A crash course in EXP

Mike Petersen

January 27, 2022

This tutorial is written to be compliant with EXP >= 7.6994, on the `devel` branch.

1 *N*-body simulation

The bread and butter of EXP is *N*-body simulation. Most of the software in the EXP tree is focused on the advancement of particles through time. A full description of all the assumptions, tuning parameters, and options will be found elsewhere. In this document, I'm simply going to get EXP started and troubleshoot any issues after the fact. We'll test everything interactively, but all options can be placed into a driver script.

Goals for this session:

1. Run test examples exercising different force methods in EXP .
2. Read EXP output files with `Python`.
3. Design new *N*-body experiments in EXP .

We'll follow a 'Frequently Asked Questions' format for the tutorial. The questions are listed below in blue. Examples where we run some test *N*-body case are listed below in red.

The material for this tutorial is contained in `NBodyTutorial.tar.gz`. You should unpack this (`tar -xvf NBodyTutorial.tar.gz`) in some directory on the same machine that has EXP . Once unpacked, you will see the following structure and input files:

```
NBodyTutorial/
    Binary/
        binary.yml
        binary.bods
        logpot.yml
    SimpleHalo/
        livesphere.yml
        halo.bods
        SLGridSph.model
    DiskHalo/
        galaxy.yml
        halo.bods
        SLGridSph.model
        disk.bods
        .eof.cache.run0
        satellite.yml
```

Believe it or not, this is all you need to get started and exercise quite a bit of the EXP *N*-body codebase.

What happens when I `mpirun exp`? EXP is designed to run with Open MPI[1], so one does not simply type `exp` at the command line[2]. Rather, one types `mpirun exp` to execute jobs using the Open MPI framework. You can `man mpirun` to read the gory details, but for our purposes, it is sufficient to simply specify the number of processors, e.g. `mpirun -np 12 exp -h` to use 12 processors and read the help message.

For the purposes of this tutorial, we'll be using `SLURM` scheduling software, in interactive mode. Typically startups to get to a place to run EXP will look something like

```
srun -n1 --pty $SHELL # start a 1-node interactive job
export LD_LIBRARY_PATH=/home/mpete0_umass_edu/lib:$LD_LIBRARY_PATH # set the library path on the
which exp # /home/mpete0_umass_edu/bin/exp
ldd /home/mpete0_umass_edu/bin/exp # validate the libraries (check that none are missing!)
```

[1] Message Passing Interface, a protocol by which different distributed computers communicate

[2] For completeness, some EXP routines will run without mpirun, and all can be run with mpirun. It is generally recommended to take advantage of the parallel processing capability, and use mpirun.

When EXP is first called, you will see a stanza listing the processors that have been engaged, as well as a banner listing the version header:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% This is EXP 7.6994 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Repository URL | https://mdweinberg@bitbucket.org/mdweinberg/exp.git    %
%%%%% Current branch | devel                                                 %
%%%%% Current commit | f4090892c7b638112c1163bf57c0157cc67d6e72              %
%%%%% Compile time    | 2022-01-27 09:31:25 UTC                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

To feed an input file to EXP , the standard usage is something like `mpirun -np 12 exp config.yml` where the input file is in .yml format – discussed in more detail below. For now, we will consider a general global stanza. Users will always need to change specific lines in a `.yml` input file. These are found at the very beginning of the file, in the 'Global' stanza.

```
runtag : the name of your run: all files are stamped with this (no spaces!).
outdir : /path/to/output/files
nsteps: how many large timesteps to integrate (integer)
dtime : the large timestep (float). See note.
multistep : how many time subdivisions to allow. See note.
dynFrac{A,V,P}: multistep time algorithm prefactors. See note.
ldlibdir : /path/to/lib/user -- See note.
VERBOSE  : how chatty the code is
```

Notes about the above parameters:

1. `dtime` is the large(st) timestep any particle will ever take. Typically this is between 0.001 and 0.01 virial units (see below). This timestep also sets the index for output routines (see below).
2. The smallest allowed step will be $2^{multistep}$ times smaller than `dtime`. For example, if `dtime=0.01` and `multistep=3`, the smallest timestep will be $0.01/8 = 0.00125$. Typically, multistep is in the 4-6 range. Opening more levels will add computational time.
3. The prefactors, which set the multistep level for particles, should be of the order how many steps you wish to take per orbital period. We usually shoot for about 100 steps per orbital period, so a first estimate[3] of the `dynfrac` parameters should be $\simeq 0.01$.
4. `ldlibdir` is the path where the `user` modules are installed. This is not needed to make EXP run, but if you wish to use a user module (see below), you will need this directory to be correct. On startup, EXP will search this directory for libraries and print a list of the available routines.
5. `VERBOSE` sets how much to report. `VERBOSE=0` will run the code in 'silent' mode. `VERBOSE=4` will return timing information as the code is running, and is a typical choice. Higher values of `VERBOSE` will return deeper debug information, and is usually not needed.

When should I use `sphereSL/cylinder/direct/noforce/other`? EXP is unique in the variety of force evaluation methods offered for N-body interaction, and crucially, the ability to mix and match different force methods. Choosing the correct force method for your problem is usually straightforward, by following this cheatsheet:

1. `direct` computes softened direct interactions between all particles, and should be used for small numbers of bodies, and/or investigations that require physical effects. Different softening kernels are available, which we will not discuss here. We recommend using the spline kernel (set `type:Spline`).
2. `sphereSL` uses Sturm-Liouville solutions to the Poisson equation to represent radial basis elements, and spherical harmonics for angular basis elements. This should be used for roughly spherical components where one wishes to include self gravity.
3. `cylinder` starts with a high-order Sturm-Liouville basis and creates empirical orthogonal functions that best match the target density distribution[4]. This should be used for flattened objects, like discs.
4. `noforce` applies no force, and should be used if one is running pure test particles[5]. This will almost always be used with some sort of external imposed potential, typically through a `user` module.

[3] Optimising runtime by adjusting multistep and timesteps is an advanced topic that will be covered in other tutorials as needed.
[4] If you look in the code, this will be called `EmpCylSL`.
[5] Particles which don't source their own gravity, but rather 'test' the potential.

There are other force evaluators too (purpose-built Hernquist and Clutton-Brock bases, 1d slab models, 2d disc bases), but the use of these is both advanced and atypical.

Integrating binary stars. We'll start with the simplest possible N-body integration: a binary with only two components to check for libraries and validate the code. This example uses a direct-summation force, so it does not exercise anything more than whether the code machinery will run, and serve as a first introduction to the input files. In the `NBodyTutorial/Binary/` folder, you'll find everything you need to run this example.

First, using the provided `binary.yml`, inspect the file using a text editor. You will see the Global stanza as described above, followed by a 'Component' stanza, followed by an 'Output' stanza (and two more empty stanzas: 'External' and 'Interaction', discussed below). After editing the output folder in the Global stanza, you are ready run your first EXP N-body simulation! To run interactively, simply type

```
mpirun -np 1 exp binary.yml # do a few interactive steps of the binary: always need mpirun!
```

You can break the evolution after 30 seconds or so by typing `ctrl+C`, or the whole simulation takes about a minute. If you now look in the folder (`ls -lah`), you will see new files! I now have

```
OUTLOG.run0
ORBTRACE.run0
run0.levels
config.run0.yml
current.processor.rates.run0
```

We've asked EXP to print the general log file (`OUTLOG`), a trace of the orbits (`ORBTRACE`), the multistep level diagnostics (`run0.levels`). EXP will also print the configuration file `config` to record the inputs, and a diagnostic file recording the speed of different processors (only useful on heterogenous systems, you shouldn't have to worry about this). We'll look at the majority of the diagnostics a bit later, but if you wanted to read the `ORBTRACE` file, it's a straightforward ascii file that can be read and plotted in Python [6]:

```
import numpy as np
import matplotlib.pyplot as plt

def read_orbtrace(filename):
    """definition to read EXP 'ORBTRACE'-style files
    One large dictionary is returned, with sub-dictionaries.
    Each orbit is returned as a sub-dictionary, with positions,
      velocities, and accelerations.
    """
    # each orbit has various quantities printed:
    names = ['x','y','z','u','v','w','ax','ay','az','lev']
    # open the file once to see the structure
    A = np.genfromtxt(filename,skip_header=100)
    # how many entries are there in the first row?
    columns = A.shape[1]
    # compute the number of orbits
    n_orbits = int((columns-1)/10)
    # now go back and read in the data to create separate orbits
    A = np.genfromtxt(filename,skip_header=columns+1)
    Orbit = dict()
    for orbit in range(n_orbits):
        Orbit[orbit] = dict()
        for ikey,key in enumerate(names):
            Orbit[orbit][key] = A[:,10*orbit+ikey+1]
    return Orbit


O = read_orbtrace('ORBTRACE.run0')
plt.plot(O[0]['x'],O[0]['y'],color='black')
plt.plot(O[1]['x'],O[1]['y'],color='red')
```

The resulting figure is shown in Figure 1. We've successfully run a first integration using the EXP machinery! We'll come back for the other output files shortly.

[6] Also available as a GitHub gist.

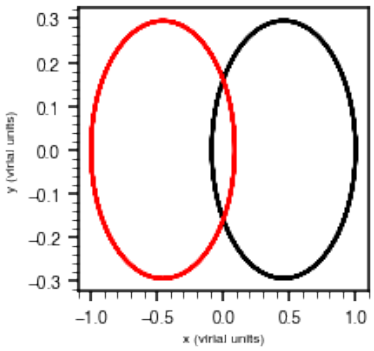

Figure 1: The result from the simple 'Binary' EXP test.

What are the implications of $G = 1$, and why should I use virial units? EXP does not allow for different values of G (in Poisson's equation[7], $\nabla^2\Phi = 4\pi G\rho$), but rather sets $G = 1$. EXP is best run in so-called *virial units*, as the code is hard-wired to only allow a gravitational constant $G = 1$ (this is different than GADGET-X !). Using $G = 1$ means that virial units are typically convenient numbers, but means that porting initial conditions and masses to run in EXP takes particular care (see equations below[8]).

In virial units, we set $R_{\rm vir} = M_{\rm vir} = V_{\rm vir} = T_{\rm vir} = 1$. Using these units[9], one can then back out various physical length, velocity, mass, and time scales ($R_{\rm physical}, M_{\rm physical}, V_{\rm physical}, T_{\rm physical}$, and even the physical force $F_{\rm physical}$) by simply multiplying the virial quantity by the physical quantity, provided three of the four are specified. Typically one will specify $R_{\rm vir}$, $M_{\rm vir}$, $V_{\rm vir}$ through the initial conditions, and let the time be set:

$$\begin{aligned} T_{\rm physical} &= \frac{R_{\rm physical}}{V_{\rm physical}} \\ M_{\rm physical} &= \frac{1}{G} R_{\rm physical} V^2_{\rm physical} \\ F_{\rm physical} &= \frac{V_{\rm physical}}{T_{\rm physical}} \end{aligned} \tag{1}$$

Note that here, $T_{\rm physical}$ is in kpc/km s^{-1}, but one can straightforwardly convert to Gyr, if desired – but using interchangably is typically fine[10]. For example, in the Milky Way, we will assume[11] that $R_{\rm physical} = 300$kpc, $V_{\rm physical} = 120$ km s^{-1}, $M_{\rm physical} = 10^{12}M_\odot$. This makes the physical time unit $T_{\rm physical} = 2.44$Gyr. One advantage of virial units is that they make the scaleability of simulations readily apparent. One can simply scale up or down the simulation to be equally applicable from (stellar) clusters to Milky-Way-sized halos, provided the right physical scalings.

Another advantage of operating in virial units for N-body work is that the timestep will perfectly correspond to some fraction of an orbit at the virial radius. This dictates the timestep recommendations above, set in the Global stanza.

How do I tell EXP about the configurations I want? In the binary example, we saw the different stanzas of the .yml input in action, but we didn't explore in any detail. Arguably the most important from the N-body perspective is the stanza that specifies the components. We'll look at some test cases here.

```
1  Components:
2    - name       : binary
3      parameters : {nlevel: 1, indexing: true}
4      bodyfile   : binary.bods
5      force      : {id : direct, type : Spline}
```

[7] Note also that spherical model tables for basis construction **do not** include 4π (which is applied throughout the code), but is rather just the second numerical derivative to match the Poisson equation, $\nabla^2\Phi = 4\pi G\rho$. One must take care when constructing tables – for next time!

[8] And use the value of G for astronomy purposes: $G = 4.3009125 \times 10^{-6}$(km s^{-1})^2 kpcM$_\odot{}^{-1}$.

[9] Recall the the underpinning for the units is simply

$$V_{\rm vir} = V(R_{\rm vir}) = \sqrt{\frac{GM_{\rm vir}}{R_{\rm vir}}} = 1.$$

[10] Further easing our translations, the conversion from s/Gyr to km/kpc is nearly unity: $3.154 \times 10^{16}/3.086 \times 10^{16} = 1.02203$.

[11] Using a mass model for the Milky Way, I set the virial radius and total mass, then backed out the velocity by taking the observed circular velocity at the solar circle (230km s^{-1}) and dividing by the virial velocity at that the solar circle.

The `parameters` list under the component name is a catchall that we will fill out more later. The two parameters we currently set are `nlevel`, which sets the frequency that multistep levels are reported (in units of `dtime` from the Global stanza), and `indexing`, which tells EXP to keep track of particle indices (typically not needed, but it is safest to keep this `true`).

The input location of points to be integrated is `bodyfile`. The format of the bodyfile is simple ascii, with each line of the file a simple list of

```
mass xposition yposition zposition xvelocity yvelocity zvelocity
```

An additional header line specifying the number of bodies, as well as the number of additional integer and float attributes (not needed for standard EXP usage, so simply set both to 0):

```
nbodies nintegerattributes nfloatattributes
```

Using the Binary example, if we inspect `binary.bods`, we see (the entire file!)

```
mpete0_umass_edu@login:/nas/astro-th/mpetersen/Binary$ more binary.bods
2 0 0
   1.0     1.0  0.0  0.0  0.0   0.2  0.0
   1.0    -1.0  0.0  0.0  0.0  -0.2  0.0
```

You should be able to translate the different elements of the file using the instructions above.

The force is specified with `id` call in the `force` entry, which will also take any and all additional parameters. In this case, we used `direct`, so the only force in the problem was each body in the binary influencing each other. We're now ready for a more complex example, which is our first foray into the guts of EXP technology.

Evolving an isolated halo. This test is located in `NBodyTutorial/SimpleHalo`. The test is a simple example of using an adaptive spherical basis, `sphereSL`. The model only has one component, a spherical halo. The target halo model is specified here in the ascii file SLGridSph.model.

```
- name       : dark halo
  parameters : {nlevel: 1, indexing: true}
  bodyfile   : halo.bods
  force :
    id : sphereSL
    parameters :
      modelname       : SLGridSph.model
      Lmax            : 6
      nmax            : 20
      numr            : 4000
      rmin            : 0.0001
      rmax            : 1.95
      rs              : 0.0667
      self_consistent : true
```

We see the familiar component `name`, `parameters`, and `bodyfile`. However, we're now seeing our first basis function parameters with a new force force option: the `sphereSL` force class. When using `sphereSL`, we need a spherical model file. In this case, the model file is called `SLGridSph.model`. We can inspect the file:

```
mpete0_umass_edu@login:/nas/astro-th/mpetersen/SimpleHalo$ more SLGridSph.model
! Scaling:  R=2  M=1.008  alpha=1  beta=3  rcore=0.015  rtrunc=15  wtrunc=4
! 1) = r   2) = rho   3) = M(r)   4) U(r)
      1000
  3.333333333333e-05  4.049325987474e+04  0.000000000000e+00 -1.597315671103e+01
```

(the file continues.) This is the general format for spherical model tables in EXP : four columns, one each of spherical radius, three-dimensional density, mass enclosed, and potential. There is also one additional required header row that specifies the number of entries in each column (here, 1000). The first two rows are c omments, with a leading '!'. More comment rows may be added. Construction of spherical models is covered in the next tutorial.

With this spherical model table, we can then choose the harmonic order `lmax` (we chose 6, which will give us all spherical harmonics up to $\ell = 6$ as the angular solution) and the number of radial functions per harmonic order `nmax` (we chose 20, which is typical for halo simulations). The additional parameters set the minimum and maximum radius (0.0001 and 1.95, respectively, in virial units!) and number of grid points (4000). Under the hood, EXP uses a rescaling of the radial interval to the interval [-1, 1] with a scale factor $r_s = 0.0667$ using the scaling function $x = (r/r_s - 1)/(r/r_s + 1)$. Therefore, r_s should be chosen to be roughly the scale parameter

of the model. The parameter `self_consistent` tells the code to recompute the basis at every step. If false, the gravitational field is fixed after the first evaluation[12].

This simulation will best run with slightly more computational power, so we will request more nodes from `SLURM`, and again run the simulation in interactive mode:

```
srun -n12 --pty $SHELL
mpirun -np 12 exp livesphere.yml
```

Immediately after starting the simulation, a new file will appear in the directory: `SLGridSph.cache.run0`. This file holds the functions for the basis, which may be read back in by EXP (more on this later).

Primary outputs from EXP (How do I read the log files? How do I read the phase-space files? How do I read coefficients?) In the 'Output' stanza, we've seen a few different entries so far. If we look at the SimpleHalo run, we see

```
Output:
  - id : outlog
    parameters : {nint: 1}
  - id : outpsn
    parameters : {nint: 10}
  - id : outchkpt
    parameters : {nint: 50}
```

The parameter `nint` set the frequency that outputs are created, in units of `dtime` (from the 'Global' stanza). After the simulation completes (about three minutes), we also see new files in the directory, in addition to the same files we saw in the binary example run (abridged list):

```
OUTLOG.run0
OUT.run0.00000
OUT.run0.chkpt
```

Each file is generated by a different output routine. The most common output routines are

1. `orbtrace`, a helper to record a subset of orbits. We have already seen this in action above. Useful for very small components, or to gain extra diagnostic information about a handful of (say, 100) orbits.

2. `outlog`, which prints the general diagnostics for each component. See immediately below for an example.

3. `outpsn`, the most common phase-space writer, in 'Phase-Space Protocol' format. These files start with `OUT.` and end in a 5-digit number. This method is best used for large files, as it will write in binary format to consolidate space. The downside is that it requires a custom reader. EXP will handle these files internally, and a `Python` reader is available in the source branch here[13].

4. `outchkpt`, the checkpoint writer, used for restarting long simulations. These files start with `OUT.` and end in `.chkpt`. Typically these can have very infrequent saves, but are the best file to restart a run[14].

5. `outcoef`, the coefficient writer. EXP will handle the format flexibly. `Python` support is offered by `OutCoef`, in the EXPTOOL tree, located here (to be consumed into EXP). See below in the disc+halo example. The file formats differ between a `sphereSL` and `cylinder` component.

6. `outpsq`, a parallel-write version of the phase space files. These files start with `SPL.` and end in 5-digit numbers. Best used when running in GPU mode when outputting phase spaces frequently, where write times can be a large part of the runtime. These files can be read in using the same routine as `outpsn` in the source tree.

7. `outascii`, an ascii version of the phase-space writer. Best used for small numbers of particles, only.

Additionally, many tools are already available for N-body analysis. EXP provides a suite of tools for N-body analysis in the Analysis folder. EXPTOOL offers comprehensive `Python` analysis support, tailored to match EXP . In some cases, the needed code has been merged into the EXP source tree; in other cases there is still ongoing development. In particular, reading phase-space files can be straightforwardly done in `Python` using the `psp_io` routine, and specifying the name of the component you want to read:

```
O = psp_io.Input('OUT.run0.00000','dark_halo')
```

[12]This is how one can 'freeze' components, such that they act as test particles, but you have all the advantages of the initial basis flexibility.

[13]Or the cutting-edge version as part of EXPTOOL , called `particle.py`, located here.

[14]Using the 'Global' stanza parameter `infile : OUT.run0.chkpt`, for example.

This will create a class which holds the phase space data as a `.data` dictionary, with keys holding the position and velocity. For example, to make a simple scatter plot of the $x - y$ plane using the above output,

```
plt.figure(figsize=(4,4))
plt.scatter(O.data['x'],O.data['y'],color='black',s=2.)
```

Another good place to start for analysing any run is with the `OUTLOG` file, which can be straightforwardly read using the below definition[15]:

```
import numpy as np
import matplotlib.pyplot as plt

def read_outlog(filename):
    """definition to read EXP 'OUTLOG'-style files
    One large dictionary is returned, with sub-dictionaries.
    Each component is returned as a sub-dictionary, with positions,
      velocities, and energy conservation.
    An additional sub-dictionary, with global quantities, is also included.
    """
    # each orbit has various quantities printed:
    global_columns = ['time','mass','bodies','x','y','z','u','v','w',\
        'Lx','Ly','Lz','KE','PE','VC','E','2T/VC','clock','used']
    component_columns = ['mass','bodies','x','y','z','u','v','w',\
        'Lx','Ly','Lz','Cx','Cy','Cz','KE','PE','VC','E','2T/VC','used']
    # open the file once to see the structure
    A = np.genfromtxt(filename,skip_header=6,delimiter='|')
    # how many entries are there in the first row?
    columns = A.shape[1]
    # compute the number of components
    n_components = int((columns-19)/20)
    # go back and prepare to get the component names
    B = np.genfromtxt(filename,skip_header=2,comments="@",delimiter='|',dtype='S20',\
        invalid_raise=False)
    Log = dict()
    # first read in the global component
    Log['global'] = dict()
    for ikey,key in enumerate(global_columns):
            Log['global'][key] = A[:,ikey]
    for component in range(n_components):
        # get the correct component name
        st = B[0][20*component+19].split()
        if len(st)>1:
            st = [s.decode() for s in st[0:-1]]
            component_name = '_'.join(st)
        else:
            component_name = st[0].decode()
        Log[component_name] = dict()
        for ikey,key in enumerate(component_columns):
            Log[component_name][key] = A[:,20*component+ikey+19]
    return Log


O = read_outlog('OUTLOG.run0')
print(O.keys())
plt.figure(figsize=(4,3))
plt.plot(O['global']['time'],O['dark_halo']['KE'],color='black')
plt.plot(O['global']['time'],O['dark_halo']['PE'],color='red')
plt.xlabel('time_(virial_units)')
plt.ylabel('energy_(virial_units)')
```

The resulting plot is shown in Figure 2. The kinetic and potential energy show show drift, but the sum is (largely) conserved. We can also use an additional EXP helper routine to inspect the headers of `PSP` files, using the command `pspinfo OUT.run0.00000`. We're now ready to mix together multiple components, which we will do as a stellar disc and dark halo example.

[15] Also available as a GitHub gist.

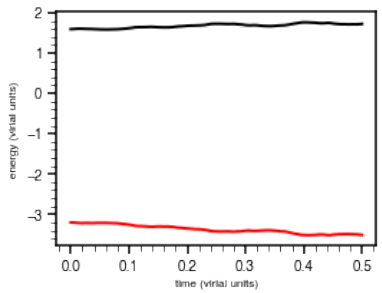

Figure 2: Kinetic and potential energy from the 'SimpleHalo' test.

A first disc+halo integration. This test is located in `NBodyTutorial/DiskHalo`. The test is a simple example of using an adaptive cylindrical basis, `cylinder`, combined with the same spherical basis we used in the previous example. If we inspect the `.yml` file, we now see an additional entry in the 'Components' stanza:

```
  - name       : star disk
    parameters : {nlevel: 1}
    bodyfile   : disk.bods
    force :
      id : cylinder
      parameters :
        acyl           : 0.01
        hcyl           : 0.001
        mmax           : 6
        ncylorder      : 8
        lmax           : 32
        nmax           : 32
        ncylnx         : 256
        ncylny         : 128
        logr           : false
        self_consistent: true
```

Crucially, we now need a basis for the stellar disk. For the purposes of this tutorial, we have provided a basis cache file. We will cover generating cylindrical bases in the next tutorial session. The provided basis cache file was produced in the `x86_64` architecture, with the parameters specified above. This cache file contains a pre-made set of disk basis functions. It will be read from the 'outdir' directory in the Global stanza. If you'd like to have a different outdir from this current working dir, please move `.eof.cache.run0` to your 'outdir'.

In general, EXP will default to remake the cylindrical basis if it finds that the cachefile does not equal the specified parameters. However, this is computationally expensive. Therefore, particular care should be taken if one is passing a cachefile to EXP in order to make sure it is accepted. The other specified parameters must match the cachefile, or EXP will attempt to remake the basis. When running EXP and inspecting the outputs, you can see if the cached basis was accepted by looking for

```
EmpCylSL::cache_grid: file read successfully
EmpCylSL::read_cache: table forwarded to all processes
Cylinder parameters:  nmax=32 lmax=32 mmax=6 mlim=-1 ncylorder=8
  ncylodd=-1 rcylmin=0.001 rcylmax=20 acyl=0.01 hcyl=0.001 expcond=1
  pcavar=0 pcaeof=0 nvtk=1 npca=50 npca0=0 pcadiag=0
  eof_file=.eof.cache.run0 override=false logarithmic=false vflag=0
```

where the last lines report the full set of parameters used to generate the basis. In this case, we see that we are using harmonic order $m = 6$ (similar to the halo), $n = 8$ (fewer than the halo). The basis was designed to approximate an exponential disc with a scale height to scale length ratio of 1:10. Other parameters will be covered in later tutorials.

Two more pieces of crucial information to note in the `.yml` file. First, there are now entries in the 'Interaction' stanza. Whenever a run includes multiple components

```
Interaction:
  dark halo : star disk
  star disk : dark halo
```

Second, we ask EXP to record the component coefficients using `outcoef`. Unlike the other output routines, here we must specifically request the output for each component.

```
  - id : outcoef
```

```
    parameters : {nint: 1, nintsub: 10, name: star disk, filename: outcoef.star.run0 }
  - id : outcoef
    parameters : {nint: 1, nintsub: 10, name: dark halo, filename: outcoef.dark.run0 }
```

We have introduced another option, `nintsub`, which records coefficients at the requested `multistep` level. Here we are requesting 10, so that the coefficients are printed at every substep.

This simulation is small, and will also run (fairly) quickly as an interactive job (try to replicate from the commands above!). When complete, we will see two new files in the directory:

```
outcoef.dark.run0
outcoef.star.run0
```

Which hold the coefficient time series. Next, we will cover how to do a simple analysis of the coefficients.

Coefficient analysis In all likelihood, you are here for the EXP coefficients in one form or another. To read the outputs from the coefficient files, one may either interface with various EXP routines (including binary file structure if one wishes to read into another program, such as MSSA), or use a `Python` interface that will read the coefficients directly. The `Python` version is offered as a standalone tool.

```
from exptool.io import outcoef

O = outcoef.OutCoef('outcoef.dark.run0')
plt.figure(figsize=(4,3))
for n in range(0,5):
    plt.plot(O.T,O.coefs[:,0,n],color=cm.viridis(n/4.))

plt.xlabel('time (virial)')
plt.ylabel('amplitude')

# don't plot the lowest-order function,
# so we can see some dynamic range
plt.figure(figsize=(4,3))
for n in range(1,5):
    plt.plot(O.T,O.coefs[:,0,n],color=cm.viridis(n/4.))

plt.xlabel('time (virial)')
plt.ylabel('amplitude')
```

The same routine will also read cylindrical files, as shown in this example.

```
O = outcoef.OutCoef('outcoef.star.run0')
plt.figure(figsize=(4,3))
for n in range(0,5):
    plt.plot(O.T,O.coefs[:,0,0,n],color=cm.viridis(n/4.))

plt.xlabel('time (virial)')
plt.ylabel('amplitude')
```

The figures from the two blocks of code are shown in Figure 6.

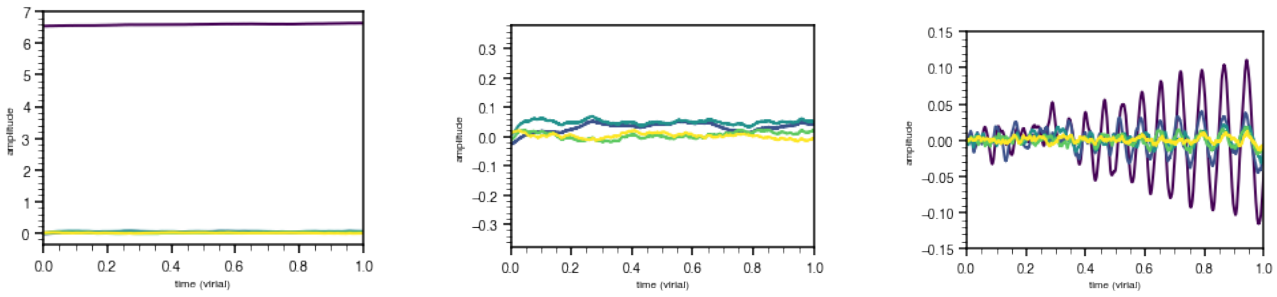

Figure 3: Three plots of the coefficients in the 'DiskHalo' test.

Special circumstances, User modules (How do I set up a particular specialised test in EXP?) The easiest way to create extendable features in EXP is through user modules. For example, we can create a

static potential background in which to run additional models through a User extension. When EXP starts up, you will see a list of the available routines:

```
ExternalCollection:
-------------------------------------------------------------------------------
Loaded user libraries <libdisk.so libebar.so libhalo.so liblogpot.so libmndisk.so libuser.so>
-------------------------------------------------------------------------------
Available user routines are <userdisk userebar userhalo userlogp usermndisk usersat>
-------------------------------------------------------------------------------
```

(There will likely be many more!) If you *do not* see a list of user modules when booting EXP , you likely need to check your `ldlibdir` parameter in the Global stanza. Some user routines are generally available, but the release plan for EXP will include a small number, with the expectation being that users will write their own. The typical call to a user module will look like:

```
External:
  - id : userlogp
    parameters : {R : 0.1, b : 1., c: 1., v2 : 0.9}
```

When user routines are initialised, you will typically get a banner that describes the routine and the rough effect it will have. Let's try a couple examples, building on what we tried before. First, let's re-run the binary elements in an imposed logarithmic potential using the parameters above.

The binary elements, in a fixed logarithmic potential. This test is in the `Binary/` directory, but uses a different input file (`logpot.yml`). Our first exposure to a user routine will be running the binary elements with no self gravity, but rather in an imposed logarithmic potential. The commands to run the simulation are identical, and we've only changed a handful of things in he `.yml` file. In particular, this is our first use of `noforce`. The resulting figure is shown in Figure 4. We've used our first imposed potential and `user` routine!

If we take a peek under the hood at the code that enabled the integration here, it's really not too bad at all – one can perhaps start to see how the code framework would be extendable. You *do not* have to use `noforce` when using an external potential: you could still allow the binary elements to feel each other's gravity by enabling `direct`. If one were to run a cluster of stars in an imposed background galaxy potential (for example), one would still want the self-gravity of the cluster to be on, for instance.

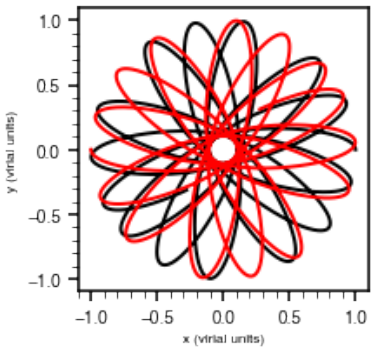

Figure 4: The result from the simple 'Logarithmic Potential' EXP test.

Perturbing the galactic disc with a satellite. Another `user` module one may wish to use is `usersat`, which introduces a softened point mass satellite to the simulation. All components will feel the force from the satellite. The main code for `usersat` is found here, though users will (currently) need to follow through the code to find all the options.

To set the satellite parameters, one must specify the mass (`mass`) and the Plummer softening (`core`). Additionally, one sets the key times in the simulation: `ton/toff` sets the centre of the error function turn-on/off, `delta` sets the width of the error function, and `toffset` slides the orbits from the fiducial time (based on trajectory types, below, the fiducial time is usually when pericentre is reached). We also set `orbit=true` to print the trajectory of the orbit. One may also set `shadow=true` to set an exact replica of the satellite that cancels out the odd-order components – usually used for testing bar formation.

There are four options for `trajtype`:

1. `circ`, a forced circular orbit in the potential of the simulation. (`trajtype=0`)
2. `bound`, a self-consistent orbit in the potential of the simulation (`trajtype=1`, `SatelliteOrbit.cc`)

3. `unbound`, an unbound, self-consistent orbit in the potential of the simulation (`trajtype=2`, `UnboundOrbit.cc`)

4. `linear`, a specified linear flyby (`trajtype=3`, `LinearOrbit.cc`)

Each trajectory has several options, passed through a `config` stanza. For example, a full `usersat` call setting a satellite on a linear orbit flying by the galaxy might look something like:

```
External:
  - id : usersat
    parameters :
      orbit     : true
      ton       : 0.2
      core      : 0.03
      mass      : 0.05
      delta     : 0.05
      toffset   : 0.2
      shadow    : true
      trajtype  : 3
      config:
        X0      : 0.2
        Y0      : -0.5
        Z0      : 0.
        Vsat    : 1.
```

Running the simulation will generate `UserSat.run1.1`, which is the trajectory file for the satellite. Note that for a linear orbit, the parameters for the trajectory are the initial positions $\{X, Y, Z\}$, and Y velocity (`Vsat`). One may also rotate the orbit of the satellite using three Euler angles `THETA`, `PSI`, and `PHIP` to re-orient the initial plane.

The same diagnostic plots from Figure 6 are shown in Figure 5, except for the satellite case. The left-most image has been replaced with the satellite's radius.

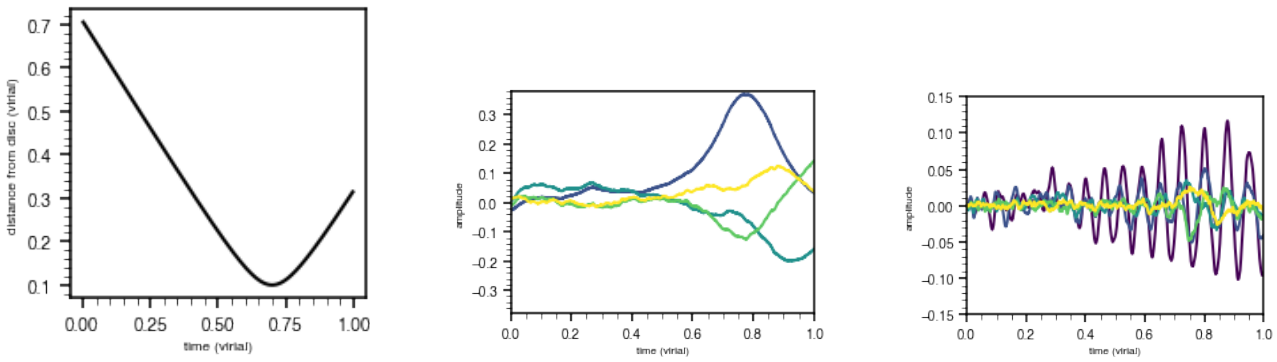

Figure 5: Three plots from the 'Perturbed DiskHalo' test.

Danger zones (What shouldn't I try to do with EXP, yet [at least not without a diagnostic plan]?) The primary limitation for the basis function pieces of EXP lies in the perturbative nature of the basis representation. If the mass distribution of a particular component gets 'too far' away from the initial distribution, the theory behind basis function expansions will no longer work. The 'feel' of when a basis works or doesn't work has been honed by practitioners, but needs to be studied further. **In this session, we have not designed our own bases: this is a goal of the next session.** Without designed bases, we are restricted in what problems we can currently study – any components that do not match the density distribution of a component we have already used (e.g. the halos and the particular exponential disc we used) are currently inaccessible.

Centering A note about centering[16]. The default recommendation is that one *not* change the expansion centre from the inertial centre. This default will work for a wide variety of problems that are angular-momentum conserving, including secular disc evolution, small halo perturbations, bar formation, etc. This will *not* work in the case of something like a large satellite flyby, where the response of the host galaxy is important. In such a case, one must engage the centering routines, which are specified per component, in the component stanza, under 'parameters'. For example:

[16]The topic of centering in basis function expansions is likely the topic of several papers, so we will be *extremely* brief here.

```
- name       : dark halo
  parameters :
    nlevel    : 1
    indexing  : true
    EJ        : 2
    EJdryrun  : false
    nEJwant   : 1000
    nEJkeep   : 0
    EJdiag    : true
```

This set of configurations tells EXP to engage the `Orient` code in configuration `EJ : 2`, which tracks the potential centre, *does* apply the centre (`EJdryrun : false`), uses the 1000 particles nearest to the potential centre (`nEJwant`), does not apply any smoothing (`nEJkeep`), and prints the diagnostics (`EJdiag : true`). This will now output a file tagged with `.orient.`. **Please be careful with centering: the algorithm can fail in certain circumstances. Always check the diagnostics for anything 'weird'!**

The `.orient.` files are your diagnostic friends. There is GitHub gist available to read the output files here. There are several types of columns, which dump the diagnostics from the code (see `Orient.cc`):

1. The axis orientation computed from regression (if `nEJkeep=0`, this will be the same as the particle algorithm)
2. The axis orientation computed from the particle algorithm
3. The centre computed from regression (if `nEJkeep=0`, this will be the same as the particle algorithm)
4. The centre computed from velocity integration (analytic)
5. The centre computed from the particle algorithm

General N-body warnings There are also generic N-body warnings to try not to run afoul of:

1. DO NOT just blindly interpret the outputs from an EXP (or *any* N-body) simulation. Simulations are complex machines with hundreds of input choices, some of which have been made implicitly. Always check the diagnostics provided for anything that 'looks funny': rapidly changing or discontinuous quantities and zeros are two primary warning signs.
2. Don't trust something that seems too good to be true. I'm thinking of timesteps, which can easily be too large, or trusting results on scales that simulations are simply not senstive to (below the 'resolution'). An interesting experiment with EXP is breaking the timesteps by tinkering with the multistep settings. As you change `multistep` and the `dynfrac` values, you can check how the timesteps changes by tracking the `.levels` file.

GPU instructions. One of the other highly attractive features of EXP is its easy portability to Graphical Processing Units (GPUs). Any of the runs above could be replicated using the GPU side of EXP , but as accessing GPUs tends to be cluster-specific, we will not test GPU runs during the tutorial. However, if you know how to access GPUs on your own machine, it is trivial to convert an EXP `.yml` script to run on GPUs. Simply add the following line to the Global stanza:

```
use_cuda : true
```

The rest is figuring out how to properly harness the hardware. The typical call to get an interactive GPU job will look something like

```
srun -n8 -p astro_th_gpu --mem=24G --gres=gpu:2 --pty $SHELL
```

where `-p` specifies the name of the queue (partition) that hosts the GPUs, and `gres=gpu:2` requests GPU hardware, and specifically two of them.

2 BFE expansion

One application of the machinery in EXP is the BFE expansion of 'arbitrary' distributions. EXP makes the process relatively painless, at the expense of pre-planning and models that must be carefully designed to take advantage of the high-fidelity, low-noise advantages of BFE. This tutorial covers how to design your own basis. The main reference for the theory behind the bases is 'EXP : N-body integration using basis function expansions' by Petersen, Weinberg, Katz (2022).

The first order of business is to decide which EXP expansion method works for you. There are two different basis methods inside EXP :

1. `sphereSL` uses Sturm-Liouville solutions to the Poisson equation to represent radial basis elements, and spherical harmonics for angular basis elements. This should be used for roughly spherical components where one wishes to include self gravity.

2. `cylinder` starts with a high-order Sturm-Liouville basis and creates empirical orthogonal functions that best match the target density distribution[17]. This should be used for flattened objects, like discs.

We'll cover both in this tutorial, focusing first on spherical distributions before moving to disc distributions. Whichever basis you use, the units must be the same between the basis and particles: that is, if you have particles in virial units, the basis must be in virial units, but if you have particles in physical units, the basis must be in physical units.

Goals for this session:

1. Understand what's in a spherical and cylindrical basis.

2. Build a simple, arbitrary spherical basis.

3. Build an exponential disc basis (and plan for extensions).

4. Use `haloprof` or `diskprof` to output coefficients.

We'll use the same tutorials as in the first session. We'll assume in general that you have run the simulations to generate data, but the lessons here may be straightforwardly extended. We'll again mostly work in the EXP directory `utils/Analysis`, which should be the primary codebase resource for extensions.

2.1 Spherical models

Construction of a spherical basis is straightforward, but the selection of the best radial profile is not always simple. For a simple spherical distribution, the solution will be the monopole, or simple density distribution with radius. For the table, we need the density profile as a function of spherical radius r ($\rho(r)$), the mass enclosed profile as a function of radius ($M_{\rm enc}(r)$), and the potential as a function of radius ($\Psi(r)$).

In the case where the functional form of the density profile $\rho(r)$ is known, with spherical radius r, constructing the spherical model table is straightforward. These steps are implemented in `makemodel`[18], but we list them here for completeness. The general form for the potential is (see Binney & Tremaine chapter 2)

$$\Psi(r) = -4\pi G \left[\frac{1}{r} \int_0^r dr' r'^2 \rho(r') + \int_r^\infty dr' r' \rho(r') \right]. \tag{2}$$

We can compute the potential numerically to high precision, simply from the density. Compute the mass enclosed and the gravitational potential energy through recursion (starting from $n = 1$, with $M_0 = 0$, $W_0 = 0$, $r_0 = \text{rmin}$):

$$\begin{aligned} M_n &= M_{n-1} + 2\pi \left(r_{n-1}^2 \rho_{n-1} + r_n^2 \rho_n \right) (r_n - r_{n-1}) \\ W_n &= W_{n-1} + 2\pi \left(r_{n-1} \rho_{n-1} + r_n \rho_n \right) (r_n - r_{n-1}) \end{aligned} \tag{3}$$

where the r values are chosen to cover the range of the model. The potential is then computed as

$$\Psi(r) = \frac{-M}{r} - (W(\infty) - W(r)) \tag{4}$$

where we can see the correspondence with equation 2 by term, with $4\pi G = 1$.

[17] If you look in the code, this will be called `EmpCylSL`.
[18] Available here.

The virial units of the realised model may be scaled conveniently to a target total mass $M_{\rm target}$ and radius $r_{\rm target}$ by defining

$$\begin{aligned}\beta &= \frac{M_{\rm target}}{M(\infty)}\frac{r(\infty)}{r_{\rm target}} \\ \gamma &= \sqrt{\frac{M(\infty)r(\infty)}{M_{\rm target}r_{\rm target}}}\frac{r(\infty)}{r_{\rm target}}\end{aligned} \tag{5}$$

and then scaling the relevant quantities through unit analysis:

$$\begin{aligned}r_{\rm scaled} &= \beta^{-1/4}\gamma^{-1/2}r_{\rm virial} \\ \rho_{\rm scaled} &= \beta^{3/2}\gamma\rho_{\rm virial} \\ M_{\rm scaled} &= \beta^{3/4}\gamma^{-1/2}M_{\rm virial} \\ \Psi_{\rm scaled} &= \beta\Psi_{\rm virial}\end{aligned}$$

One advantage of the scaling is that one need not know ρ_0, the central density, a priori.

I have a known spherical analytic density profile. How do I build a basis? If you are designing a model from scratch, this is almost certainly the situation you will find yourself in.

The code inside of EXP to build a spherical basis is remarkably straightforward. Consider the following lines from `haloprof` and `SphereSL.H`:

```
1  //the call from utils/Analysis/haloprof
2  SphereSL ortho(halo, LMAX, NMAX, 1, rscale, true, NPART);
3  // the call as defined in utils/Analysis/SphereSL.H
4  SphereSL(std::shared_ptr<SphericalModelTable> mod, int LMAX, int NMAX,
5              int CMAP=0, double SCALE=0.067, bool COVAR=false, int NPART=0);
6  // see also include/SLGridMP2.H
```

That's all you need to build a spherical basis inside EXP ! Obviously there are some subtleties, but we already recognise some of the parameters: `LMAX` is the harmonic order and `NMAX` is the radial order. We will learn how to build the `SphericalModelTable` below. The other parameters control scalings and are generally not needed unless using EXP in some (currently) non-standard way.

To choose a minimum (`RMIN`) and maximum radius (`RMAX`) for the spherical expansion, one should consider the smallest radius that will be probed by particles and the largest radius that has a well-defined density. These numbers both tend to be set by the sampling of the density distribution (number of particles).

Example: a case where the density profile is known. Making a Hernquist basis table for EXP . Consider the generalised two-power distribution for a halo:

$$\rho(r) \propto \left(\frac{r}{a}\right)^{-\alpha}\left(1+\frac{r}{a}\right)^{-(\beta-\alpha)}. \tag{6}$$

The Hernquist profile is given by $\alpha = 1$, $\beta = 4$. We can construct the model file by

```
1  def twopower_density(r,a,alpha,beta):
2      """a twopower density profile"""
3      ra = r/a
4      return (ra**-alpha)*(1+ra)**(-beta+alpha)
5
6  rvals = np.linspace(1.,300.,1000)
7  dens  = twopower_density(rvals,35.,1,4)
8  plt.figure(figsize=(4,4))
9  plt.plot(np.log10(rvals),np.log10(dens))
10 plt.xlabel('log radius (kpc)')
11 plt.ylabel('log density')
```

We can check our numerically-computed potential against the analytic potential for a Hernquist model, given by (cf. BT08 eq. 2.67)

$$\Psi_{\rm Hernquist} = -G\int_r^\infty dr\frac{M(r)}{r^2} = -4\pi G\rho_0 a^2\frac{1}{2\left(1+\frac{r}{a}\right)} \tag{7}$$

Additionally, getting the overall normalisation (i.e. mass) is not crucial as the coefficients can take care of the overall normalisation.

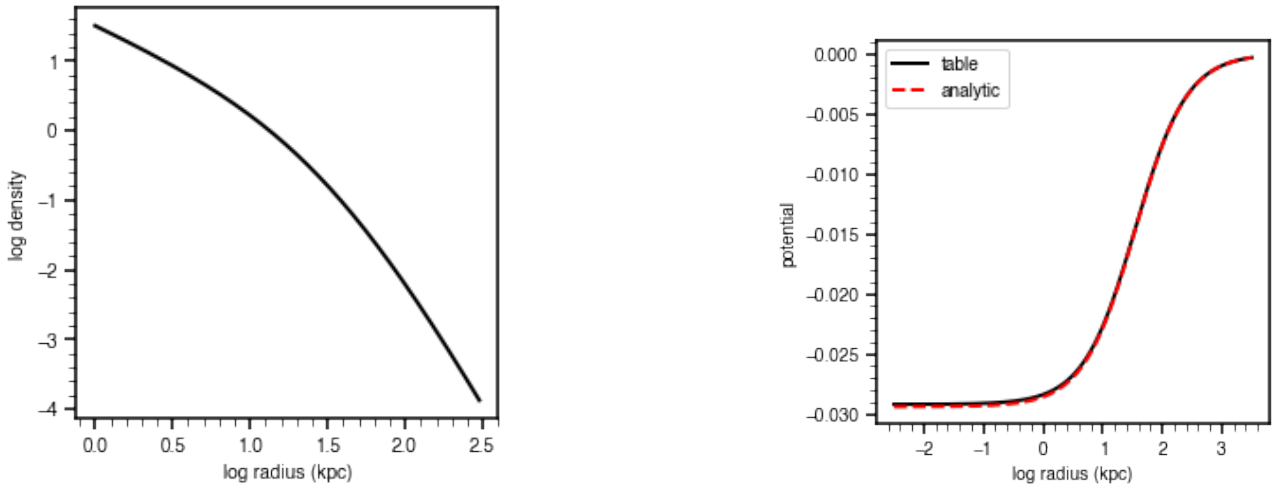

Figure 6: Two plots validating the model table construction.

I have a spherical-ish distribution of particles. How do I build a basis? The procedure is very similar to the process above: you must define a spherical density distribution, and then generate a spherical model file. Typically we'd recommend one of two options:

1. If the number of particles is large enough (e.g. over a million), construct an empirical density profile. **The profile must be sufficiently smooth (by-eye inspection) as the basis uses these values directly.**
2. Define a functional form that is nearly the density profile of the spherical distribution and design the basis from that.

Either option should be straightforward to implement; we can supply code to do either, if helpful.

I have a spherical-ish distribution of particles and a basis. How do I compute coefficients? As a test, let's see if we can recreate the same coefficients for the satellite-affected `DiskHalo` run from the first tutorial.

```
mpirun -np 16 haloprof --LMAX=4 --NMAX=16 --MODFILE=SLGridSph.model
  --outdir=/nas/astro-th/mpetersen/NBodyTutorial/DiskHalo --coefs=coefs
  --dir=/nas/astro-th/mpetersen/NBodyTutorial/DiskHalo --beg=0 --end=100 --prefix=OUT
  --filetype=PSPout --RMAX=1 --RSCALE=0.067 --CONLY -v --runtag=run1 --compname="dark halo"
```

We can now compare against the original coefficients from the run:

```
from exptool.io import outcoef
C = outcoef.OutCoef('/Users/mpetersen/Downloads/coefs.coefs')
C2 = outcoef.OutCoef('/Users/mpetersen/Downloads/outcoef.dark.run1')

plt.figure(figsize=(4,4))

plt.plot(C2.T,-C2.coefs[:,0,1],label='original',color='black')
plt.plot(C.T,C.coefs[:,0,1],label='reconstructed',color='red',linestyle='dashed')

plt.legend()
plt.xlabel('time')
plt.ylabel('amplitude')
```

2.2 Cylindrical models

Cylindrical models are modestly less well-studied than spherical models, for the sole reason that we are the only ones using them presently. One should generally have have some sense of the underlying structure in the disc component in order to realise a disc basis, but one can also make a basis directly from particles. In particular, one can either

1. Start from a parameterised function for the disc basis (primary mode), or

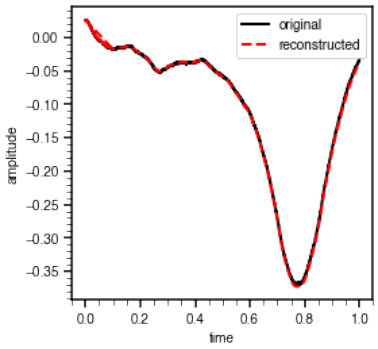

Figure 7: Comparison of the spherical coefficients from the simulation, and reconstructed.

2. Feed in a particle distribution and let the algorithms do the work, if one has a distribution of particles already (most relevant for cosmological simulations).

We'll cover both cases.

How do I design a basis that will work for an analytic exponential disc? The most well-studied `cylinder` basis case is that of the exponential disc. One can extend the models to additional density distributions, but these are not currently as well-studied (though they would be a fun project!). The program `cylcache` is specifically designed for the purpose of building an analytic disc basis.

EXP currently supports the following disc geometries as analytic densities:

1. constant cylindrical slab
2. Gaussian radial profile, constant vertical profile
3. Miyamoto-Nagai (MN)
4. Standard one-component exponential
5. Two-component exponential (sum of two exponentials with different a and z)

If all you care about is constructing a basis, you can simply run `gendisk` in the default mode and stop after the basis is constructed. We are currently working to reset default values. The most relevant settings (with typical values) are

```
MMAX = 6 # the number of azimuthal harmonics
NORDER = 12 # the number of radial basis elements per harmonic
ASCALE = 0.0141 # the scale length of the expansion
HSCALE = 0.002 # the scale height of the expansion
```

You will also need a halofile. You can simply pull one from a different tutorial directory.

To run,

```
srun -n64 --exclusive --pty $SHELL
cd /nas/astro-th/mpetersen/NBodyTutorial/CylBasis
mpirun gendisk --conf=template.config # generate a default file
mpirun -np 64 gendisk --input=model.conf

mpirun -np 16 cylcache --ASCALE=0.01 --HSCALE=0.001 --LMAX2=32 --NMAX2=32
```

I have a disc-like distribution of particles. How do I flexibly build a basis? EXP can also build a `cylinder` basis from particles themselves and some simple assumptions about the density distribution. This is expensive, and has not yet been fully reviewed for force accuracy. Within constrained situations, however, this can be a very workable option. The telltale signs to look for in basis construction will look something like:

```
EmpCylSL::cache_grid: error opening file
Cylinder: can not read EOF file </nas/astro-th/mpetersen/NBodyTutorial/DiskHalo/.eof.cache.run2>
Cylinder: will attempt to generate EOF file, this will take some time (e.g. hours) . . .
EmpCylSL::make_sl(): making SLGridSph with <Exponential> model
```

Example: building a cylindrical basis to run N-body If EXP doesn't find the cylinder cache, EXP will attempt to make a basis from scratch. You can see an example of this by saving `NBodyTutorial/DiskHalo/galaxy.yml` as `NBodyTutorial/DiskHalo/galaxyadapt.yml` and changing the disc component stanza to read:

```
- name       : star disk
  parameters : {nlevel: 1}
  bodyfile   : disk.bods
  force :
    id : cylinder
    parameters:
      acyl            : 0.01
      hcyl            : 0.001
      mmax            : 4
      ncylorder       : 4
      lmax            : 32
      nmax            : 32
      ncylnx          : 256
      ncylny          : 128
      logr            : false
      self_consistent : true
      PNUM            : 1
```

The last parameter, `PNUM`, sets the number of azimuthal resolution elements for computing the basis. If the distribution is axisymmetric (i.e. an initial disc), this may be safely set to 1, which results in an enormous speed-up in basis construction. You should also change the runtag setting from `run0` to `run2` in all four places it occurs in `galaxyadapt.yml`. If you now run

```
mpirun -np 64 exp -f galaxyadapt.yml
```

EXP will create a disc basis and start running the simulation. **Note that without some tinkering under the hood, the method of generating a cylindrical cache from the particles should only be used on an exponential disc. Please talk with Mike or Martin if you want to test a different density distribution.**

I have a disc-like distribution of particles and a basis. How do I compute coefficients? EXP has a cylindrical equivalent of `haloprof` called `diskprof` . The calls are slightly different (but will be homogenised!), but the basic idea is the same. The largest change is that one should feed in a cachefile (although `diskprof` will attempt to make a basis if a suitable cachefile is not found – with the same caveats as above).

A typical call will look something like

```
mpirun -np 16 diskprof  -v --cachefile=.eof.cache.run2
  --outdir=/nas/astro-th/mpetersen/NBodyTutorial/DiskHalo --coeffile=coefs
  --beg=0 --end=100 --prefix=OUT --filetype=PSPout  --runtag=run2 --compname="star disk"
```

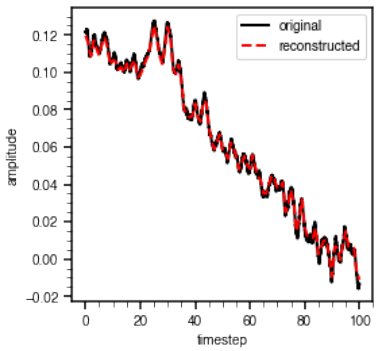

Figure 8: Comparison of the spherical coefficients from the simulation, and reconstructed.

2.3 Validating the basis

Validating bases is a large topic in it's own right, so we will not cover it in detail here. In general, the best defense against wonky bases is simply to inspect them. EXP will dump basis images if asked, and the Python-based EXPTOOL has readers that will interface with the caches. **If in doubt when designing a new basis, ask!**