Fewbody: A Brief Developer's Guide

John M. Fregeau

Kavli Institute for Theoretical Physics, UCSB, Santa Barbara, CA 93106, USA

ABSTRACT

This document serves as a brief guide for those who want to use *Fewbody* through its C interface. Looking back now on a code I wrote several years ago, if I had to do it again, *yes*, I would use different naming conventions for some things, and *yes*, I would write it in an object-oriented style with a modern language like Python or OCAML. But the code works, and once you learn the data structures and the general organization it isn't that hard to use.

Subject headings: celestial mechanics, stellar dynamics — methods: numerical

1. Introduction

The details of *Fewbody* are presented in Fregeau et al. (2004). You should at least skim that document to understand what *Fewbody* does and what it's used for. Although the command-line utilities that come with *Fewbody* are fairly flexible, you may still want to use *Fewbody* through its C interface. While I strove to keep the code organized and transparent, there are of course things I would now change thanks to the wisdom of hindsight. For example, in one data structure one of the elements has the same name as the structure data type name—not the best choice. But the code works and is relatively easy to use once you understand the data structures.

To download *Fewbody*, head over to `http://alum.mit.edu/www/fregeau/code/fewbody/`. It should be available at that address for the foreseeable future, assuming I can always find a place to host it.

2. Data Structures

The data type in which the properties of the stars and their gravitational organization is stored is called `fb_hier_t`. The structure element `.hier` of this data type holds the dynamical properties of the stars, binaries, triples, etc. Specifically, `.hier` is an array of `fb_obj_t`s (you can think of a `fb_obj_t` as either a star or a bound pair), as shown in Fig. 1 for an `fb_hier_t` initialized with 4 stars. The array is filled first with the single stars, then with the maximum possible number of binaries (2, some of which may be unused), then with the maximum possible number of triples (1, which may be unused), and so on until all possible hierarchies fill the array. For example, for a `fb_hier_t` initialized with $N = 8$, the last element of `.hier` would be an octuple. Since the `.hier` array is maximally packed, it takes some thought to determine the index of the first hierarchy of order n. Fortunately, this information is calculated and stored when the `fb_hier_t` is initialized, in the structure element `.hi` (short for "hier index"). `.hi` is an array whose element n is the first index of the hierarchy of order n. For Fig. 1, for example, the first (and only) triple has index `.hi[3]=6` (remember that C uses zero-based arrays, unlike Fortran). So if our `fb_hier_t` shown in Fig. 1 is stored in a variable called `myhier`, the second binary would be accessed like so: `myhier.hier[myhier.hi[2]+1]`. The triple would be accessed like this: `myhier.hier[myhier.hi[3]+0]`. Since the index information is stored in

`.hi` and doesn't have to be remembered by the user, it is often convenient to think of `.hier` as arranged as a sort of pyramid, as shown in Fig. 2. We'll use this layout more later.

As you may have already guessed, the stars actually "sit" in memory in the single star slots. Each higher order hierarchy contains orbital properties and points (literally, in the C sense) to two lower order hierarchies. For example, a triple would point to a binary and a single star, and contain the orbital properties of the "outer" binary. The binary it points to would contain the "inner" orbital properties, and point to two single stars. This is shown on the right side of Fig. 3, with single stars shown as filled circles and higher order hierarchies shown as open circles. Note that by construction, gravitational hierarchies are represented by binary trees in *Fewbody*. Thus each `fb_obj_t` can either point to two lower order `fb_obj_t`s, or *no* `fb_obj_t`s (each pointer set to `NULL`). This means we can't represent the stable figure-eight orbit (Heggie 2000). Since one has never been observed in nature and the theoretically expected formation rate is incredibly low, this is an approximation that seems safe.

So far we have described only how the data structures can be arranged in *Fewbody*, but not how *Fewbody* actually does things. When the `fewbody()` function returns (more of that later), the `.hier` is organized as the set of gravitational hierarchies that are unbound from each other. So if a system of 4 stars is arranged as a bound triple and a single star that are not bound to each other, it will look like the right side of Fig. 3. For convenience, `fb_hier_t` contains an element `.obj` that points to the separate hierarchies in `.hier`.

For completeness, let's go through all the pieces of the `fb_hier_t` data type. (You can follow along by looking at `fewbody.h` from your source distribution.) We've already described `.hier`, `.hi`, and `.obj`. `.narr[n]` gives the number of hierarchies of order n. For example, for Fig. 3, we have `.narr[2]=1`, `.narr[3]=1`, and `.narr[4]=0`. `.nstar` gives the current number of single stars, and would be 4 in this example. `.nobj` gives the number of used elements in the `.obj` array, and would be 2 in this example. Finally, `.nstarinit` is the number of stars that was used to initialize the `fb_hier_t`. `.nstar` can be less than `.nstarinit` if stars have collided and merged.

Let's talk about the `fb_obj_t` data type. This is the data type that makes up `.hier` and `.obj` in `fb_hier_t`. It represents either a node or a leaf in a gravitational binary tree. Since physically it represents a star or a bound pair of somethings, it's name is meant to evoke a generic gravitational "object". If you take a look at the definition of this data type in `fewbody.h`, you'll see this structure has many elements. Most notably, it has an array of two pointers to other `fb_obj_t`s in the element `.obj`. If these pointers are `NULL` the object is a single star. If they both point to something it's a bound pair of somethings. (If only one pointer is set something went terribly wrong.) The total number of stars contained in all lower levels of the tree can be found in the element `.n`. It should be evident directly from the data type definition that there are elements that describe dynamical properties (mass, position, velocity), single star specific quantities (radius, internal energy, internal angular momentum), and quantities that are only used when the `fb_obj_t` represents a bound pair (semi-major axis, eccentricity, angular momentum unit vector, Runge-Lenz vector, orbital reference time, and mean anomaly). `.ncoll` is an element specific to single stars, and tells you how

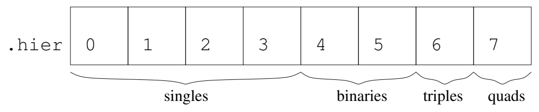

Fig. 1.— The element `.hier` of the `fb_hier_t` data type is an array of `fb_obj_t`s. This is an example for a `fb_hier_t` initialized with 4 stars.

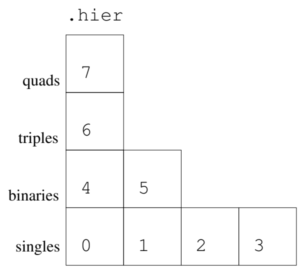

Fig. 2.— It is often convenient to think of the `.hier` element as arranged in a sort of pyramid.

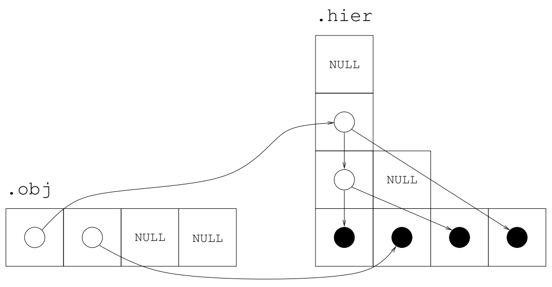

Fig. 3.— The stars are arranged in hierarchies, as shown on the right side of this figure. The `.obj` element of `fb_hier_t` is an array that points to the separate hierarchies in the system. Each hierarchy pointed to in `.obj` is gravitationally *unbound* from the other hierarchies. So in this example there are separately a triple and a single star that are unbound from each other.

many stellar mergers this star represents. If it is 1 there have been no mergers, if it is n there have been $n-1$ mergers. `.ncoll` is the used length of the `.id` array, which contains the unique user-assigned identifiers of all the stars that have merged in this star.

3. Basic Usage

Although the source for the command line utilities that come with *Fewbody* may appear complicated, there are in fact just a few things that need to be set to have *Fewbody* evolve a system. First, a `fb_hier_t` has to be initialized with the desired number of stars; let's say we have one called `myhier`. Then the properties of the single stars have to be set in `myhier.hier`. Note that the units used (for lengths, masses, time, etc.) can be any units you like, as long as it is a system of units in which the gravitational constant $G = 1$. You may make use of the `fb_units_t` data type for this purpose, if you wish. Next, you must set some parameters in a `fb_input_t`—we'll call it `myinput`—which include integration parameters and tolerances, and the stopping time of the calculation. Then you're ready to run *Fewbody*. If we call our time variable `t`, then we run *Fewbody* like this:

```
  myretval = fewbody(myinput, &myhier, &t);
```

where `myretval` will contain output information from *Fewbody* in the `fb_ret_t` type. The time `t` will be updated with the time at the end of the calculation, and `myhier` will be returned with the gravitational organization of the N-body system at the end of the calculation. As we will see below, `myhier` can be traversed using a variety of techniques to get the hierarchical organization of the system, extract values, etc.

4. An Example

How about an example showing the minimum amount of setup needed to start a *Fewbody* run? For this example, let's assume we have 5 stars that we want to initialize with given positions and velocities, and see what the outcome is. Units are often a subtle and tricky thing. Let's initialize everything in cgs units, then define our units with a routine we write. *Fewbody* will do the normalization for us. I assume you're familiar enough with C to properly prototype all functions, include the appropriate headers, etc. Here we go:

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <getopt.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_rng.h>
#include "../fewbody.h"

int calc_units(fb_hier_t hier, fb_units_t *units)
{
  /* mass unit is total system mass */
  units->m = hier.hier[hier.hi[1]+0].m + hier.hier[hier.hi[1]+1].m +    \
    hier.hier[hier.hi[1]+2].m + hier.hier[hier.hi[1]+3].m +             \
    hier.hier[hier.hi[1]+4].m;
  /* length unit is one AU */
  units->l = FB_CONST_AU;
  /* everything else is derived */
  units->E = FB_CONST_G * fb_sqr(units->m) / units->l;
  units->v = sqrt(units->E/units->m);
  units->t = units->l / units->v;
  return(0);
}

int main(int argc, char *argv[])
{
  int j;
  double t;
  fb_hier_t hier;
  fb_input_t input;
  fb_ret_t retval;
  fb_units_t units;
  char string1[FB_MAX_STRING_LENGTH], string2[FB_MAX_STRING_LENGTH];

  /* set input parameters */
  input.ks = 0; /* turn K-S regularization off */
  input.tstop = 1.0e4; /* stopping time in units of units.t */
  input.Dflag = 0; /* don't output dynamical info to stdout */
  input.dt = 0.0; /* irrelevant when Dflag=0 */
  input.tcpustop = 120.0; /* stopping CPU time in seconds */
  input.absacc = 1.0e-9; /* integrator absolute accuracy */
  input.relacc = 1.0e-9; /* integrator relative accuracy */
  input.ncount = 500; /* number of integration steps between calls to fb_classify() */
  input.tidaltol = 1.0e-6; /* tidal perturbation required to force numerical */
                           /* integration of a binary node */
  input.fexp = 1.0; /* radius expansion factor of merger products */
  fb_debug = 0;

  /* initialize a few things */
  t = 0.0;
  hier.nstarinit = 5;
  hier.nstar = 5;
  fb_malloc_hier(&hier);
  fb_init_hier(&hier);

  /* set stellar properties */
  for (j=0; j<hier.nstar; j++) {
```

```
  hier.hier[hier.hi[1]+j].ncoll = 1;
  hier.hier[hier.hi[1]+j].id[0] = j;
  snprintf(hier.hier[hier.hi[1]+j].idstring, FB_MAX_STRING_LENGTH, "%d", j);
  hier.hier[hier.hi[1]+j].n = 1;
  hier.hier[hier.hi[1]+j].obj[0] = NULL;
  hier.hier[hier.hi[1]+j].obj[1] = NULL;
  hier.hier[hier.hi[1]+j].Eint = 0.0;
  hier.hier[hier.hi[1]+j].Lint[0] = 0.0;
  hier.hier[hier.hi[1]+j].Lint[1] = 0.0;
  hier.hier[hier.hi[1]+j].Lint[2] = 0.0;
  hier.hier[hier.hi[1]+j].R = 1.0e-3 * FB_CONST_RSUN;
  hier.hier[hier.hi[1]+j].m = FB_CONST_MSUN;
  hier.hier[hier.hi[1]+j].x[2] = 0.0; /* everything in x-y plane */
  hier.hier[hier.hi[1]+j].v[2] = 0.0;
}

/* set some positions and velocities */
hier.hier[hier.hi[1]+0].x[0] = 10.0 * FB_CONST_AU;
hier.hier[hier.hi[1]+0].x[1] = 0.0 * FB_CONST_AU;
hier.hier[hier.hi[1]+0].v[0] = 10.0e5;
hier.hier[hier.hi[1]+0].v[1] = 0.0;

hier.hier[hier.hi[1]+1].x[0] = 10.0 * FB_CONST_AU;
hier.hier[hier.hi[1]+1].x[1] = 10.0 * FB_CONST_AU;
hier.hier[hier.hi[1]+0].v[0] = -10.0e5;
hier.hier[hier.hi[1]+0].v[1] = 0.0;

hier.hier[hier.hi[1]+2].x[0] = -30.0 * FB_CONST_AU;
hier.hier[hier.hi[1]+2].x[1] = -5.0 * FB_CONST_AU;
hier.hier[hier.hi[1]+0].v[0] = 0.0;
hier.hier[hier.hi[1]+0].v[1] = 5.0e5;

hier.hier[hier.hi[1]+3].x[0] = 0.0 * FB_CONST_AU;
hier.hier[hier.hi[1]+3].x[1] = 27.0 * FB_CONST_AU;
hier.hier[hier.hi[1]+0].v[0] = -5.0e5;
hier.hier[hier.hi[1]+0].v[1] = -4.0e5;

hier.hier[hier.hi[1]+4].x[0] = 10.0 * FB_CONST_AU;
hier.hier[hier.hi[1]+4].x[1] = -32.0 * FB_CONST_AU;
hier.hier[hier.hi[1]+0].v[0] = 5.0e5;
hier.hier[hier.hi[1]+0].v[1] = -1.0e5;

/* set units with our own routine, then normalize */
calc_units(hier, &units);
fb_normalize(&hier, units);
```

```
  /* call Fewbody to evolve system */
  retval = fewbody(input, &hier, &t);

  /* all the rest is parsing the output */
  if (retval.retval == 1) {
    fprintf(stderr, "encounter complete.\n");
  } else {
    fprintf(stderr, "encounter NOT complete.\n");
  }

  fprintf(stderr, "final configuration:  %s  (%s)\n",
          fb_sprint_hier(hier, string1),
          fb_sprint_hier_hr(hier, string2));

  fprintf(stderr, "t_final=%.6g (%.6g yr)  t_cpu=%.6g s\n",     \
          t, t*units.t/FB_CONST_YR, retval.tcpu);

  fprintf(stderr, "DeltaL/L0=%.6g  DeltaL=%.6g\n", retval.DeltaLfrac, retval.DeltaL);
  fprintf(stderr, "DeltaE/E0=%.6g  DeltaE=%.6g\n", retval.DeltaEfrac, retval.DeltaE);
  fprintf(stderr, "Rmin=%.6g (%.6g RSUN)  Rmin_i=%d  Rmin_j=%d\n",      \
          retval.Rmin, retval.Rmin*units.l/FB_CONST_RSUN, retval.Rmin_i, retval.Rmin_j);
  fprintf(stderr, "Nosc=%d (%s)\n", retval.Nosc,
          (retval.Nosc>=1?"resonance":"non-resonance"));

  fprintf(stderr, "orbital parameters of outermost binaries:\n");
  for (j=0; j<hier.nobj; j++) {
    if (hier.obj[j]->n >= 2) {
      fprintf(stderr, "j=%d  a=%.6g AU  e=%.6g\n", j,
              hier.obj[j]->a * units.l / FB_CONST_AU, hier.obj[j]->e);
    }
  }

  fb_free_hier(hier);
  return(0);
}
```

And here is the output of this code on my machine (it may differ from machine to machine due to round-off error and the inherent chaos in the N-body problem):

```
encounter NOT complete.
final configuration:  nstar=5 nobj=1:  [[[0 2] [1 3]] 4]  (quintuple)
t_final=10000.9 (711.854 yr)  t_cpu=0.47 s
DeltaL/L0=3.60588e-08  DeltaL=1.08282e-09
DeltaE/E0=-2.94682e-07  DeltaE=3.94031e-09
Rmin=0.000125687 (0.0270158 RSUN)  Rmin_i=2  Rmin_j=4
Nosc=0 (non-resonance)
```

```
orbital parameters of outermost binaries:
j=0  a=19.1876 AU  e=0.744792
```

My hope is that the code example above answers more questions than it raises. It may seem verbose, but I encourage you to read through it line by line, as I have tried to comment the important parts.

5. Another Example

Suppose you want to do something more advanced with *Fewbody*. Suppose you have evolved a system with a fairly large number of stars, and want to find the relative inclination in any triples that result. Here's how you might do something like this:

```
for (j=0; j<hier.nobj; j++) {
  if (hier.obj[j]->n == 3) {
    /* determine which element is inner binary */
    if (hier.obj[j]->obj[0]->n == 1) {
      is = 0;
      ib = 1;
    } else {
      is = 1;
      ib = 0;
    }

    /* calculate outer angular momentum  */
    fb_cross(hier.obj[j]->obj[0]->x, hier.obj[j]->obj[0]->v, l0);
    fb_cross(hier.obj[j]->obj[1]->x, hier.obj[j]->obj[1]->v, l1);

    for (i=0; i<3; i++) {
      Lout[i] = hier.obj[j]->obj[0]->m * l0[i] + hier.obj[j]->obj[1]->m * l1[i];
    }

    /* calculate inner angular momentum */
    fb_cross(hier.obj[j]->obj[ib]->obj[0]->x, hier.obj[j]->obj[ib]->obj[0]->v, l0);
    fb_cross(hier.obj[j]->obj[ib]->obj[1]->x, hier.obj[j]->obj[ib]->obj[1]->v, l1);

    for (i=0; i<3; i++) {
      Lin[i] = hier.obj[j]->obj[ib]->obj[0]->m * l0[i] +
               hier.obj[j]->obj[ib]->obj[1]->m * l1[i];
    }

    /* acos returns a value between 0 and PI */
    inc = acos(fb_dot(Lin, Lout)/(fb_mod(Lin)*fb_mod(Lout)));
    fprintf(stderr, "j=%d  inc=%g radians\n", j, inc);
  }
}
```

6. The Functional Interface

By request, I have created a rudimentary functional interface to *Fewbody*, which uses function calls to access and set elements of a `fb_hier_t`, traverse its structure, etc. The functions are essentially shortcuts to using the arrays defined in the data types. Here's one self-explanatory example from `fewbody_ui.c`:

```
int fbui_make_pair(fb_obj_t *parentobj, fb_obj_t *obj1, fb_obj_t *obj2)
{
  parentobj->obj[0] = obj1;
  parentobj->obj[1] = obj2;
  return(0);
}
```

And you use it like this:

```
  fbui_make_pair(fbui_hierarchy_binary(&hier, 0),
                 fbui_hierarchy_single(&hier, 1),
                 fbui_hierarchy_single(&hier, 2));
```

I have created a full example of how to use the functional interface in the file `binsingle_simple.c`. Please have a look at it to see how this interface is used in practice. In it you will also notice some other useful utilities provided by *Fewbody*, including a routine to randomly orient a binary (when you don't care about orientation, phase, etc., and want to average over them in an ensemble of scattering calculations), and a routine to bring a pair of objects in from infinity along a hyperbolic orbit to the point where you want to start numerical integration (this is very useful for scattering experiments).

7. Appendix

For completeness, this section provides details on *Fewbody* data types, in the form of tables. The other *Fewbody* data types are documented fairly well in `fewbody.h`.

REFERENCES

Fregeau, J. M., Cheung, P., Portegies Zwart, S. F., & Rasio, F. A. 2004, MNRAS, 352, 1

Heggie, D. C. 2000, MNRAS, 318, L61

This preprint was prepared with the AAS $\LaTeX$ macros v5.0.

`.ncoll`	length of `.id` array
`.id`	array of stellar ids
`.idstring`	object id in string form
`.m`	mass
`.R`	radius (stars only)
`.Eint`	internal energy (stars only)
`.Lint`	internal angular momentum (stars only)
`.x`	position vector
`.v`	velocity vector
`.n`	number of leaves below and including this node in tree
`.obj`	array of two pointers to child nodes
`.a`	semi-major axis (bound pairs only)
`.e`	eccentricity (bound pairs only)
`.Lhat`	angular momentum unit vector (bound pairs only)
`.Ahat`	Runge-Lenz vector (bound pairs only)
`.t`	orbital reference time (bound pairs only)
`.mean_anom`	orbital mean anomaly (bound pairs only)

Fig. 4.— `fb_obj_t`, the basic data type for stars and bound pairs of somethings.

`.nstarinit`	number of stars used to initialize `fb_hier_t`
`.nstar`	current number of stars
`.nobj`	number of separate, bound hierarchies—used length of `obj` array
`.hi`	array of first indices for a given hierarchy order in `.hier`
`.narr`	array containing number of hierarchies of given order
`.hier`	array of `fb_obj_t`s holding gravitational organization of system
`.obj`	array of pointers to separate, bound hierarchies in `.hier`

Fig. 5.— `fb_hier_t`, the basic data type for gravitational hierarchies.