User Guide for falcON

version of 03/12/2002

Summary
falcON is the "Force Algorithm with Complexity $\mathcal{O}(N)$)" which is described in Dehnen (2002).
With this packages, you can use falcON in subroutine form as Poisson solver for particle based simulations.
The package also has a full N-body code, based on falcON, called gyrfalcON ("GalacY simulatoR using falcON"), which employs the N-body tool box NEMO. This code features individual adaptive time steps employing a block-step scheme, but can also be used in single-time-step mode (in which case momentum is exactly conserved).

1 Guarantee

This package comes with absolutely no guarantee whatsoever! The unpacking, installation, and usage of the code is entirely at the risk of the user alone.

2 Credit

Any scientific publication or presentation which has benefited from using any part of this package should quote the paper

Dehnen, W., 2002, JCP, 179, 27.

(please find a pdf file of this paper in the subdirectory `FALCON/doc`.)

3 Unpacking

After downloading the file `falcON.tar.gz`, copy it into some directory, say `FALCON`. Then unpack it typing (after `cd FALCON`)

```
tar zxf falcON.tar.gz,
```

which should create the sub-directories `src`, `inc`, and `doc`, as well as several other files.

4 Installation

You need to make the library `libfalcON.a` and possibly the executables you want to use, see §§ below. The code is written entirely in C++ and it is strongly recommended to use a compiler that understands standard C++, e.g. GNU's gcc version 3.2 (or higher). If you want to use any other compiler than gcc, edit the file `make.defs` and change the entry for `C++COMP`. However, I cannot recommend using the Intel compiler version 6.0 (it produces slower code). The `Makefile` is intended for use with GNU make.

In order to allow the code to understand NEMO data format and parameter I/O, you must invoke NEMO **before** compilation.

The making takes a little while but should not produce any warning or error messages. Otherwise something might be wrong. To generate the library as well as a test program `TestGrav`, type

```
make TestGrav
```

The executable `TestGrav` (as well as all other executables generated from this package) lives in a subdirectory

```
FALCON/$(MACHTYPE)_$(OSTYPE),
```

where `MACHTYPE` and `OSTYPE` are environment variables unique to the machine type and operating system. In this way, you may have versions of the executables and the library (which is in subdirectory `FALCON/$(MACHTYPE)_$(OSTYPE)/lib`) for several hosts on the same file system.

5 Testing `falcON`

Please run `TestGrav` in order to get some rough check on the validity of your library. Issuing the command

```
    TestGrav 2 1 1000000 901 0.01 1
```

shall generate a Hernquist sphere with $N = 10^6$ particles, build the tree (twice: once from scratch and once again) and compute the forces using a softening length of $\epsilon = 0.01$ scale radii with the P_1 kernel (see §6). The output of this command may look like

```
 time needed for set up of X_i:          1
 time needed for falcON::grow():         2.58
 time needed for falcON::grow():         1.63
 time needed for falcON::approximate():  9.02

 state:                 tree built
 root center:           0 0 0
 root radius:           1024
 bodies loaded:         1000000
 total mass:            1
 N_crit:                6
 cells used:            353419
 maximum depth:         21
 current theta:         0.6
 current MAC:           theta(M)
 softening length:      0.01
 softening kernel:      P1
 Taylor coeffs used:    84569
 interaction statitics:
     type          approx   direct      total
 # body-body :          -        0          0 =        0%
 # cell-body :    2115758   477698    2593456 =   18.342%
 # cell-cell :   11237586   254997   11492583 =   81.279%
 # cell-self :          -    53678      53678 =     0.38%
 # total     :   13353344   786373   14139717 =  100.000%

 ASE(F)/<F^2>      = 0.001598375617
 max (dF)^2        = 0.8182717562
 Sum m_i acc_i     = -2.655207831e-09 1.600820587e-09 2.495622042e-10
```

Note that the second tree-build is somewhat faster then the original one. Note also the the total-momentum change (last line) vanishes within floating point accuracy – that's a generic feature of `falcON`.

6 Choice of the Softening Kernel and Length

The code allows for various forms of the softening kernel, i.e. the function by which Newton's $1/r$ is replaced in order to avoid diverging near-neighbour forces. The following kernel functions are available ($x := r/\epsilon$)

name	density (is proportional to)	a_0	a_2	f
P_0	$(1+x^2)^{-5/2}$	∞	∞	1
P_1	$(1+x^2)^{-7/2}$	π	∞	1.43892
P_2	$7(1+x^2)^{-9/2} - 2(1+x^2)^{-7/2}$	0	∞	2.07244
P_3	$9(1+x^2)^{-11/2} - 4(1+x^2)^{-9/2}$	0	$-\pi/40$	2.56197

Note, that P_0 is the standard Plummer softening, however, **recommended** is the use of P_1 and P_2. There are several important issues one needs to know about these various kernels.

First, the softening length ϵ is just a parameter and using the same numerical value for it but different kernels corresponds in effect to different amounts of softening. Actually, this softening is weakest for the Plummer sphere: at fixed ϵ, the maximal force is smallest. In order to obtain comparable amounts of softening, larger ϵ are needed with all the other kernels. An idea of the factor by which ϵ has to be enlarged can be obtained by

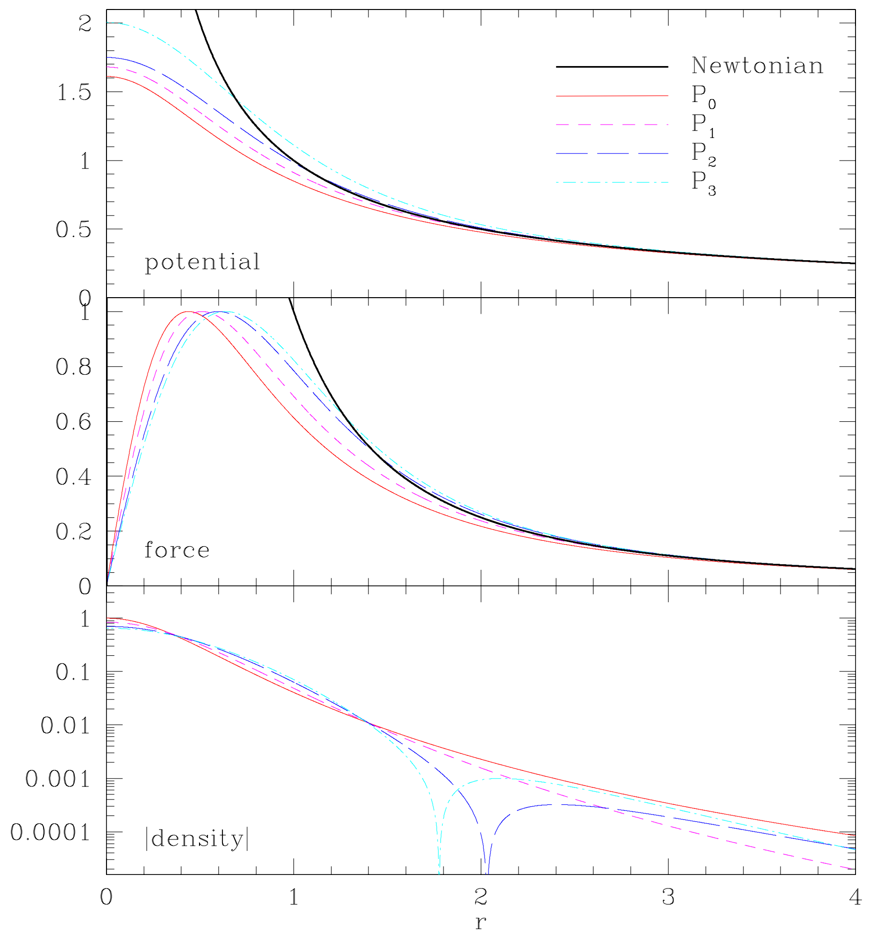

Figure 1: Potential, force, and density for the softening kernels of the table, including the standard Plummer softening (P_0). The softening lengths ϵ are scaled such that the maximum force equals unity. The kernels $P_{>0}$ approach Newtonian forces more quickly at larger r than does P_0. The kernels P_2 and P_3 have slightly super-Newtonian forces (and negative densities) in their outer parts, which compensate for the sub-Newtonian forces at small r.

setting ϵ such that the maximum possible force between any two bodies are equal for various kernels. The last column in the previous table gives these factors. Note, that using a larger ϵ with other than the P_0 kernel does **not** mean that your resolution goes down, it in fact increases, see Dehnen (2001), but the Poisson noise is more suppressed with larger ϵ. It is recommended not to use Plummer softening, unless (i) in 2D simulations, as here ϵ is the average scale-height of the disk, and, perhaps, (ii) in simulations made to compare with others that use Plummer softening (for historical reasons).
Second, as shown in Dehnen (2001), Plummer softening results in a strong force bias, due to its slow convergence to the Newtonian force at $r \gg \epsilon$. This is quantified by the measure a_0, which for P_0 is infinite. In Dehnen (2001), I considered therefore other kernels (not mentioned above), which have finite support, ie. the density is exactly zero for $r \geq \epsilon$. This discontinuity makes them less useful for the tree code (which is based on a Taylor expansion of the kernel). In order to overcome this difficulty, the kernels P_1 to P_3, which are continuous in all derivatives, have been designed as extensions to the Plummer softening, but with finite a_0 (P_1), zero a_0 but infinite a_2 (P_2), or even zero a_0 and finite a_2 (P_3).

7 Choice of the Tolerance Parameter

The code falcON approximates an interaction between two nodes, if their critical spheres don't overlap. The critical spheres are centered on the nodes' centers of mass and have radii

$$r_{\rm crit} = r_{\rm max}/\theta \qquad (1)$$

where $r_{\rm max}$ is the radius of a sphere that is guaranteed to contain all bodies of the node (bodies have $r_{\rm max} = 0$), while θ is the tolerance parameter. The default is to use a mass-dependent $\theta = \theta(M)$ with $\theta_0 \equiv \theta(M_{\rm tot})$ being the parameter, see Dehnen (2002). For near-spherical systems or groups of such systems, θ_0 of 0.6 gives relative forces error of the order of 0.001, which is generally believed to be acceptable. However, the force error might often be dominated by discreteness noise, in which case a larger value does no harm. For disk systems, however, a small theta might be a better choice.
The recommendation is to either stick to θ_0 no larger than about 0.6, or perform some experiments with varying θ_0 (values larger than 0.8, however, make no sense, as there is hardly any speed-up).

8 Use of `falcON` as Poisson Solver

8.1 with C++

In order to make use of the code, you need to insert the C macro

```
    #include <falcON.h>
```

somewhere at the beginning of your C++ source code. Make sure that the compiler finds the file `falcON.h` by including "`-I FALCON/inc`" among your compiler options. The usage of the code in your application is explained in gory detail in the file `falcON.h` (don't forget that `class falcON` lives in namespace `nbdy`). In order to make an executable, add the linker options "`-LFALCON/$(MACHTYPE)_$(OSTYPE)/lib -lfalcON -lm`" (expand the macros in your makefile) so that the library is loaded.
For examples of code using `falcON.h`, see the files `TestGrav.cc` and `TestPair.cc` in subdirectory `src/mains/`, which may be compiled by typing "`make TestGrav`" and "`make TestPair`" and produce a short summary of their usage when run without arguments.

8.2 with C

In order to make use of the code, you need to insert the C macro

```
    #include <falcON_C.h>
```

somewhere at the beginning of your C source code. Make sure that the compiler finds the file `falcON_C.h` by including "`-I FALCON/inc`" among your compiler options. The usage of the code in your application is explained in gory detail in the file `falcON_C.h`. In order to make an executable, add the linker options "`-LFALCON/$(MACHTYPE)_$(OSTYPE)/lib -lfalcON -lstdc++ -lm`" (expand the macros in your makefile) so that the library is loaded.
For examples of code using `falcON_C.h`, see the files `TestGravC.cc` and `TestPairC.cc` in subdirectory `src/mains/`, which may be compiled by typing "`make TestGravC`" and "`make TestPairC`" and produce a short summary of their usage when run without arguments.

8.3 with FORTRAN

In order to make use of the code, you need to insert

```
    INCLUDE 'falcON_C.h.f
```

somewhere at the beginning of your FORTRAN program. Make sure that the compiler finds the file `falcON.f` by including "`-I FALCON/inc`" among your compiler options. The usage of the code in your application is explained in gory detail in the file `falcON.f`. In order to make an executable, add the linker options "`-LFALCON/$(MACHTYPE)_$(OSTYPE)/lib -lfalcON -lstdc++ -lm`" (expand the macros in your makefile) so that the library is loaded.
For examples of code using `falcON.f`, see the files `TestGravF.F` and `TestPairF.F` in subdirectory `src/mains/`, which may be compiled by typing "`make TestGravF`" and "`make TestPairF`". Just run these code, they are self-explanatory and provide some statistics output. You may also use the input files given and run them as "`TestGravF < treeF.in`" and "`TestPairF < pairF.in`".

9 The N-body code `gyrfalcON`

The package also contains a full N-body code, called "`gyrfalcON`" (GalaxY simulatoR using `falcON`)[1]. If you want to use this code, you need first to install and invoke the N-body tool box `NEMO`, version 3.0.13 or

[1]Called "YancNemo" in former versions of this package.

higher[2], see http://www.astro.umd.edu/nemo.
Then type

```
make gyrfalcON
```

which should produce the executable `gyrfalcON` in the subdirectory

```
FALCON/$(MACHTYPE)_$(OSTYPE)
```

(add it to your `$PATH`).
`gyrfalcON` comes with the usual `NEMO` help utility: calling

```
gyrfalcON help=h
```

produces the following overview over the options.

```
in                : input file                                         [???]
out               : file for primary output; required, unless resume=t  []
tstop             : final integration time                             [10]
step              : time between primary outputs; 0 -> every step      [1]
logfile           : file for log output                                [-]
out2              : file for secondary output stream                   []
step2             : time between secondary outputs; 0 -> every step    [0]
theta             : tolerance parameter at M=M_tot                     [0.6]
hgrow             : grow fresh tree every 2^hgrow smallest steps       [0]
Ncrit             : max # bodies in un-split cells                     [6]
eps               : softening length OR maximum softening length       [0.05]
kernel            : softening kernel of family P_n (P_0=Plummer)       [1]
hmin              : tau_min = (1/2)^hmin                               [6]
Nlev              : # time-step levels                                 [1]
fac               : tau = fac/acc          \   If more than one of     []
fph               : tau = fph/pot           |  these is non-zero, we   []
fpa               : tau = fpa*sqrt(pot)/acc /   use the minimum tau.   []
resume            : resume old simulation?  that implies:
                    - read last snapshot from input file
                    - append primary output to input (unless out given) [f]
give              : list of output specifications. Recognizing:
                      m: mass                            (default)
                      x: position                        (default)
                      v: velocity                        (default)
                      a: acceleration
                      p: N-body potential
                      l: time-step level (if they exist)
                      f: body flag
give2             : list of specifications for secondary output        [mxv]
Grav              : Newton's constant of gravity                       [1]
potname           : name of external potential                         []
potpars           : parameters of external potential                   []
potfile           : file required by external potential                []
startout          : primary output for t=tstart?                       [t]
VERSION           : 03/December/2002  WD
                    compiled  Dec  3 2002, 10:52:40                    [1.5]
```

The last column indicates the default value, with '`[???]`' indicating that the value for the keyword must be given. A filename '`-`' means that output is written to stdout and can be piped into another command. A filename '`.`' means that no output is made at all.

9.1 log output

By default, log output is written to stdout, which prevents the usage of the pipe to transfer N-body data. By setting '`logfile`' one may overcome this.

[2]Older versions of this package contained a non-`NEMO` code, called "YANC". This code was never properly tested and has hence been deprecated.

9.2 Data I/O

The initial conditions, including the simulation time, are read in from the file given with the parameter '`in`' and **must** be in `NEMO` snapshot format. If '`in=-`', the code expects input from a pipe. Unless the keyword '`resume=t`' (see below) the first snapshot in this file is used.
The code allows for two distinct data outputs, one into '`out`' and another optional one into '`out2`'. Output is made every '`step`' and '`step2`' time units, respectively. The type of N-body data written is controlled by the options '`give`' and '`give2`', which are character strings containing the letters '`m`', '`x`', '`v`', '`a`', '`p`', '`l`', '`f`', indicating that the masses, positions, velocities, accelerations, potentials, time-step levels, and body flags shall be given. The default is '`mxv`', i.e. masses, positions, and velocities.
Traditionally on linux systems, there is a limit of 2Gb on the size of files. This will cause trouble with `NEMO` snapshot files, since the snapshots of all output times are written to one file. To overcome this, you must (i) configure `NEMO` appropriately (use '`configure --enable-lfs`' when installing and (ii) ensure that your file systems supports large files – consult your system administrator.

9.3 `falcON` parameters

The parameters '`eps`' and '`kernel`' control the softening, see $6. The parameter '`Ncrit`' sets the maximum number of bodies in a leaf cell of the tree. The default value of `Ncrit` is set to yield highest performance.

9.4 Time step and simulation time

The (shortest) time step is controlled by the parameter '`hmin`' and is equal to

$$\tau_{\min} = 2^{\texttt{hmin}}.$$

The simulations stops if the simulation time has been advanced to the value of the parameter '`tstop`'. If `tstop` equals the initial simulation time, the initial forces are computed and, if so desired, output is made for this time only.

9.5 Resuming an old simulation

An old simulation may be resumed by setting the keyword '`resume`'. In this case, the last snapshot from the input file is taken as initial conditions and output is appended to the input file.

9.6 Adding an external potential

On top of the N-body forces, an external potential may be added using the usual `NEMO` keywords '`potname`', '`potpars`', and '`potfile`'. When '`potname`' is given, a corresponding shared object file is loaded dynamically and the potential is initialized with the parameters given with '`potpars`' or the data file given with '`potfile`', whatever applies.

9.7 Adaptive time stepping

You can use `gyrfalcON` without adaptive time steps, by setting '`Nlev=1`' (default). In this case, also much of the (small) overhead that is necessary for adaptive time steps is avoided and a standard leap-frog time integrator is used.
When '`Nlev`'> 1, a block-step scheme with `Nlev` time step levels is used, i.e. the longest step contains $2^{\texttt{Nlev}-1}$ shortest steps. The bodies' individual time-step levels are adapted in an (almost) time symmetric fashion to be on average

$$\tau = \min\left(\frac{\texttt{fac}}{a}, \frac{\texttt{fph}}{|\Phi|}, \texttt{fph}\sqrt{\frac{|\Phi|}{a}}\right), \tag{2}$$

where Φ and a are the gravitational potential and the modulus of the acceleration. The parameters `fac`, `fph`, and `fpa` determine the stepping. If either of them is zero, it is ignored.
In order to make a sensible choice for the parameters '`hmin`', '`Nlev`', and '`fac`', '`fac`', use the following method. (1) Decide on the smallest time step: think what time step you would use in a single-time-step leap-frog scheme and then set $\tau_{\min}$ to about half of that. (2) Decide on the largest time step, whereby ensuring that

orbits in regions of very low density are accurately integrated when using the above criterion (2). (3) Do some tests with varying '`fac`', '`fph`', and '`fpa`' (set '`tstop`' to 0), in order to check that the distribution of bodies over the time steps is reasonable, in particular there should be a few percent in the smallest time step.
When using adaptive time stepping, it may be worth your while to use a larger value for '`Ncrit`' than default. This reduces the time of tree-building and increases that for the force computation. However with adaptive time stepping the relative contribution of tree-building to the total CPU time budget is much larger than without.
Note that using this scheme is sensible only if you really have a very inhomogeneous stellar system, because otherwise, the simple single-time-step leap-frog is only slightly less efficient but somewhat more accurate. In particular, with the block-step scheme, the total momentum is not conserved, but with the single-time-step leap-frog it is.

9.8 Examples

In order to integrate a Plummer sphere with $N = 10^5$ particles, you may issue the command

```
mkplummer - 100000 seed=1 scale=1 | gyrfalcON - eps=0.1 plum.snp
```

which first creates initial conditions from a Plummer model, which are then piped into `gyrfalcON`. `gyrfalcON` creates an output file '`plum.snp`' containing output every full time unit until time $t = 10$. The log output looks like

```
# ----------------------------------------------------------------------------------------
# "gyrfalcON - eps=0.1 plum.snp VERSION=1.5"
#
# run at  Tue Dec  3 12:11:36 2002
#     by  "dehnen"
#     on  "milkyway"
#
#    time      energy       -T/U     |L|       |v_cm|  build   force    step     accum
# ----------------------------------------------------------------------------------------
 0          -0.1473791    0.50316 0.0010761  6e-09    0.2    0.98       1.18      1.18
 0.015625   -0.147379     0.50317 0.0010762  1.9e-09  0.15   0.99       1.15      2.36
 0.03125    -0.1473791    0.50316 0.0010762  5.6e-09  0.15   0.98       1.14      3.53
 0.046875   -0.147379     0.50317 0.0010762  5.1e-09  0.14   1          1.15      4.71
 .
 .
 .
 9.9531     -0.1473785    0.49925 0.0010798  4.1e-09  0.15   0.99       1.14    758.39
 9.9688     -0.1473783    0.49926 0.0010799  5.8e-09  0.14   1          1.15    759.57
 9.9844     -0.1473778    0.49927 0.0010799  2.3e-09  0.14   1.01       1.16    760.76
 10         -0.1473785    0.49927 0.0010799  5.8e-09  0.15   1          1.17    761.96
```

The column '`|v_cm|`' gives the center-of-mass motion, which stays constant (within floating point precision) due to the momentum-conserving nature of `falcON`. The last four columns contain the CPU time in seconds spent on the tree building, force computation, and full time step, as well as the accumulated time.

10 `addgravity` and `getgravity`

The public version of the `falcON` package contains two further `NEMO` executables. `addgravity` simply adds acceleration and potential to every body in the snapshots of a `NEMO` snapshot file. `getgravity` computes the gravity generated by the particles (sources) in the snapshots of a `NEMO` snapshot file at the positions of the particles (sinks) in another snapshot. This is useful for, for instance, computing the rotation curves of N-body galaxies.

11 Bugs and Features

11.1 Test-particles

`falcON` does not support the notion of a test particle, i.e. a body with zero mass. Such bodies will never get any acceleration (that is because the code first computes the force, which is symmetric and hence better suited for mutual computations, and then divides by the mass to obtain the acceleration). To overcome this, you may use tiny masses, but note that the forces created by such light bodies will be computed, even if they are tiny.

Actually, this is exactly what we do in `getgravity`.

11.2 Bodies at identical positions

The code cannot copy with more than `Ncrit` bodies at an identical position (within floating point accuracy). Such a situation would result in an infinitely deep tree; the code aborts with an error message.

11.3 A mysterious bug

There seems to exist an odd bug that I'm currently trying to understand: Occasionally, a run of `gyrfalcON` crashes with 'Segmentation fault'. However, apparently nothing is wrong with the simulation data, and the simulation can be continued (via the '`resume`' option) from the last snapshot output. This error is not reproducible and hence hard to track down and weed out.

12 References

Dehnen, W., 2001, MNRAS, 324, 273
Dehnen, W., 2002, JCP, 179, 27