User Guide for falcON

version of November 12, 2003

Summary. `falcON` is the "Force Algorithm with Complexity $\mathcal{O}(N)$)" which is described by Dehnen (2000, 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` ("GalaxY 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 papers

Dehnen, W., 2000, ApJ, 536, L39,
Dehnen, W., 2002, JCP, 179, 27.

(please find pdf files of these papers in the subdirectory `falcON/doc`.)

3 Unpacking

After downloading the file `falcON.tgz`, unpack it typing

```
tar zxf falcON.tgz,
```

which should create the directory `falcON` with 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++, I recommend GNU's gcc version 3.3, but not versions earlier than 3.2. 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 (it produces slower code, I tried versions 6.0 and 7.0). 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.

To generate the library as well as a test program `TestGrav` and the N-body code `gyrfalcON`, type

```
make
```

The making takes a little while but should not produce any warning or error messages. Otherwise something might be wrong. The executables live 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 Individual Softening Lengths

Individual softening lengths are a new feature (as of Sep-2003, before they have been restricted to the proprietary version). They are enabled, but not obligatory (in fact default is always to have a globally constant ϵ), if line 19 of the `Makefile`

```
DSOFT                     := -DfalcON_INDI
```

is not commented out (by a # in the first column).

The softening length ϵ_{ij} used in the interaction of nodes with individual softening lengths ϵ_i and ϵ_j is simply the arithmetic mean of the two. The softening length ϵ_i of a cell is the arithmetic mean of the softening lengths of all its bodies. See also section **??** below.

6 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 §7). The output of this command may look like

```
 time needed for set up of X_i:                 0.89
 time needed for falcON::grow():                1.74
 time needed for falcON::grow():                0.76
 time needed for falcON::approximate_gravity():  8.1

 state:                 tree re-grown
 root center:           0 0 0
 root radius:           1024
 bodies loaded:         1000000
 total mass:            1
 N_crit:                6
 cells used:            353419
 of which were active   353419
 maximum depth:         21
 current theta:         0.6
 current MAC:           theta(M)
 softening:             global
 softening length:      0.01
 softening kernel:      P1
 Taylor coeffs used:    84282 in 4 chunks of 22092
 interaction statitics:
     type          approx   direct      total
 # body-body :          -        0          0 =        0%
 # cell-body :    2125639   479315    2604954 =   18.325%
 # cell-cell :   11301382   254998   11556380 =   81.297%
 # cell-self :          -    53678      53678 =    0.378%
 # total     :   13427021   787991   14215012 =  100.000%

 ASE(F)/<F^2>      = 0.001597663853
 max (dF)^2        = 0.8176034689
 Sum m_i acc_i     = -1.562529217e-09 1.506183733e-09 6.141109288e-10
```

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

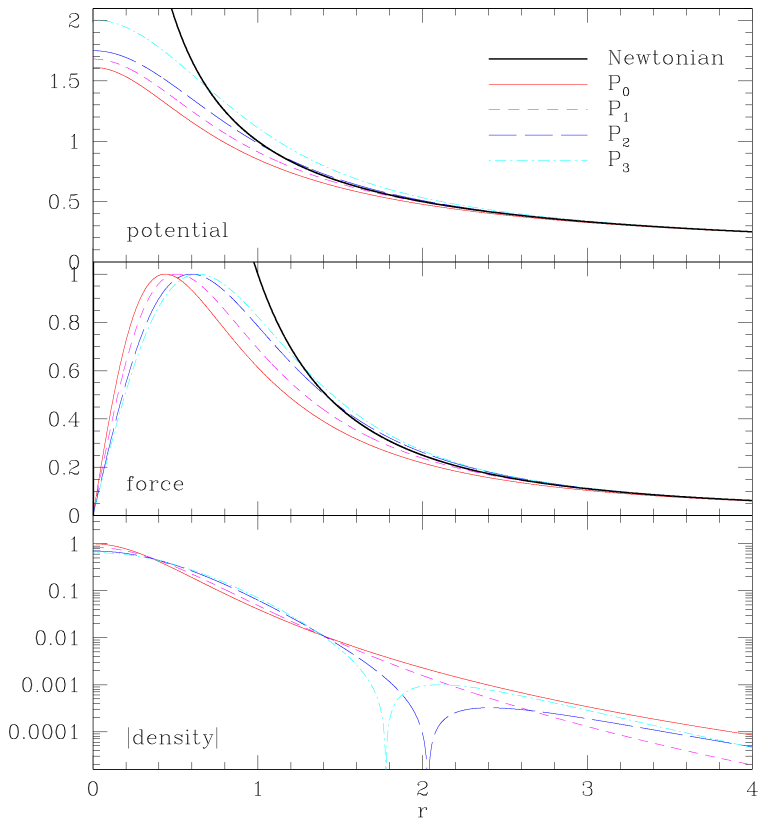

Figure 1: Potential, force, and density for the softening kernels of the table, including the standard Plummer softening (P_0). The softening lengths ϵ are scaled such that the maximum force equals unity. The kernels $P_{>0}$ approach Newtonian forces more quickly at larger r than does P_0. The kernels 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.

7 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 or 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 strongest 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 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) you want $\epsilon \equiv 0$, (ii) in 2D simulations, as here ϵ is the average scale-height of the disk, and, perhaps, (iii) 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).

8 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

$$r_{\mathrm{crit}} = r_{\mathrm{max}}/\theta \tag{1}$$

where r_{max} is the radius of a sphere that is guaranteed to contain all bodies of the node (bodies have $r_{\mathrm{max}} = 0$), while θ is the tolerance parameter. The default is to use a mass-dependent $\theta = \theta(M)$ with $\theta_0 \equiv \theta(M_{\mathrm{tot}})$ being the parameter, see Dehnen (2002). For near-spherical systems or groups of such systems, θ_0 of 0.6 gives relative force errors 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 smaller tolerance parameter, e.g. $\theta_0 = 0.5$, 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).

9 Use of falcON as Poisson Solver

9.1 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.

9.2 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.

9.3 With FORTRAN

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

```
INCLUDE 'falcON.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 programs, 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`.

10 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 higher[2], see http://www.astro.umd.edu/nemo. It is recommended to configure `NEMO` with `configure --enable-single --enable-lfs`.

Then type

```
make gyrfalcON
```

which should produce the executable `gyrfalcON` in the subdirectory (add it to your `$PATH`)

```
falcON/$(MACHTYPE)_$(OSTYPE).
```

`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 [default: never]           []
step              : time between primary outputs; 0 -> every step     [1]
logfile           : file for log output                               [-]
stopfile          : stop simulation as soon as file exists            []
logstep           : # blocksteps between log outputs                  [1]
out2              : file for secondary output stream                  []
step2             : time between secondary outputs; 0 -> every step   [0]
theta             : tolerance parameter at M=M_tot                    [0.64]
hgrow             : grow fresh tree every 2^hgrow smallest steps      [0]
Ncrit             : max # bodies in un-split cells                    [16]
eps               : >=0: softening length
                    < 0: use individual fixed softening lengths       [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,    []
fpa               : tau = fpa * sqrt(pot)/acc |  we use the minimum    []
fea               : tau = fea * sqrt(eps/acc) /  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)
```

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

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

```
                      x: position                          (default)
                      v: velocity                          (default)
                      a: acceleration
                      p: N-body potential
                      P: external Pot (added to pot before output)
                      e: individual eps_i (if they exist)
                      l: time-step level (if they exist)
                      f: body flag                                      [mxv]
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]
lastout          : primary output for t=tstop?                          [t]
VERSION          : 17-sep-2003 Walter Dehnen                            [1.7.1PSI.gcc-3.2]
COMPILED         : Oct  7 2003, 15:56:10, with gcc-3.2, on andromeda.star.le.ac.uk []
```

The last column indicates the default value, with '`[???]`' indicating that the value for the keyword must be given, while '`[]`' means that the corresponding feature is not used by default.

10.1 Parameters of `gyrfalcON`

10.1.1 Parameters Controlling I/O

`in` is the file from which the initial conditions (including the initial simulation time) are read. Unless the keyword `resume=t` (see below) the first snapshot in this file is used. If `in=-`, `stdin` instead of a file are used for data input. This is useful for piping the data from another program creating the data, see §10.2 for an example.

`out` is the file for primary data output. If `out=-`, data are written to `stdout` (useful for piping into an analysis tool), and if `out=.` no output is made.

`give` specifies which data are given with primary data output. `give` must be a unbroken list of characters, each of which indicates a datum (per body) to be written. `m`, `x`, `v` mean mass, position, and velocity, respectively; `a`, `p`, and `P` mean acceleration, N-body potential and (if applicable) external potential, respectively. The external potential is simply added to the potential datum, i.e. the snapshot contains the sum of both potentials. `e` is individual softening length and `l` the time step level. By default `give=mxv`, i.e. masses and phase-space coordinates are given.

`step` is the time in simulation units between primary data outputs.

`startout` indicates whether primary output shall be made already for the initial simulation time.

`lastout` indicates whether primary output shall be made for the last simulation time, even if not at an ordinary output `step`.

`out2`, if given, secondary output to the file named is made. With `out2=-`, secondary outpus is written to `stdout`.

`give2` specifies in the same way as `give` which data are given with secondary data output.

`step2` gives the simulation time between secondary outputs.

`logfile` is the file to which log output (time, total energy, angular momentum, CPU time consumption etc) is written. Default is `logfile=-`, resulting in log output written to `stdout` (note however that only one type of output may sensibly written to `stdout` – the code will issue an error if you want it to write several different data streams to `stdout`).

`logstep` gives the number of blocksteps between log outputs.

`resume` this option allows to resume an old simulation which has been prematurely stopped. When `resume=t`, the last snapshot from the `in` file is used as initial condition and primary data output is appended to that same file.

10.1.2 Parameters Controlling Time Integration

`tstop` is the simulation time at which the simulation shall be stopped. By default, the simulation will not be stopped at a pre-defined time. An alternative method for stopping the simulation is by the use of the parameter `stopfile`. If `tstop` equals the initial simulation time, the initial forces are computed and, if so desired, output is made for this time only.

`stopfile`, if given, the code checks after each full time step whether a file of the name given with `stopfile` exists. If it does, the simulation is stopped immediately.
This mechanism allows a controlled end of a simulation: a analyzing tool, which obtains the simulation data via the pipe from `gyrfalcON` may find that a pre-set condition for ending the simulation is satisfied and create a `stopfile`.

`Nlev` sets the number of time-step levels. If `Nlev=1` (default), a leap-frog integrator with constant global time step $\tau_{\min}$ is used.
Otherwise, if `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 as controlled the the parameters `fac`, `fph`, `fpa`, and `fea`.

`fac`, `fph`, `fpa`, `fea` control the average time step of a body to be

$$\tau = \min\left\{ \frac{\texttt{fac}}{|\boldsymbol{a}|},\ \frac{\texttt{fph}}{|\Phi|},\ \texttt{fpa}\frac{\sqrt{|\Phi|}}{|\boldsymbol{a}|},\ \texttt{fea}\sqrt{\frac{\epsilon}{|\boldsymbol{a}|}} \right\}, \qquad (2)$$

where Φ, $\boldsymbol{a}$, and ϵ are the gravitational potential and acceleration and the softening length. The parameters `fac`, `fph`, `fpa`, and `fea` determine the stepping. If either of them is zero, it is ignored.
In order to make a sensible choice for the parameters `hmin`, `Nlev`, `fac`, `fph`, `fpa`, and `fea`, use the following method. (i) 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. (ii) 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). (iii) Do some tests with varying `fac`, `fph`, `fpa`, and `fea` (set `tstop=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.
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.

10.1.3 Parameters Controlling Gravity

`theta` gives the tolerance parameter for `falcON`. It is recommended to stick to the default value, but see §8.

`Ncrit` gives the minimal number of bodies required before a tree cell is splitted into several cells. This parameter controls the speed with which the tree is build. The default is chosen to yield best performance for a single time step leap-frog scheme. If you are using individual adaptive time steps (see below), you may consider to increase `Ncrit` somewhat. This has little effect on the forces (actually makes them slightly more accurate) but may reduce the CPU time consumption (only with individual adaptive time steps).

`eps` gives the softening length, see also §7.
If `eps` <0, it is assumed that the bodies have individual softening lengths, which are given with the initial conditions. The individual softening lengths will be kept fixed at their initial values throughout the simulation. With this method, you can have larger softening length for more massive bodies, in order to have the same maximum force (requiring $\epsilon_i^2 \propto m_i$).
I have no experience with this new feature of the code, so be careful and watch out for possible bugs.

`kernel` specifies the softening kernel, see also §7.

`hmin` determines the smallest time step as $\tau_{\min} = 2^{-\texttt{hmin}}$.

`Grav` specifies the numerical value of Newton's constant of gravity.

`hgrow` with this option you can suppress the re-growing of the tree every (smallest) time step. Instead, the tree is grown only every $2^{\texttt{hgrow}}$ steps and re-used otherwise.
Note, however that re-using the tree violates time symmetry. I have not much experience with this option and recommend not to use it, unless you want to validate it first.

potname, if given, an external gravitational potential with that name is used. The code searches for a shared object file and loads it dynamically.
potpars, if potname is given, the external potential is using the parameter list provided with potpars, if any.
potfile, if potname is given, the external potential is using the file potfile, if any.

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.

10.2 An Example

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

```
mkplummer - 100000 seed=1 scale=1 | gyrfalcON - plum.snp tstop=10 eps=0.1
```

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 - . tstop=10 eps=0.1 logfile=plum.log VERSION=1.7.2I.gcc-3.3"
#
# run at  Tue Oct  7 19:45:01 2003
#     by  "dehnen"
#     on  "nevis"
#     pid  27011
#
#    time      energy       -T/U     |L|       |v_cm|  build   force    step     accum
# ------------------------------------------------------------------------------------
 0          -0.1469416    0.50366 0.0010751  6.3e-09  0.13   0.87         1         1
 0.015625   -0.1469415    0.50367 0.0010751  6.2e-09  0.09   0.84      0.93      1.95
 0.03125    -0.1469419    0.50366 0.0010751  6.2e-09  0.08   0.83      0.93      2.89
 0.046875   -0.146942     0.50366 0.0010751  6.2e-09  0.08   0.86      0.95      3.85
 .
 .
 .
 9.9531     -0.1469412    0.49914 0.0010775  6.5e-09  0.08   0.83      0.93   600.569
 9.9688     -0.1469415    0.49915 0.0010776  6.5e-09  0.08   0.84      0.93   601.499
 9.9844     -0.1469412    0.49915 0.0010776  6.6e-09  0.08   0.83      0.92   602.429
 10         -0.1469408    0.49917 0.0010776  6.6e-09  0.08   0.84      0.92   603.359
```

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.

11 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.

12 Bugs and Features

12.1 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`.

12.2 Bodies at Identical Positions

The code cannot cope 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.

12.3 Unknown Bugs

A bug that lead `falcON` or `gyrfalcON` to occasionally crash with ‘Segmentation fault’ I have recently tracked down and debugged (as of 3rd April 2003). However, there seems still to be a similar bug around, which is not reproducible and hence hard to track down and weed out. Measures have been taken to solve this problem eventually. If you ever encounter a problem that you think might be a bug and which is not mentioned in this documentation, please report it to me (`walter.dehnen@astro.le.ac.uk`). Thanks.

13 References

Dehnen, W., 2000, ApJ, 536, L39
Dehnen, W., 2001, MNRAS, 324, 273
Dehnen, W., 2002, JCP, 179, 27