Exercises with Gaia RVS and AGAMA

Basic exercise Using one of three model Galactic potentials, compute actions, angles and frequencies for a sample of the RVS stars. Plot the density of stars in the (J_ϕ, J_r) and (J_z, Ω_z) planes. Understand the gross structures of the distributions you obtain. What fine structure can you see?

Secondary exercise Pick another potential and compare the distributions you obtain. Comment on differences in the (J_ϕ, Ω_z) distributions.

Tertiary exercise Plot the mean or median value of v_ϕ in each pixel of the (X, Y) phase plane, where $X = \sqrt{2J_z}\cos\theta_z$ and $Y = \sqrt{2J_z}\sin\theta_z$. What can you see? How does this relate to arXiv:1804.10196?

Code It's essential to first install AGAMA from https://github.com/GalacticDynamics-Oxford/Agama. Change my makefile so it points to where you've installed AGAMA.

I'm not a Python user, so I'm providing only C++ code for the exercises. This code needs to be linked to agama.so, which will be created when AGAMA is installed. If you want to use Python, you'll have to write wrappers for my code.

Once you've created appropriate data file, run `heid_ex.cpp`. This will include lots of AGAMA headers but also two files `star.h` and `starx.h` that define classes for stars. I provide example files described below, but you'll need to hack these to suit your own conventions regarding i/o, etc. My classes read and write data compressed using code you'll find in `compress.cpp`

The code in `heid_ex.cpp` picks a potential and then, using omp to speed up operations, it computes the AAvars for each star. Before you run this code you should read the RVS data, creating a `star` object for each read item, and write it in one of NPROC files, where NPROC is the number of cores you have (or a multiple thereof). The reads and writes should be handled by the `star` class. The code in `heid_ex.cpp` will produce NPROC files in which each star has AAvars and frequencies.

Example star classes My `star.h` has a method `bool readin()` that reads a running index ind, a Gaia id, Bayesian estimate of distance `s1`, an inverse parallax `opi`, etc, etc from an ASCII file with 32 columns. It also contains a method `void putit()` that writes to file a selection of the data in compressed form and a method `bool getit()` that recovers the star's data from the compressed file.

My `starx.h` has a creator `starx(star&,ActionAngles&,Frequencies&,InternalUnits&)` that takes in a star, its AAvars, frequencies and a structure that converts from AGAMA's internal units to kpc, km/s. It has another creator `starx(IFILE*)` that takes the star's data from a file. Finally, it has a method `void putit(FILE*)` that writes the data to file in compressed form.

You may want to replace my compression routines `void compress(FILE*,float*,int)`, `bool decompress(FILE*,float*,int)` to suit your i/o preferences. The latter returns true on success.

Example grid class You need to be able to plot densities of stars in planes, and for the tertiary exercise also plot the mean value of v_ϕ in cells in the (z, v_z) plane. Whatever you do, don't just produce a scatter diagram with a dot for each star! When data are rich, such plots destroy almost all information and in my view your referee should never let one pass.

You probably already have code to produce density plots, but in case not in `grid.h` and `grid.cpp` I provide example code that you can hack to suit your graphics system. The code defines a class `grid` that has a method `CIC(double x,double y)` that uses the cloud-in-cell algorithm to distribute to pixels a stars with coords (x, y), and a method `CIC(double f,double x,double y)` that does this plus computing the average value of f in cells. The creator is `grid(int nx,int ny,double minx,double maxx,double miny,double maxy)`. Once you've added every star to the grid, `plot_dens()` will plot the density of stars, and `plot_values()` will plot $\langle f \rangle$. You will have to supply/replace a routine `colourwh(float **m,int nx,int ny,float min,float max,float blank,int s)` that colours a $n_x \times n_y$ array of pixels according to the values pointed to by `m` with colours assigned to values in (`min,max`) pixels below `blank` being left white. The final integer controls whether a colour bar is added. You could supply or delete `mgobox`, which simply places a box with tickmarks on top of the coloured pixels.