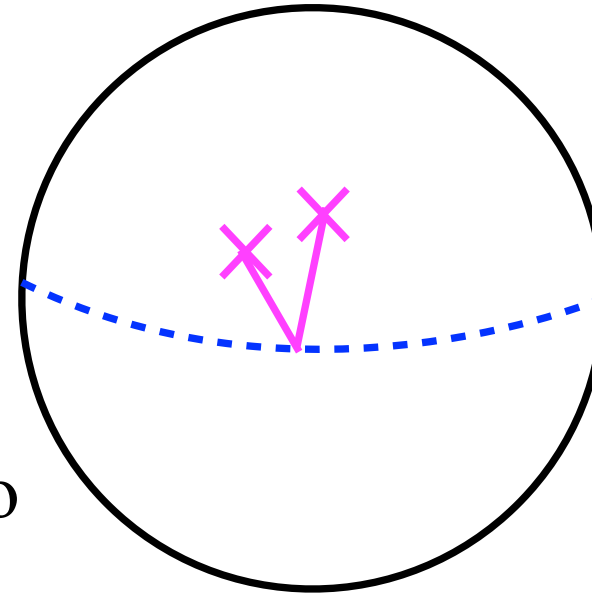
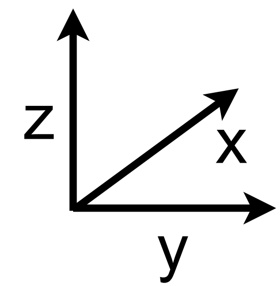
Distances on the Sphere

Angular distances on the sphere

- A few lectures ago, we discussed a problem where a star is 42.5° to the west of zenith (due to the Earth's rotation) and 23° south of zenith (due to the star's declination)



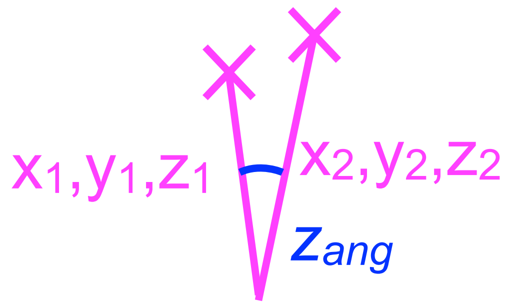
- I asked how we might combine this to find the airmass of the star
  - airmass depends on the zenith angle



- To proceed one would determine the (x,y,z) Cartesian coordinates 42.5° west of zenith, the (x,y,z) coordinates 23° south of zenith, and then use the dot product to find the angle between those vectors

Angular distances and the dot product

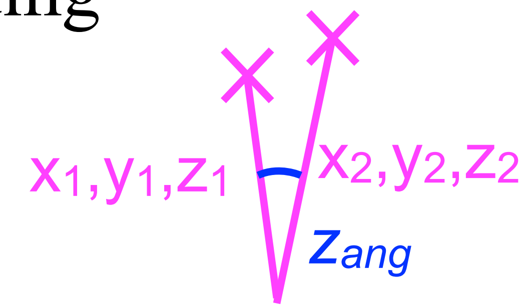
- For instance, say zenith is at (α_1,δ_1) = $(20^h24^m59.9^s, 10^o6'0'') = (306.25^o, 10.1^o)$



- If (α_2, δ_2) is 42.5^o west and 23^o south of zenith then $(\alpha_2,\delta_2) = (263.75^o,-12.9^o)$
- So, $(x_1,y_1,z_1) = (0.5821462,-0.7939473,0.1753667)$ $(x_2,y_2,z_2) = (-0.10612625,-0.96896677,-0.22325012)$
- Now, by definition of the dot product
  - $\mathbf{a}.\mathbf{b} = |a||b|\cos(z_{ang})$
  - where $(x_1,y_1,z_1).(x_2,y_2,z_2) = x_1x_2 + y_1y_2 + z_1z_2$
  - and $|(x_1,y_1,z_1)| = (x_1^2 + y_1^2 + z_1^2)^{1/2}$
- The zenith angle z_{ang} is then straightforward to find

Angular distances and the dot product

- I have cast this problem in terms of finding the airmass for arbitrary observations
- But this is a very general problem for astronomical observations

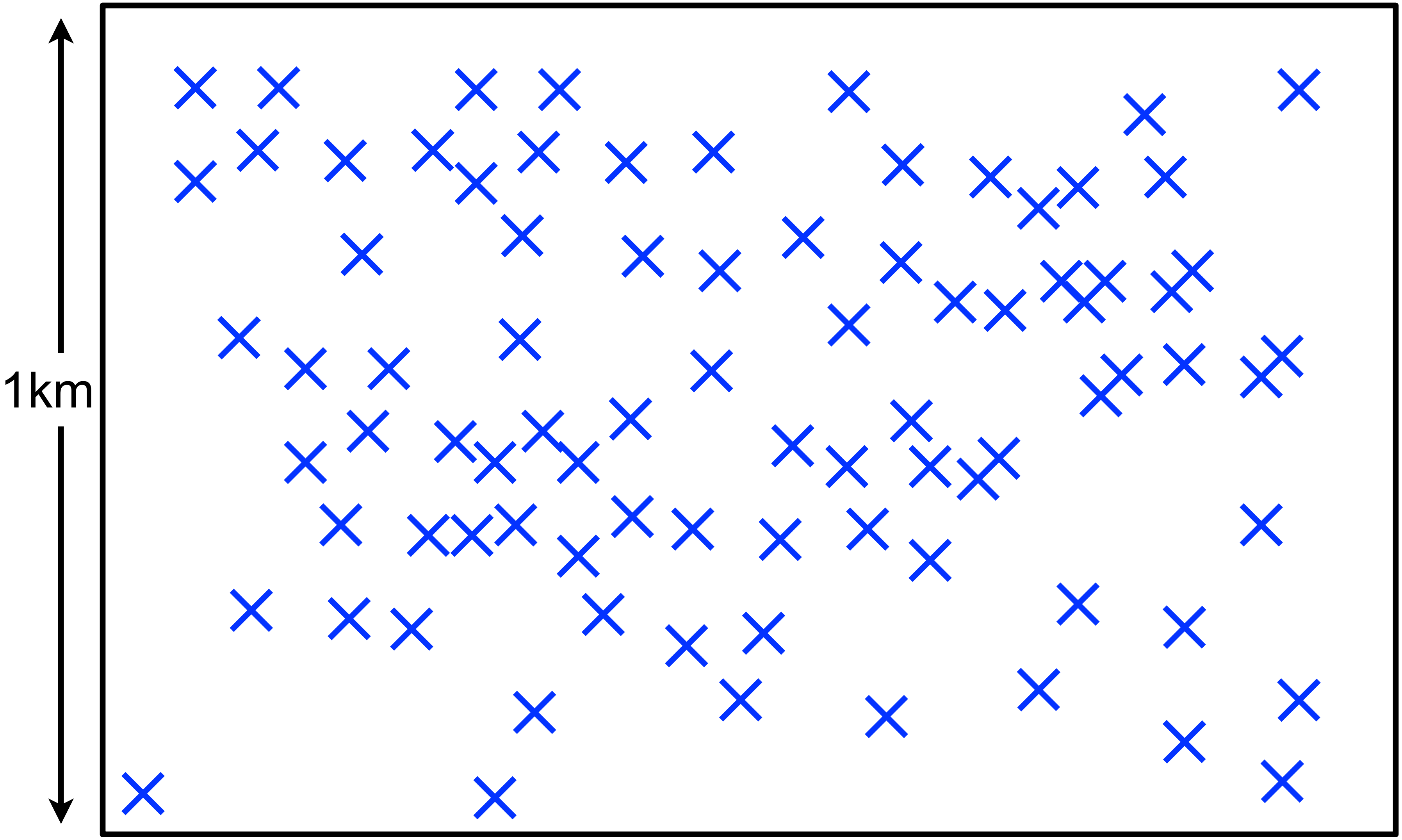


- Another common variation would be to find the angular distance between two stars (or other astronomical objects) with different (α,δ)
- The simplest approach will always be to convert to Cartesian coordinates and then use the dot product to find the angle between those coordinates
- Or, to write code to do this in the general case (or use code that someone else has written)

Multiple distances on the sphere

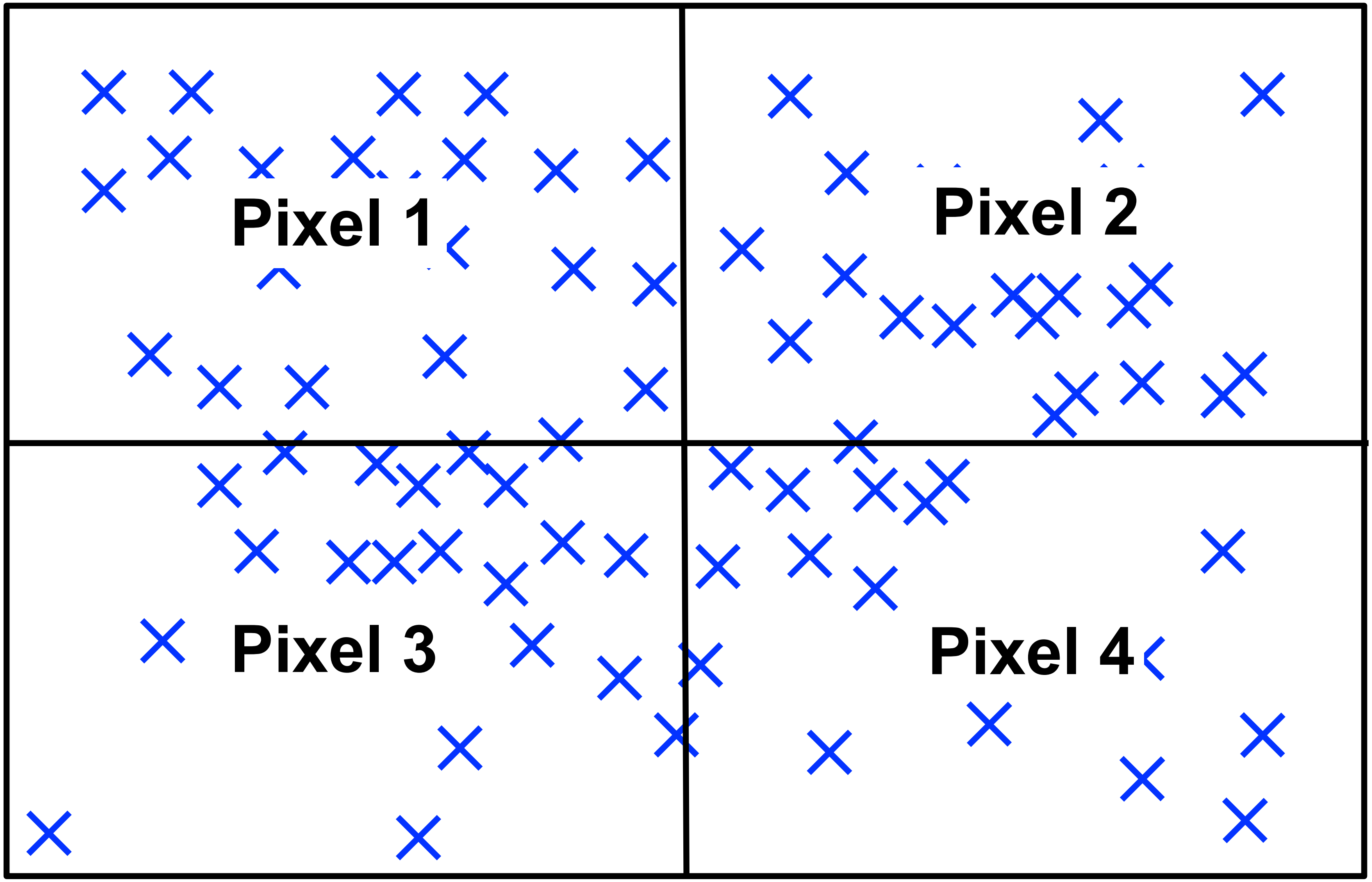
- Finding multiple distances on the sphere can quickly become difficult
  - consider a survey of 1 million galaxies, and you need the distances between all of them
    - this is (1million x 999999)/2 distances !
- Often, you won't care about all ~5 x 10^{11} distances
  - you may only want to measure the distances between objects that are close in the sky
    - say within a 1^{o} x 1^{o} field of a telescope
- This is where tree schemes and indexing schemes become very useful

The problem, in a nutshell

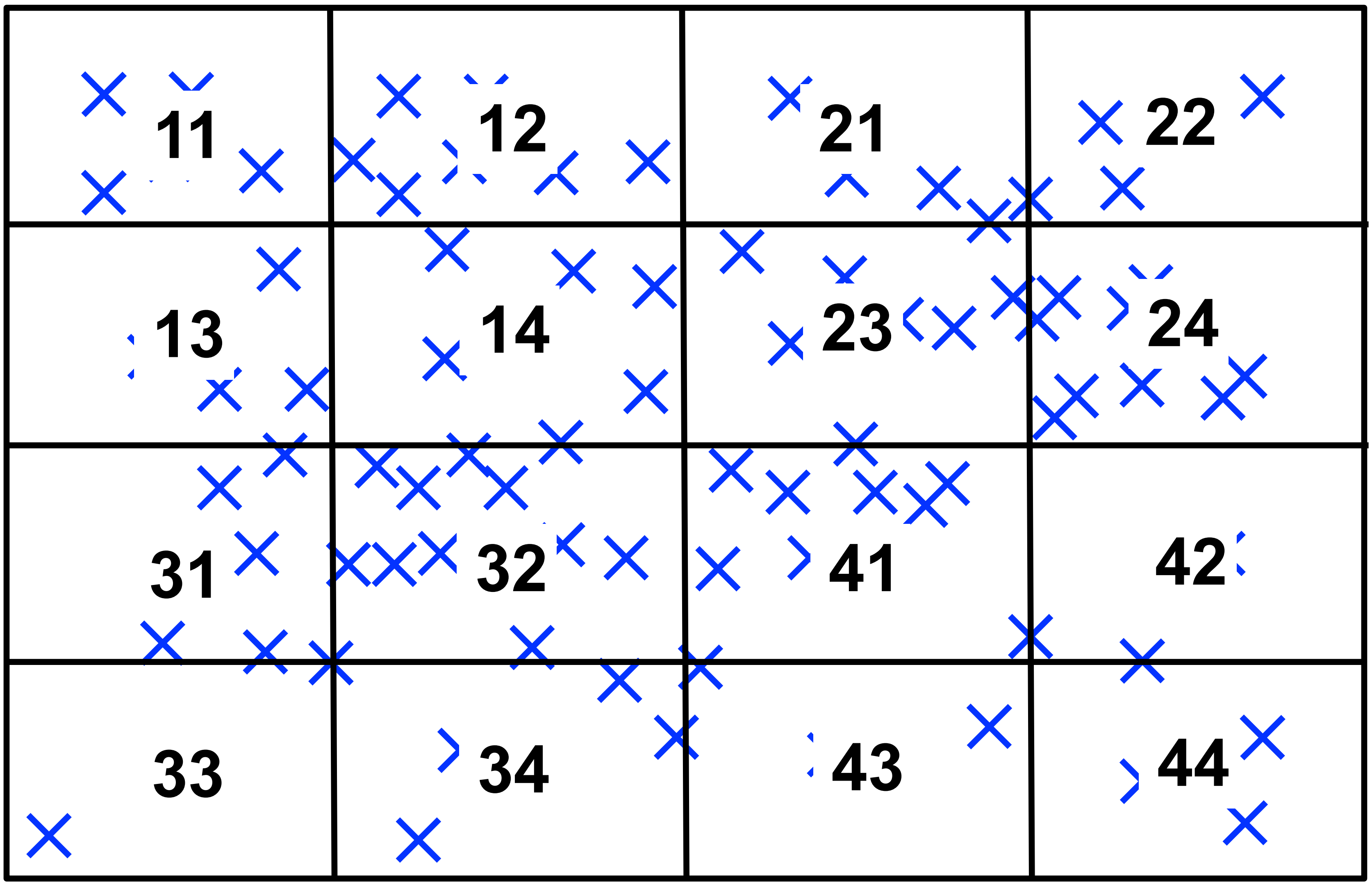


Find all pairs of points that are separated by < 100m

The quad tree



The quad tree



The quad tree

- Each point in a pixel is assigned the pixel name
  - so, the 3 points in pixel 22 are all labeled as 22
- As you progress hierarchically down the tree to the next “level”, you quadruple the number of pixels
  - and the label or “index” grows by one integer
- Eventually, each point has a unique index (is in its own pixel) and you can stop building the tree
  - if empty pixels are then discarded, you have exactly as many pixels as points
- The pixel size is known at each level, so if only points separated by, e.g., 100m, are needed, then only some subset of points in adjacent pixels need to be considered

The quad tree

- Tree structures are desirable for a number of reasons
- Recursive bisection is a very rapid algorithm
- The indexing is simple and easy to store as an integer
- Because each dimension is always being halved, the area and/or dimensions of each pixel are easy to track
  - which means that the separation between pixels at each level is easy to calculate
- Several sophisticated tree structures are used for indexing in astronomy, with one goal being to rapidly find adjacent points in a large amount of data
- The Hierarchical Triangular Mesh (HTM) is one such scheme (a full explanation is linked from the syllabus)

Coordinate matching with astropy.coordinates

- A useful routine in *astropy* that uses a tree scheme to rapidly perform distance measurements on the sphere is the *search_around_sky* procedure
- *search_around_sky* uses a *k*-dimensional (or *k*-d) tree to pixelate the sky
  - a *k*-d tree, essentially, recursively bisects a dataset perpendicular to each coordinate axis (e.g. bisects in *Cartesian x, y, z* in turn; see the *syllabus link*)
  - bisections occur close to the *median* data point along a coordinate axis, so that, on each split, about half of the data ends up in each "child" pixel
- Coordinate matching is easily my single most-used routine for data manipulation in large surveys

Angular separations between sets of points

- One common use of *separation* is to take two sets of object coordinates in arrays [ra1], [dec1] and [ra2], [dec2] and find angular distances between the objects
- The syntax for this case is, e.g.
  - *ra1, dec1 = [8.,9.,10.]*u.degree,[0.,0.,0.]*u.degree*
  - *ra2, dec2 = [18.,19.,21.]*u.degree,[0.,0.,0.]*u.degree*
  - *c1 = SkyCoord(ra1, dec1, frame=‘icrs’)*
  - *c2 = SkyCoord(ra2, dec2, frame=‘icrs’)*
  - *c1.separation(c2)*
- The result, here is *<Angle [10., 10., 11.] deg>*
  - *because [8.,0.] is separated from [18.,0.] by 10^o and [10.,0.] is separated from [21.,0.] by 11^o etc.*

Matching a set of points to another set of points

- You can also match *all* of a set of points to *all* of a second set using SkyCoord's *search_around_sky* method
- The syntax for *search_around_sky* is, e.g.
  - *ra1, dec1 = [8.,9.,10.]*u.degree,[0.,0.,0.]*u.degree*
  - *ra2, dec2 = [18.,19.,21.]*u.degree,[0.,0.,0.]*u.degree*
  - *c1 = SkyCoord(ra1, dec1, frame='icrs')*
  - *c2 = SkyCoord(ra2, dec2, frame='icrs')*
  - *id1, id2, d2, d3 = c2.search_around_sky(c1, 9.1*u.deg)*
- The result is *id1 = [1, 2, 2]* and *id2 = [0, 0, 1]*
  - *because [9.,0.]* ***is*** *separated from [18.,0.] by < 9.1^o* ***and*** *[10.,0.] is separated from [18.,0.] by < 9.1^o* ***and*** *[10.,0.] is separated from [19.,0.] by < 9.1^o*

Matching one point to a set of points

- A second use of *separation* is to take a single point in the sky and find all objects within some distance of it
- This is useful, e.g., if you are following-up a "circular" area on the sky with a spectroscopic plate
- Say I'm placing a 1.5^o radius plate at $\alpha = 20^o$, $\delta = 0^o$ and I want to know which objects in the sky will fall on it:
  - *ra1, dec1 = [20.]*u.degree,[0.]*u.degree*
  - *ra2, dec2 = [18.,19.,21.]*u.degree,[0.,0.,0.]*u.degree*
  - *c1 = SkyCoord(ra1, dec1, frame='icrs')*
  - *c2 = SkyCoord(ra2, dec2, frame='icrs')*
  - *w = np.where(c1.separation(c2) < 1.5*u.degree)*
- Here, *w* is *array([1, 2])* because the *1st* and *2nd* points are within 1.5^o of $\alpha = 20^o$, $\delta = 0^o$

Python tasks

1. Use *astropy.coordinates.SkyCoord* to convert (α_1,δ_1) = (263.75°, -12.9°) and (α_2,δ_2) = ($20^h24^m59.9^s$,10°6'0") to Cartesian coordinates, and hence use the *dot product* to determine the angle between these coordinates
   - Use *SkyCoord's separation* method to check your result
2. Use *np.random* to populate the area of the sky between $\alpha = 2^h$ and $\alpha = 3^h$ and $\delta = -2^o$ and $\delta = 2^o$ with two different sets of 100 random points (α_1,δ_1) and (α_2,δ_2)
   - plot your two sets of points, in different colors and using different symbols
3. Use the *search_around_sky* method to find which of your points are within 10' of each other (remember 1' is 1°/60)
   - plot those points on the same plot in a third color

Python tasks

4. Combine your two sets of points into one array (e.g., *ra = np.append(ra1,ra2)* and *dec = np.append(dec1,dec2)*
   - plot them all again using the same symbol and color

5. You are observing objects in your mock data set using a spectroscopic plate of 1.8° radius that is to be placed at $(\alpha,\delta) = (2^h20^m5^s, -0^o6'12'')$
   - Use *SkyCoord's separation* method to find all of the points that will fall on your plate and plot this subset of points (on the same plot as before) in a different color
   - Remember the usefulness of *np.where*