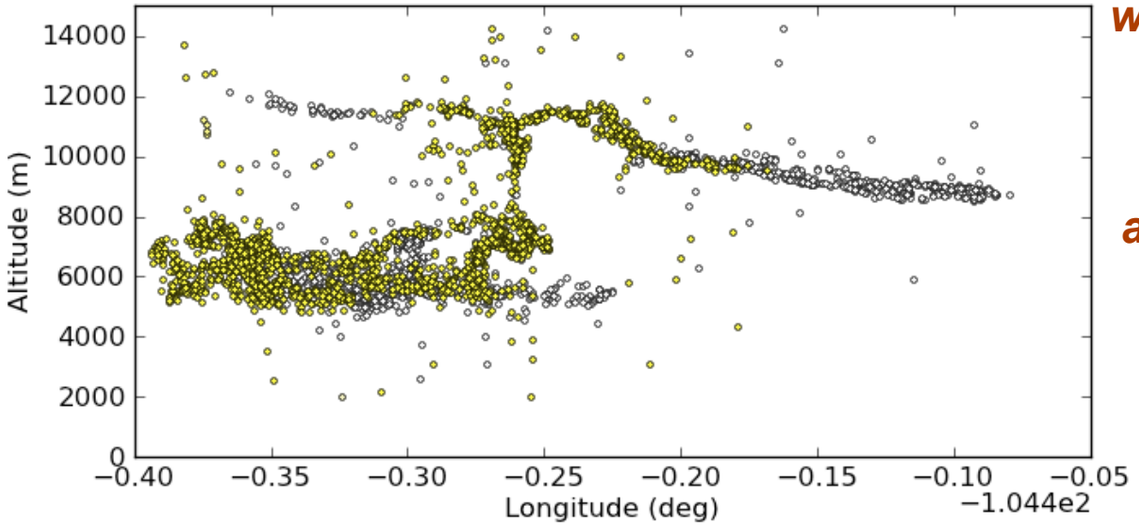
set up the pipeline: `stream ⇒ chunker ⇒ clusterer ⇒ label_aggregator ⇒ create_flash_objs`

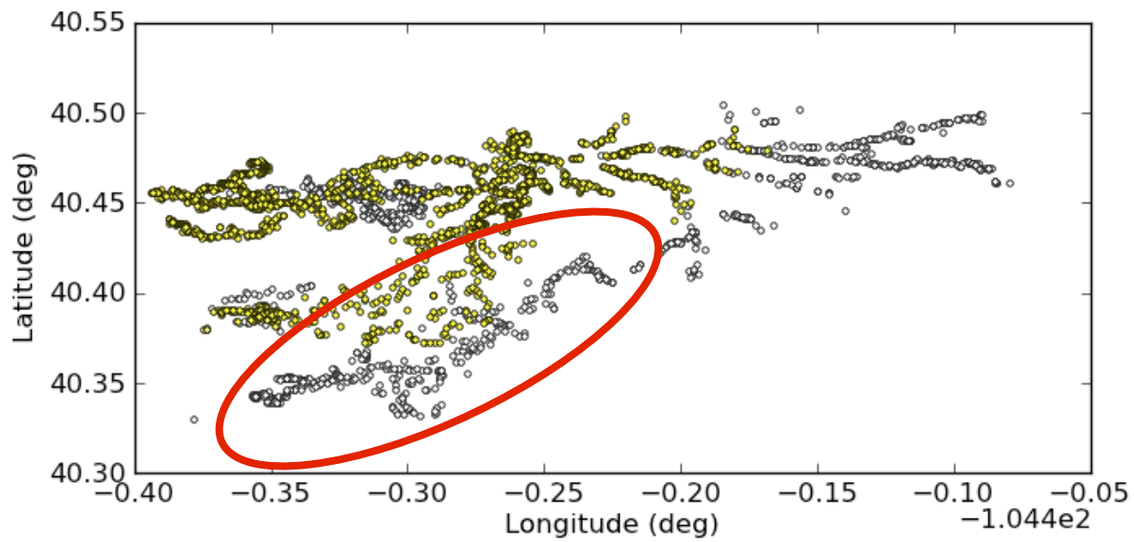
```
    label_aggregator = aggregate_ids(create_flash_objs(out, data))
    clusterer = cluster_chunk_pairs(label_aggregator, min_points=min_points)
    # Maximum 3 s flash length, normalized to the time separation scale
    chunker = chunk(XYZT[:,-1].min(), 3.0/.15,  clusterer)
    stream(XYZT.astype('float32'),chunker)
```

RESULTS: BOTH ALGORITHMS HAVE TROUBLE

Original —2 clusters

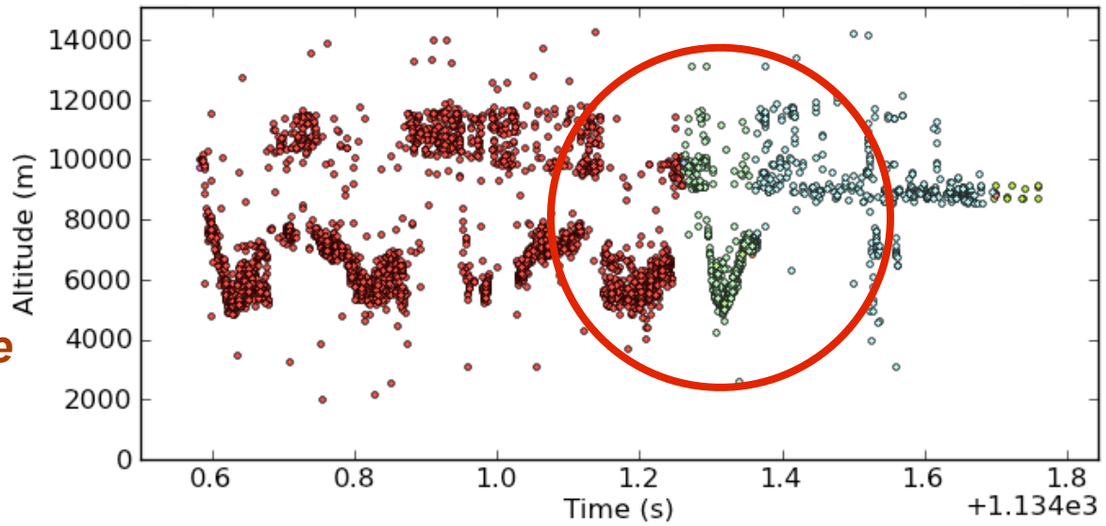


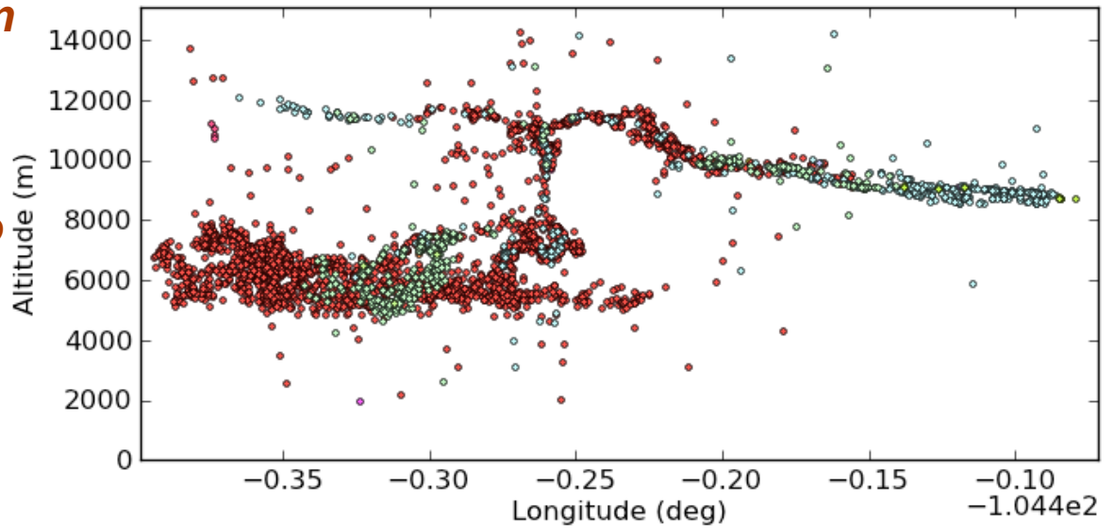


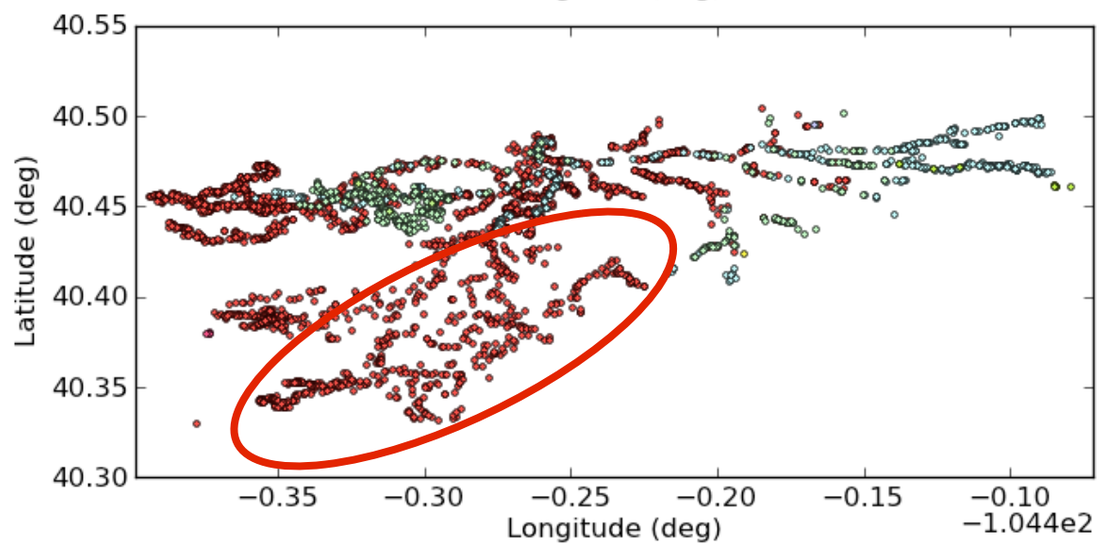


For this single flash, both algorithms have trouble with extension into areas that were previously active, but go quiet for a bit

sklearn —3 clusters







SUMMARY: NEW ALGORITHM

- At least no worse than old algorithm
  - *When it fails, it does so in similar ways*
  - *Point of this talk is not algorithm correctness*
- Advantages
  - *Clustering developed by algorithm professionals*
    - Algorithm (and not results using algorithm) has been peer reviewed
  - *Lets me focus on my area of expertise*
  - *No question about sharing with colleagues since built on open code*
- Python as a home for reference implementations of solved algorithms in computer science
  - *Minimizes time needed to field the right algorithm once it's identified*
    - dictionaries, arrays, machine learning, etc. …
    - reliable, obvious, documented

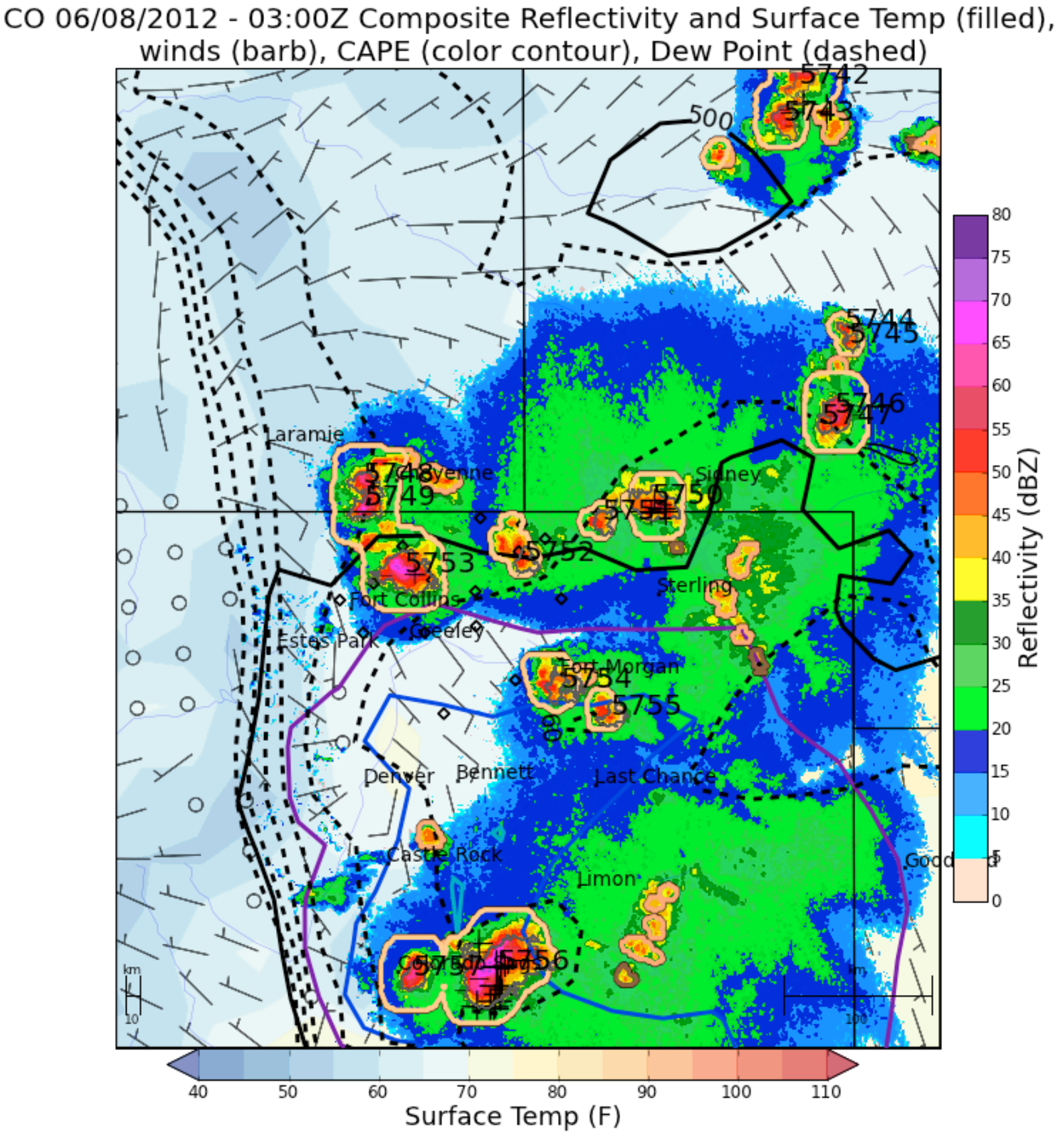
Multiple layers of open source

MULTIPLE LAYERS OF OPEN SOURCE

- New code shared with graduate student (Brody Fuchs) at Colorado State

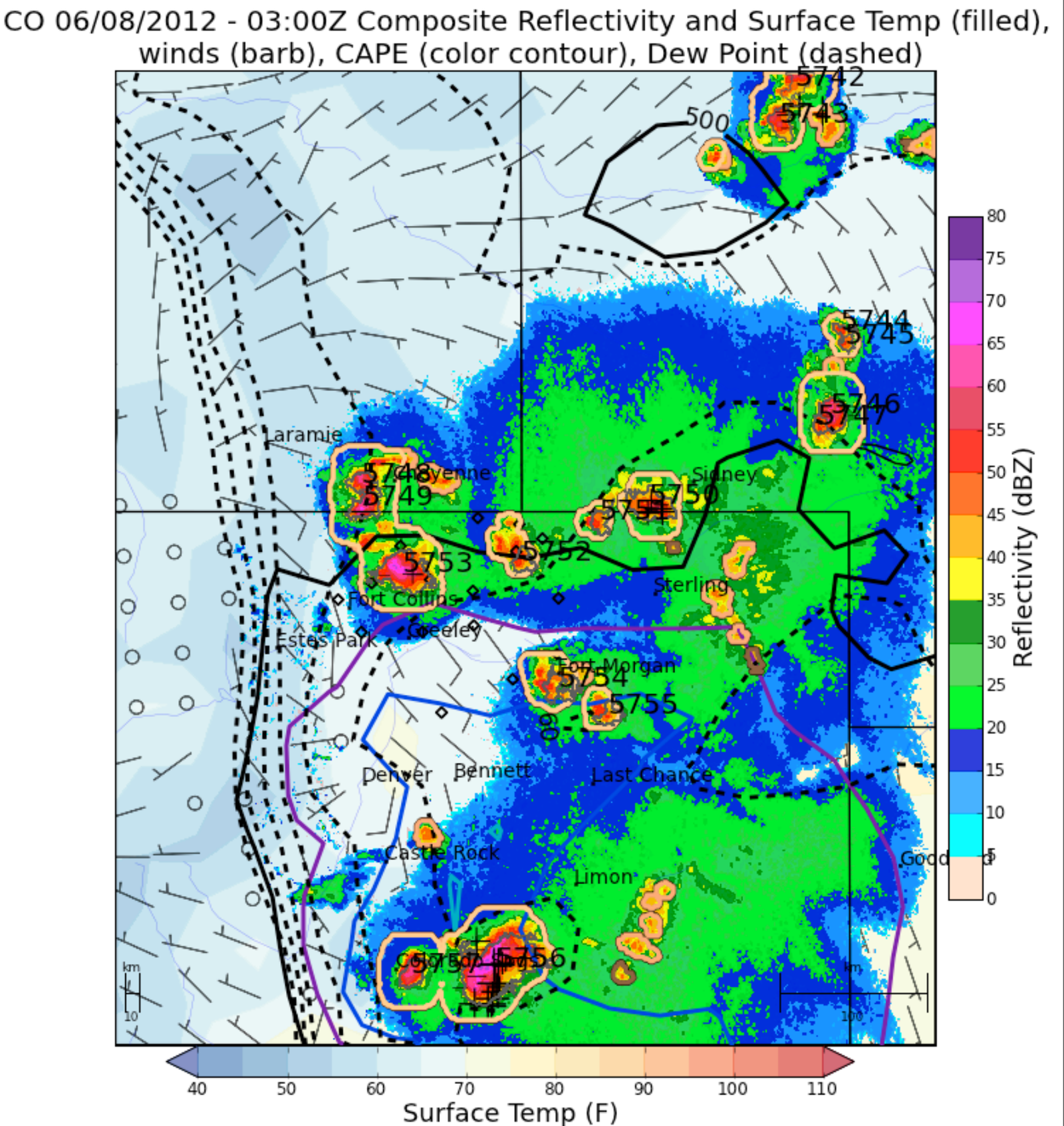
MULTIPLE LAYERS OF OPEN SOURCE

- New code shared with graduate student (Brody Fuchs) at Colorado State
  - *Compared DBSCAN flash sorting to another (third) algorithm*



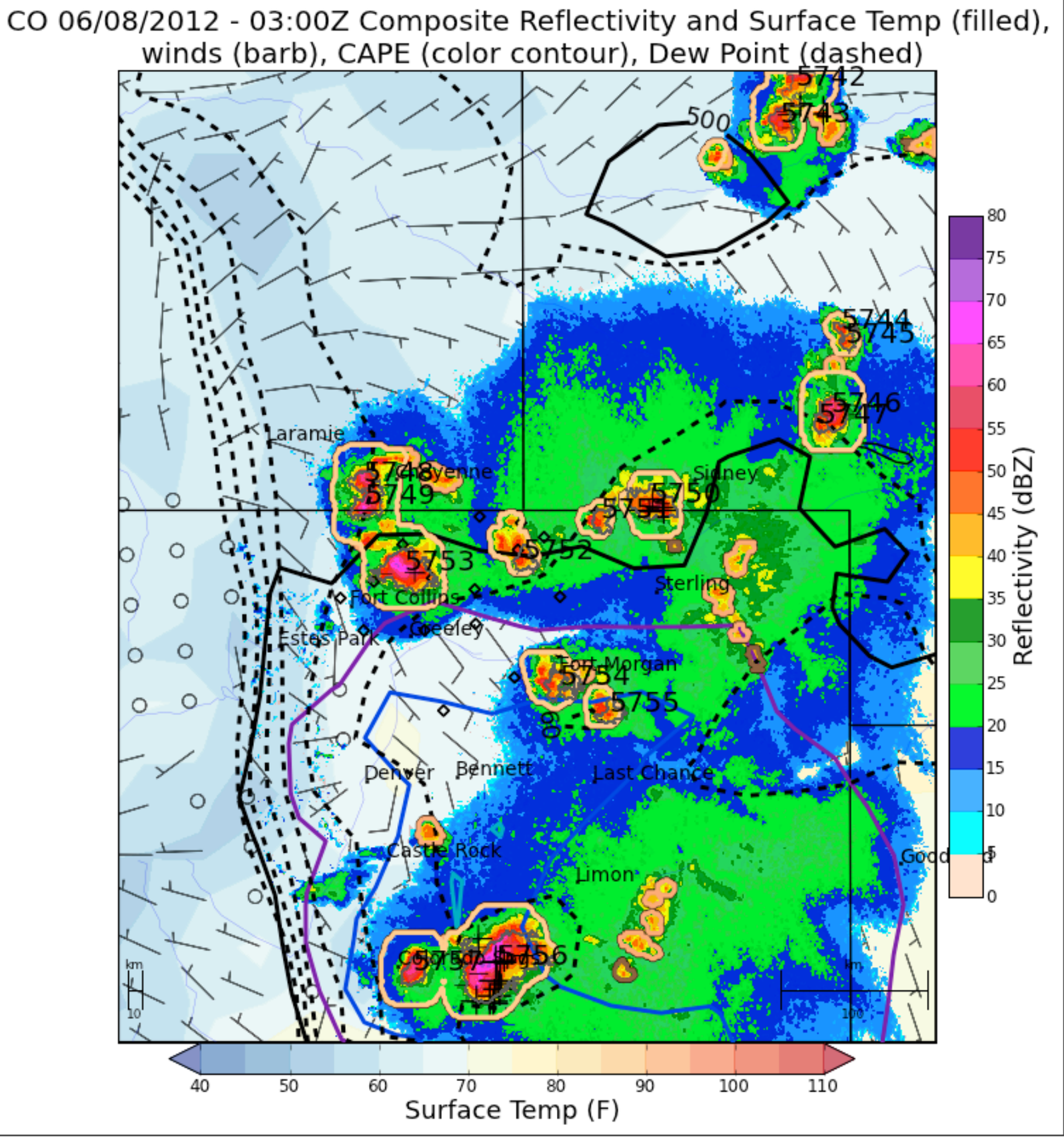
MULTIPLE LAYERS OF OPEN SOURCE

- New code shared with graduate student (Brody Fuchs) at Colorado State
  - *Compared DBSCAN flash sorting to another (third) algorithm*
  - *Within 10%*



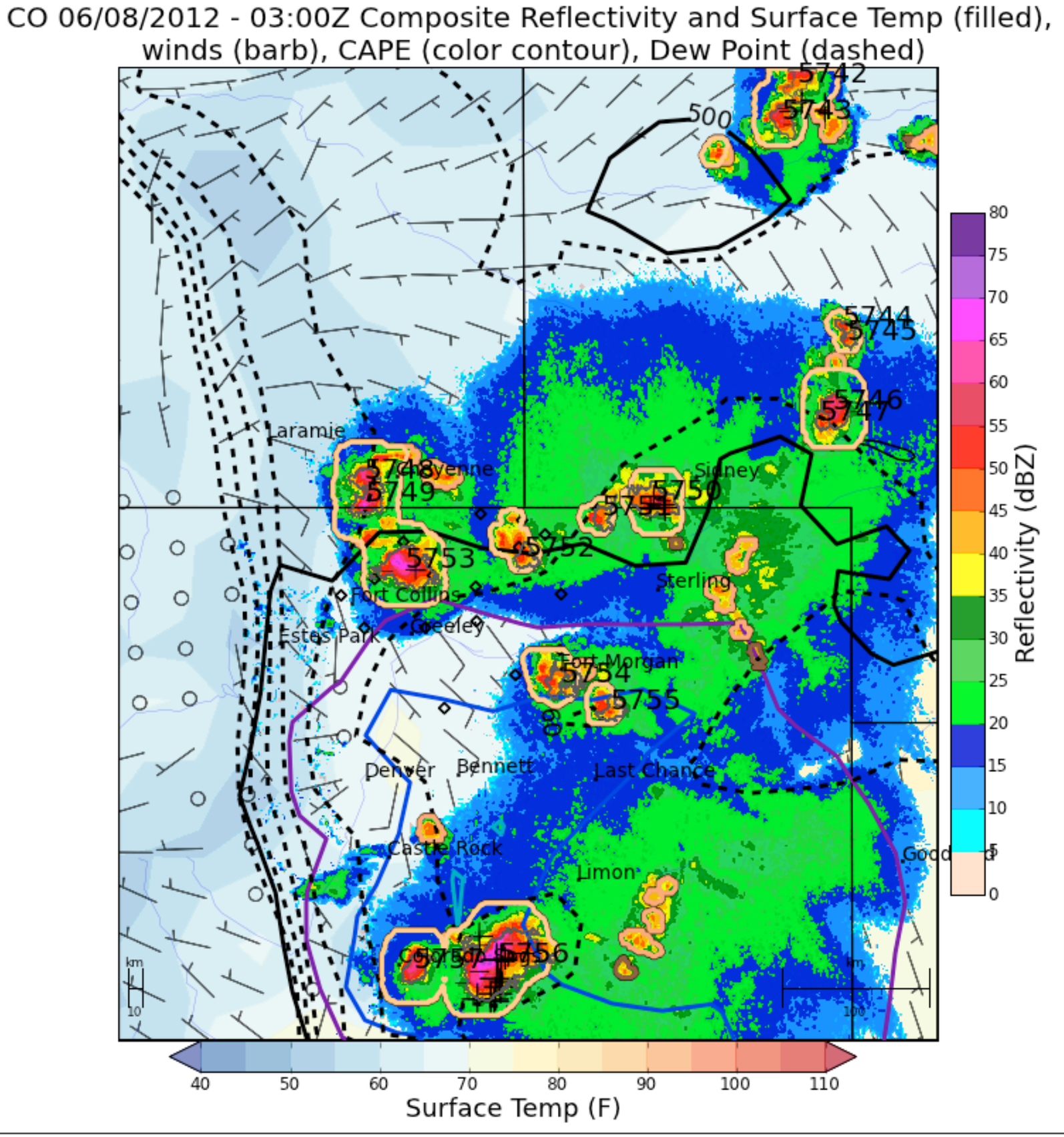
MULTIPLE LAYERS OF OPEN SOURCE

- New code shared with graduate student (Brody Fuchs) at Colorado State
  - *Compared DBSCAN flash sorting to another (third) algorithm*
  - *Within 10%*
- Modified my lmatools to work with his existing storm cell tracking code



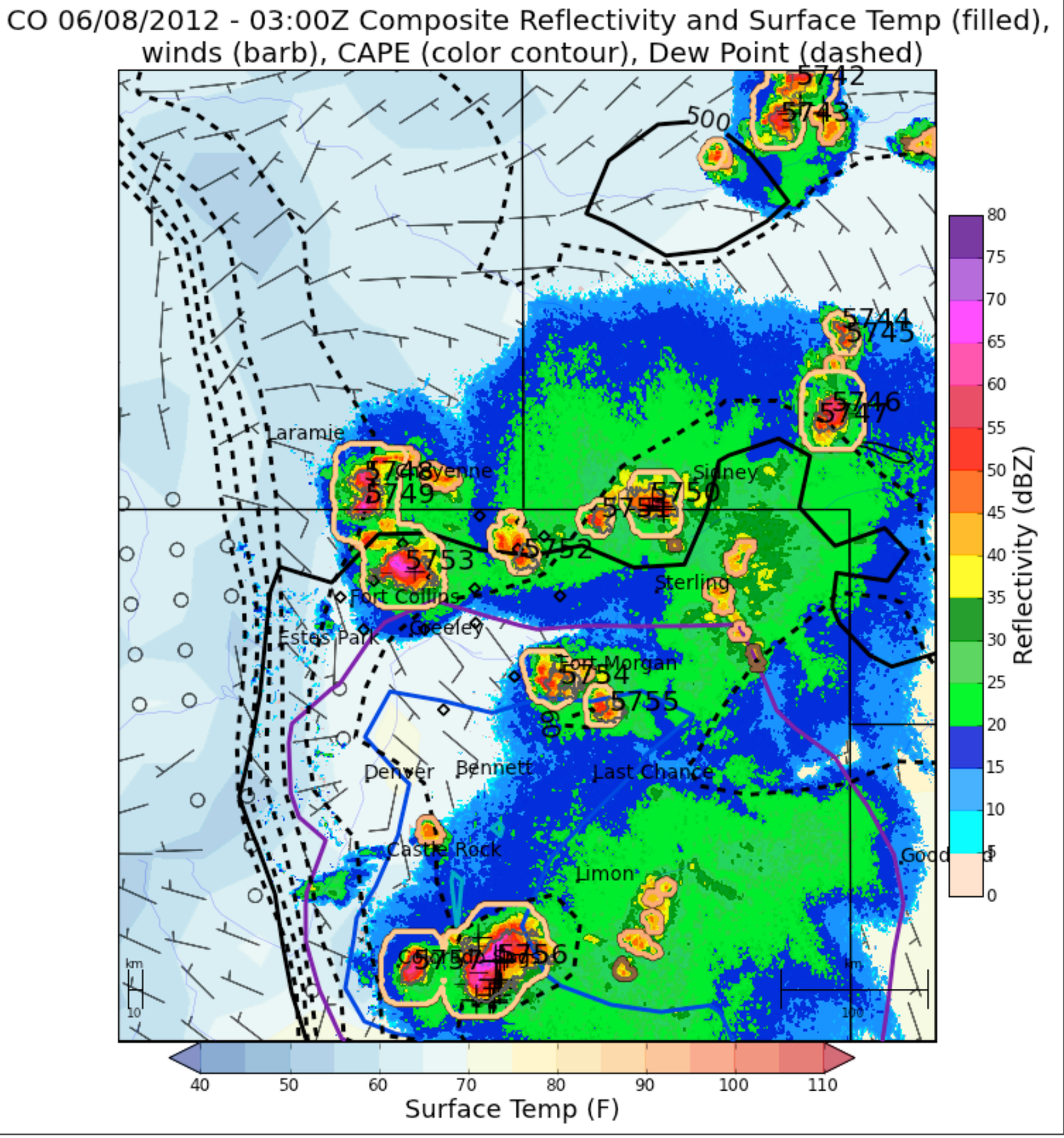
MULTIPLE LAYERS OF OPEN SOURCE

- New code shared with graduate student (Brody Fuchs) at Colorado State
  - *Compared DBSCAN flash sorting to another (third) algorithm*
  - *Within 10%*
- Modified my lmatools to work with his existing storm cell tracking code
  - *Identified some bad assumptions I made during implementation*



MULTIPLE LAYERS OF OPEN SOURCE

- New code shared with graduate student (Brody Fuchs) at Colorado State
  - *Compared DBSCAN flash sorting to another (third) algorithm*
  - *Within 10%*
- Modified my lmatools to work with his existing storm cell tracking code
  - *Identified some bad assumptions I made during implementation*
  - *Expect to integrate these generalizations to make lmatools more robust in the future*



PIPELINE EXTENSIONS

- More generally, the streaming pipeline can be triggered to do subsetting/re-transformation of data
  - *e.g., after adjusting one plot limits in two of four dimensions when panning a map view*
    - demo developed during last year's SciPy sprints
    - https://github.com/deeplycloudy/MPLIPy
    - my project for this week
- Datasets sit at the head-end of the pipe
  - *wait for message to trigger push of data*
- Pipeline can branch to multiple destinations
- Pipeline outlets are often "views" of data
  - *plots, data files, etc.*

References

Code shown in this talk:

- https://bitbucket.org/deeplycloudy/lmatools/
  - *stream and flashsort modules, coordinateSystems.py*

Coroutines

- Beazley, D. and B.K. Jones (2013): *Python Cookbook*, 3rd ed., 687 pp.