Visualizing high proper motion sources using Gaia

Author

C. E. Brasseur

Learning Goals

1. Use Astroquery to access GAIA data
2. Properly format an ADQL query
3. Use the Astropy SkyCoord class to represent stars with known proper motion and radial velocity values
4. Visualize in 2- and 3D star positions on the sky
5. Using the matplotlib animation library, visualize the changing position of stars through time

Keywords

astropy.coordinates, astroquery.gaia, matplotlib.animation

Companion Content

- Gaia Archive
- Gaia Astroquery Module
- Gaia source catalog column definitions
- SkyCoord Documentation
- Matplotlib 3D Plots

- Matplotlib Animation

Summary

In this tutorial we will use the astroquery package to get a list of stars in the Gaia catalogue with high proper motions and measured radial velocities. We will then visualize the resulting star positions on a 2D Aitoff-projected grid, and in 3 dimensions. Next we will use the functionality built into the Astropy coordinates framework to project the path of our collection of stars forward in time. Finally, we will visualize the movement of the stars by creating an animation within matplotlib.

Imports

We will use `astroquery.gaia` for data access, `astropy.coordinates` for sky position representation and manipulation, and `matplotlib` for visualization.

```
import warnings  # So we can suppress expected warnings
import numpy as np

from astroquery.gaia import Gaia  # For data access

from astropy.coordinates import (
    SkyCoord,
)  # For storing a manipulation object sky positions
import astropy.units as u

# For plotting
import matplotlib.pyplot as plt
import matplotlib.animation as animation

plt.rcParams["animation.html"] = (
    "jshtml"  # To make the animations render correctly in the notebook
)
%matplotlib inline
```

Query Gaia

The `astroquery.gaia` module uses the Astronomical Data Query Language (ADQL) to query its various databases. Here we query the `gaia_source` table in the Gaia Data Release 3 database, selecting a subset of columns to be returned and adding some conditions for the data.

```
adql_query = (
    "SELECT source_id,ra,dec,pmra,pmdec,radial_velocity,distance_gspphot "  # The co
    "FROM gaiadr3.gaia_source "  # The table we are querying
    "WHERE pm>=1000 "  # Only return rows where the proper motion is >= 1000 mas/yr
    "AND distance_gspphot<=40 "  # Only return rows where photomentric distance is <
    "AND radial_velocity IS NOT NULL)"  # Only return rows where there is a radial v
)

job = Gaia.launch_job(adql_query)

gaia_table = job.get_results()
```

```
print(f"Number of objects found: {len(gaia_table)}\n")

print("First 5 rows:")
gaia_table[:5]
```

```
Number of objects found: 218

First 5 rows:
```

Table length=5

source_id	ra	dec	pmra
	deg	deg	mas / yr
int64	float64	float64	float64
6021430250773348736	245.01042208462846	-37.524610446713915	-748.0393660219934
139860666488305664	43.0351018418911	34.384950446682915	995.8045303279958
577602496345490176	133.78195701408768	1.5418558693942335	44.94399530668833
4542821410453751680	261.9110282533942	14.482331245176582	-1118.0084498701985
5918660719983560192	266.6334444727895	-57.32506180218072	-1116.9094993763981

For the rest of this notebook we are going to take advantage of the functionality built into Astropy's `SkyCoord` class. In order to do this we are going to combine not only the star positions (including distance), but their proper motions and radial velocities to create a `SkyCoord` object that not only knows where the stars are at present but how they are moving.

```
gaia_table["position"] = SkyCoord(
    gaia_table["ra"],
    gaia_table["dec"],  # star sky positions
    distance=gaia_table["distance_gspphot"],  # star distances
    # proper motion, conventional format multiplies the ra proper motion
    # by the cosine of the declination
    pm_ra_cosdec=u.Quantity(gaia_table["pmra"]) * np.cos(gaia_table["dec"]),
    pm_dec=gaia_table["pmdec"],
    # Radial velocity
    radial_velocity=gaia_table["radial_velocity"],
)
```

Visualize star positions

In this section we use the stellar positions at the present time to visualize the spread of stars on this sky. To do this, we write a function that takes in a list of object coordinatates and builds a figure that visualizes the stars in two ways: projected onto the plane of the sky, and in three dimensional space.

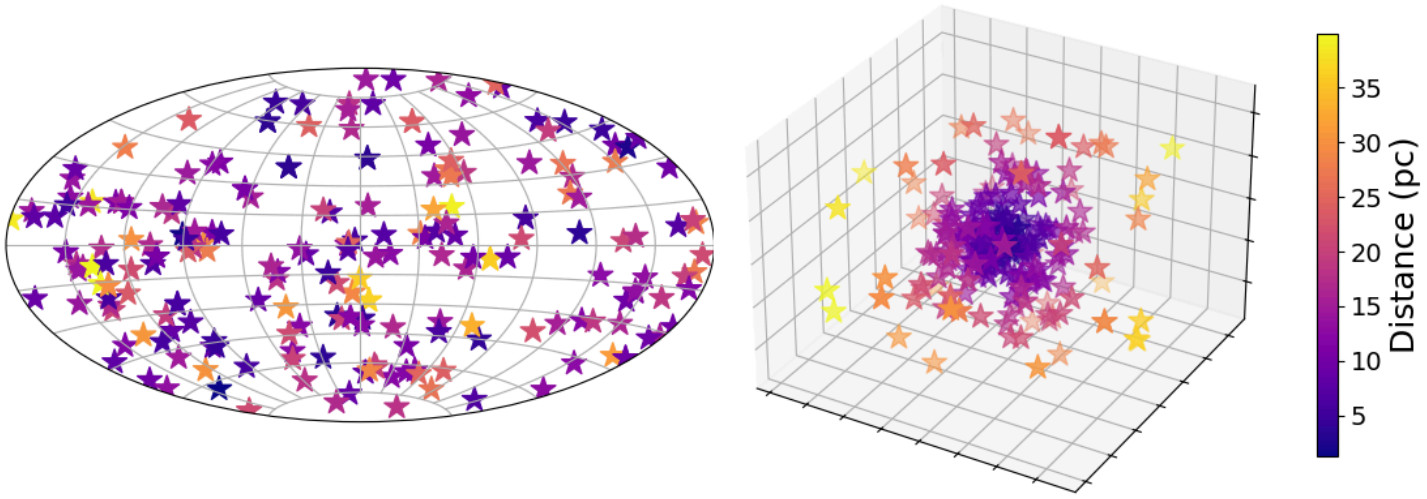
```
def star_plot(star_coords, vmin=None, vmax=None):
    """
    Take a list of stars as SkyCoord objects and build a figure showing the stars pr
    the plane of the sky using the Aitoff projection, and in three dimensional space
    on the Earth. In both plots the points are colored by distance (from Earth).

    Parameters
    ----------
    star_coords : Astropy SkyCoord object
        The coordinates of the objects to be plotted
    vmin, vmax : float
        Optional min and max values in pc for the colormap used to color the plot by
        If not set they will be the minimum and maximum distances of star_coords.

    Returns
    -------
    response : matplotlib figure

    """

    # Set the min and max values for the colormap if not given
    if not vmin:
        vmin = star_coords.distance.min().value

    if not vmax:
        vmax = star_coords.distance.max().value

    # Initialize the figure
    fig = plt.figure(figsize=(14, 7))

    # Sky Projection plot
    ax1 = fig.add_subplot(1, 2, 1, projection="aitoff")  # Add left subplot

    # Turn off the axis ticks and labels, and turn on the plot grid
    ax1.set_xticklabels([])
    ax1.set_yticklabels([])
    ax1.grid(True)

    # Plot the stars, colored by distance
    ax1.scatter(
        star_coords.ra.wrap_at(180 * u.deg).radian,
        star_coords.dec.radian,
        c=star_coords.distance,
        vmin=vmin,
        vmax=vmax,
        marker="*",
        s=200,
        cmap="plasma",
    )

    # 3D plot
    ax2 = fig.add_subplot(1, 2, 2, projection="3d")  # Add right subplot
```

```
    # Turn off axis ticks and labels
    ax2.set_xticklabels([])
    ax2.set_yticklabels([])
    ax2.set_zticklabels([])

    # Plotting the Earth as a black circle
    ax2.scatter([0], [0], [0], s=100, c="k", marker="o")

    # Plot the stars, colored by distance
    pc = ax2.scatter(
        star_coords.cartesian.x,
        star_coords.cartesian.y,
        star_coords.cartesian.z,
        c=star_coords.distance,
        s=200,
        vmin=vmin,
        vmax=vmax,
        marker="*",
        cmap="plasma",
    )

    # Adding the colorbar
    cbar = fig.colorbar(pc, shrink=0.6, location="right")
    cbar.ax.tick_params(labelsize=14)
    cbar.ax.set_ylabel("Distance (pc)", fontsize=18)

    # Remove extra space between the subplots
    fig.subplots_adjust(wspace=0)

    # This prevents getting an extra copy of the plot when you call the function
    plt.close()

    return fig
```

First we plot all the stars in our query result.

In the below plot we can see that we have good sky coverage, and the 3D plot clearly shows that we are indeed coloring the points by distance.

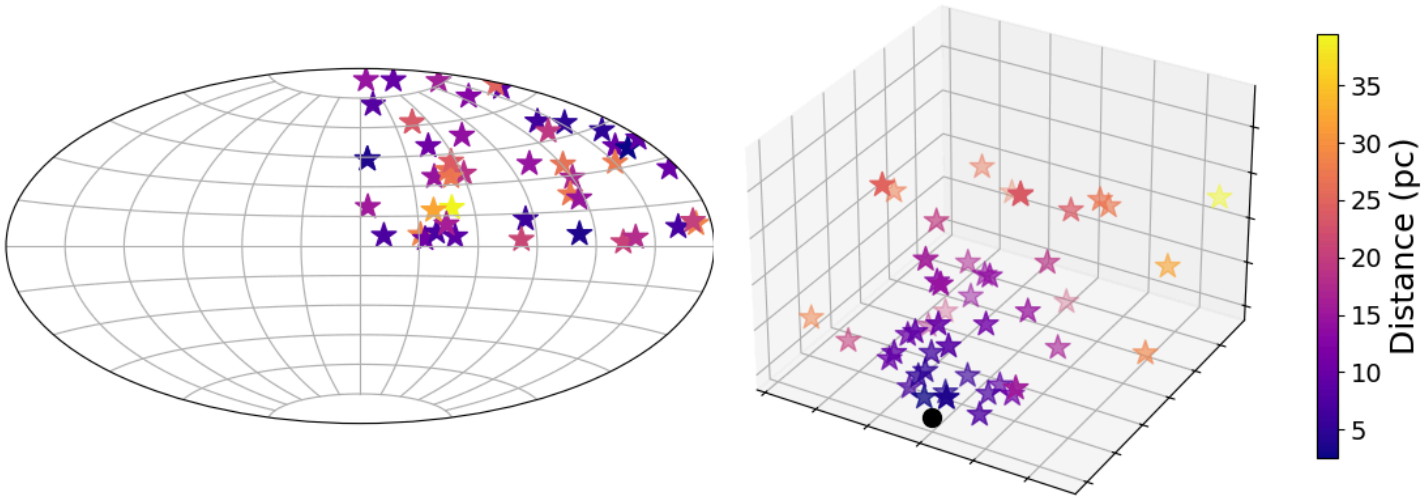
```
star_plot(gaia_table["position"])
```



We can also plot a subset of the stars, for example, below we plot only a single quadrant of the sky by imposing the condition that stars must have RA $< 180^\circ$ and Dec $> 0^\circ$.

Here we can clearly see in the plot on the left the single quandrant we are plotting, and on the right how those stars spread out from the Earth.

```
star_plot(gaia_table["position"][(gaia_table["ra"] < 180) & (gaia_table["dec"] > 0)]
```



Animate stellar trajectories

In the previous section we plotted the stars in their current positions. However, we also included in our `SkyCoord` object information about how the stars are moving, and in this section we will use that infromation to evolve their positions through time.

The function we use is `apply_space_motion`, which allows us to input a length of time (`dt`) and get back the coordinates for the object(s) after that span of time.

We will write a function to take the plot we created in the previous section and animate it, showing the trajectory of our stars as time passes.

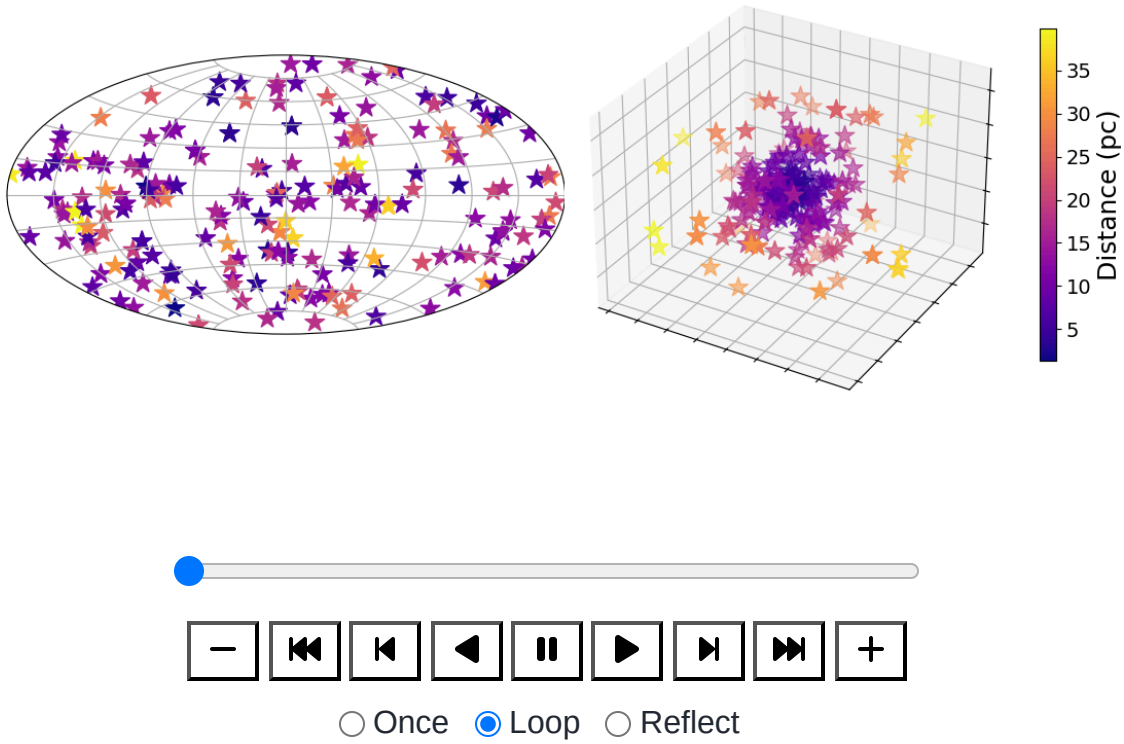
```
def star_animation(star_coords, evolution_time, steps, vmin=None, vmax=None):
    """
    Take a list of stars as a SkyCoord object and build an animation showing how the
    positions evolve over time.

    Two subplots are produced, the left showing the stars projected onto the plane o
    using the Aitoff projection, and the right in three dimensional space centered o
    In both plots the points are colored by distance (from Earth).

    Parameters
    ----------
    star_coords : Astropy SkyCoord object
        The coordinates of the objects you wish to plot
    evolution_time : float or Astropy Quantity
        The amount of time to evolve the stellar positions each step.
        If not specified as a quantity, years will be assumed.
    steps : int
        The number of time steps to take.
        The total amount of time the system will be eveolved for is evolution_time *
    vmin, vmax : float
        Optional min and max values for the colormap used to color the plot by dista
        If not set they will be the minimum and maximum distances of the input stars

    Returns
    -------
    response : matplotlib figure

    """

    # Make sure the evolution time is in years
    if not isinstance(evolution_time, u.Quantity):
        evolution_time *= u.yr

    # Create array of time deltas (all in relation to present time)
    dt_array = np.linspace(0, evolution_time * steps, steps, endpoint=False)

    # If vmin/vmax were not set we need to set them (if they are not set at all,
    # each step of the animation will have different colormap boundaries).
    if not vmin:
        vmin = star_coords.distance.min().value

    if not vmax:
        vmax = star_coords.distance.max().value

    def animate(iteration, dt_array, aitoff_scatter, cube_scatter):
        """
        This function handles updating the plot for each frame of the animation.

        Parameters
        ----------
        iteration: int
            Current iteration (frame number)
        dt_array : Quantity
```

```
        Array of time deltas
    aitoff_scatter : matplotlib PathCollection
        A matplotlib collection object that holds the 2D (left) plot data points
    cube_scatter : matplotlib Path3DCollection
        A matplotlib collection object that holds the 3D (right) plot data point
    """

    with warnings.catch_warnings():
        # Projecting more than 5 years into the future gives a "dubious year" wa
        # due to the unpredictability of leap seconds, we suppress this warning
        # we are not concerned with temporal accuracy to the second
        warnings.filterwarnings("ignore", message="ERFA function ")
        new_pos = star_coords.apply_space_motion(dt=dt_array[iteration])

    # set_offsets sets the point locations to the newly calculated coordinates
    aitoff_scatter.set_offsets(
        list(zip(new_pos.ra.wrap_at(180 * u.deg).radian, new_pos.dec.radian))
    )

    # set_array sets the colors of the points based on their new distances
    aitoff_scatter.set_array(new_pos.distance)

    # For the 3D plot we have to use the private property _offsets3d to update t
    cube_scatter._offsets3d = (
        new_pos.cartesian.x,
        new_pos.cartesian.y,
        new_pos.cartesian.z,
    )

    # set_array sets the colors of the points based on their new distances
    cube_scatter.set_array(new_pos.distance)

    return (
        aitoff_scatter,
        cube_scatter,
    )

# Call our star_plot function from the previous section to initialize the figure
fig = star_plot(star_coords, vmin=vmin, vmax=vmax)

# Get the matplotlib Collection objects that correspond to our sets of points
aitoff_scatter = fig.axes[0].collections[0]
cube_scatter = fig.axes[1].collections[1]

# Build the animation
ani = animation.FuncAnimation(
    fig,  # The figure we are animating
    animate,  # The function that updates the plot for each frame
    fargs=(dt_array, aitoff_scatter, cube_scatter),  # Args for the animate func
    frames=steps,  # The number of frames in our animation
    interval=100,
)  # Delay between frames in milliseconds

# This prevents getting an extra copy of the first frame of the plot when you ca
```

```
    plt.close()

    return ani
```

Here we show the evolution of the stellar positions over 20,000 years by animating 20 steps of 1000 years each.

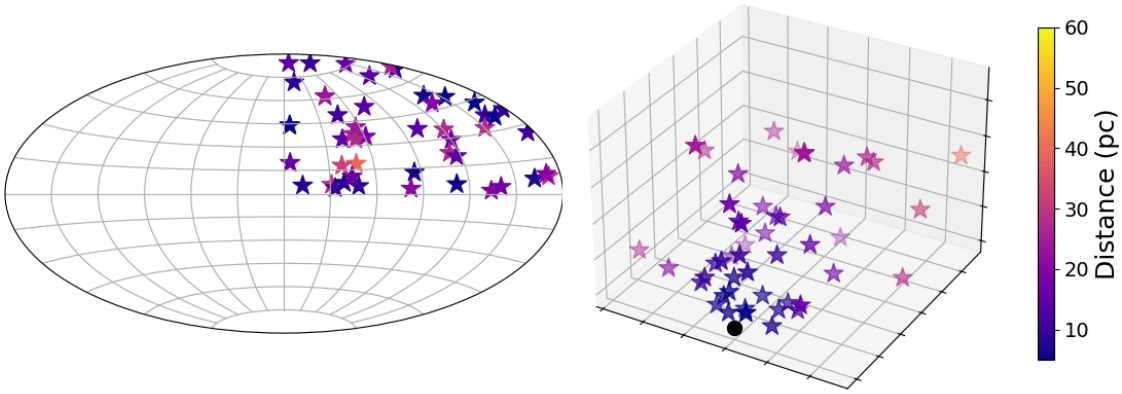
```
star_animation(gaia_table["position"], 1000 * u.yr, 20)
```



As we did for the still plots, we also plot a single quadrant of the sky by imposing the condition that stars must have RA $< 180^\circ$ and Dec $> 0^\circ$. We also increase the time step from 1,000 years to 10,000 years, meaning the full evolution time we animate is 200,000 years.

Because some of the stars are moving away from us and we are pushing further forward in time we set the colormap range to $4 - 60$ pc to cover the larger range of distances we will need to represent as the stars move.

```
star_animation(
    gaia_table["position"][(gaia_table["ra"] < 180) & (gaia_table["dec"] > 0)],
    10_000 * u.yr,
    20,
    vmin=5,
    vmax=60,
)
```



Once Loop Reflect