GadJet.jl

Release 0.1.0

Aug 23, 2020

Contents:

CHAPTER 1

Installation

Installing is as easy as usually with Julia:

```
] add GadJet
```

If you want the latest version check out the Development branch

```
] add GadJet#Development
```

CHAPTER 2

Quickstart

2.1 Reading Data

If you want to read a simulation snapshot into memory with GadJet.jl, it's as easy as this:

```
data = read_snap(filename)
```

This will return a dictionary with the header information in `data["Header"]` and the blocks sorted by particle type.

As an example, this is how you would access the positions of the gas particles:

```
data["Parttype0"]["POS"]
```

If you only want to read a specific block for a single particle type, e.g. positions of gas particles you can use the function with a specified blockname and particle type like so:

```
pos = read_snap(filename, "POS", 0)
```

This will return an array of the datatype of your simulation, usually Float32.

2.2 Quick Visualisation

For a quick glimpse at your data you can use the glimpse function (yes, I thought hard about this one…)

```
image = glimpse(filename)
```

This will return a 500x500 pixel image of the whole box, centered on the center of mass.

If you want to look at a specific range you can provide an array with the center coordinates as `center_pos = [x, y, z]` and the extent in x, y and z direction with `dx, dy, dz`.

```
    image = glimpse(filename, center_pos, dx, dy, dz)
```

CHAPTER 3

Read Snapshot Data

3.1 Reading the header

Reading the header block of the simulation can be done by using:

```
h = read_header(filename::String)
```

Where `h` is the returned header object:

```
mutable struct Header
    npart::Vector{Int32}                # an array of particle numbers per type in␣
↪this snapshot
    massarr::Vector{Float64}            # an array of particle masses per type in␣
↪this snapshot - if zero: MASS block present
    time::Float64                       # time / scale factor of the simulation
    z::Float64                          # redshift of the simulation
    flag_sfr::Int32                     # 1 if simulation was run with star formation,
↪ else 0
    flag_feedback::Int32                # 1 if simulation was run with stellar␣
↪feedback, else 0
    nall::Vector{UInt32}                # total number of particles in the simulation
    flag_cooling::Int32                 # 1 if simulation was run with cooling, else 0
    num_files::Int32                    # number of snapshots over which the␣
↪simulation is distributed
    boxsize::Float64                    # total size of the simulation box
    omega_0::Float64                    # Omega matter
    omega_l::Float64                    # Omega dark enery
    h0::Float64                         # little h
    flag_stellarage::Int32              # 1 if simulation was run with stellar age,␣
↪else 0
    flag_metals::Int32                  # 1 if simulation was run with metals, else 0
    npartTotalHighWord::Vector{UInt32}  # weird
    flag_entropy_instead_u::Int32       # 1 if snapshot U field contains entropy␣
↪instead of internal energy, else 0
```

(continues on next page)

(continued from previous page)

```
    flag_doubleprecision::Int32         # 1 if snapshot is in double precision, else 0
    flag_ic_info::Int32
    lpt_scalingfactor::Float32
    fill::Vector{Int32}                 # the HEAD block needs to be filled with␣
↪zeros to have a size of 256 bytes
end
```

This is equivalent to:

```
h = head_to_obj(filename::String)
```

If you want to read the header information into a dictionary you can use:

```
h = head_to_dict(filename::String)
```

3.2 Reading a snapshot

3.2.1 Full snapshot

If you want to read a simulation snapshot into memory with GadJet.jl, it's as easy as this:

```
    data = read_snap(filename)
```

This will return a dictionary with the header information in `data["Header"]` and the blocks sorted by particle type.

As an example, this is how you would access the positions of the gas particles:

```
    data["Parttype0"]["POS"]
```

3.2.2 Specific blocks

Reading specific blocks only works with Format 2 at the moment.

If you only want to read a specific block for a single particle type, e.g. positions of gas particles, you can use the function with a specified blockname and particle type like so:

```
    pos = read_snap(filename, "POS", 0)
```

This will return an array of the datatype of your simulation, usually Float32.

If the snapshot has no info block this will fail unfortunately.

You can still read the specific block by supplying a hand-constructed `Info_Line` object:

```
mutable struct Info_Line
    block_name::String              # name of the data block, e.g. "POS"
    data_type::DataType             # datatype of the block, e.g. Float32 for single␣
↪precision, Float64 for double
    n_dim::Int32                    # number of dimensions of the block, usually 1 or␣
↪3
    is_present::Vector{Int32}       # array of flags for which particle type this␣
↪block is present,
```

(continues on next page)

(continued from previous page)

```
                                    # e.g. gas only:  [ 1, 0, 0, 0, 0, 0 ]
                                    # e.g. gas + BHs: [ 1, 0, 0, 0, 0, 1 ]
end
```

and passing that to the function `read_block_by_name`:

```
    pos = read_block_by_name(filename, "POS", info=pos_info, parttype=0)
```

where `pos_info` is a `Info_Line` object.

`read_snap` is used mainly as a wrapper function to call `read_block_by_name`, in case you were wondering about the function name change.

I will collect some example `Info_Line` objects in a later release to be able to read some common blocks even without a info block.

3.2.3 Getting snapshot infos

If you have a Format 2 snapshot and just want to know what blocks the snapshot contains you can use the function

```
    print_blocks(filename)
```

to get an output of all block names.

If your simulation contains an INFO block you can read the info lines into `Info_Line` object like so:

```
    info = read_info(filename, verbose=true)
```

This will return an Array of `Info_Line` objects. The optional keyword `verbose` additionally gives the same functionality as `print_blocks` and prints the names to the console.

3.3 Large Simulations

For large simulations Gadget distributes snapshots over multiple files. These files contain particles associated with specific Peano-Hilbert keys.

To get all particles within a subvolume of the simulation you can use the functions `read_particles_in_box(...)` or `read_particles_in_volume(...)`.

`read_particles_in_box(...)` takes a box defined by a lower-left corner and an upper-right corner, constructs the peano hilbert keys, selects the relevant files and reads the particles from these files into a dictionary.

```
function read_particles_in_box(filename::String, blocks::Vector{String},
                               corner_lowerleft,
                               corner_upperright;
                               parttype::Int=0,
                               verbose::Bool=true)

            (...)

end
```

You can define an array of blocks you want to read, these will be read in parallel with simple multi-threading.

`read_particles_in_volume(...)` is a simple wrapper around `read_particles_in_box(...)`, where you can define a central position and a radius around it and it will construct the box containing that sphere for you and read all particles in it.

```
function read_particles_in_volume(filename::String, blocks::Vector{String},
                                  center_pos::Vector{AbstractFloat},
                                  radius::AbstractFloat;
                                  parttype::Int=0,
                                  verbose::Bool=true)

            (...)
end
```

In both functions `parttype` defines the particle type to be read, as in the previous read functions and `verbose` gives console output.

3.3.1 Filename

With the snapshots being distributed over multiple filenames you need to be careful with that keyword. In this case filename refers to the base-name. Assuming you want to read snapshot 140, which is in the snapshot directory 140 the filename is

```
filename = "path/to/your/snapshot/directories/snapdir_140/snap_140"
```

GadJet will then automatically loop through the sub-snapshots which end in ".0", ".1", ... , ".N".

3.3.2 Example

If you want to, e.g. read positions, velocities, masses, density and hsml for all gas particles within the virial radius of the most massive halo of a simulation you can do this as follows.

Assuming `pos_halo` is the position of the center of mass of the halo and `r_vir` is its virial radius you read the data with

```
blocks = ["POS", "VEL", "MASS", "RHO", "HSML"]

data   = read_particles_in_volume(filename, blocks, pos_halo, r_vir,
                                  parttype=0,
                                  verbose=true)
```

This will return a dictionary with the blocks as keys and containing the arrays for the particles.

```
data["POS"]  # array of positions
data["RHO"]  # array of densities
(...)
```

CHAPTER 4

Read Subfind Data

4.1 Reading the header

Similarly to the normal snapshot you can read the header of the subfind output into a `SubfindHeader` object

```
struct SubfindHeader
    nhalos::Int32                       # number of halos in the output file
    nsubhalos::Int32                    # number of subhalos in the output file
    nfof::Int32                         # number of particles in the FoF
    ngroups::Int32                      # number of large groups in the output file
    time::Float64                       # time / scale factor of the simulation
    z::Float64                          # redshift of the simulation
    tothalos::UInt32                    # total number of halos over all output files
    totsubhalos::UInt32                 # total number of subhalos over all output␣
↪files
    totfof::UInt32                      # total number of particles in the FoF
    totgroups::UInt32                   # total number of large groups over all␣
↪output files
    num_colors::Int32                   # number of colors
    boxsize::Float64                    # total size of the simulation box
    omega_0::Float64                    # Omega matter
    omega_l::Float64                    # Omega dark enery
    h0::Float64                         # little h
    flag_doubleprecision::Int32         # 1 if snapshot is in double precision, else 0
    flag_ic_info::Int32
end
```

using

```
h = read_subfind_header(filename::String)
```

4.2 Reading the subfind files

If you compiled Gadget with `WRITE_SUB_IN_SNAP_FORMAT` you can read the subfind output like you would a normal snapshot, with any of the above methods. For convenience you can also use a helper function provided by GadJet. Since each of the blocks is only relevant for either halos, subhalos, Fof or large groups you don't need to define a particly type, aka halo type in this case.

So in order to read the virial radius of the halos in a file you can simply use

```
R_vir = read_subfind(filename, "RVIR")
```

CHAPTER 5

Write Data

GadJet.jl can write snapshots that can be used as initial conditions.

5.1 Format 2

The safest way to write snapshots is in Format 2. Simply set up your header object and the arrays you want to write in the correct data format. For the header this is the struct `Header` and for data its usually `Array{Float32,2}`. You can then write an initial condition file by writing the header and the individual data blocks.

```
write_header(filename, header)
write_block(filename,  pos, "POS")
write_block(filename,  vel, "VEL")
write_block(filename,  id,  "ID")
```

Please note that you have to combine the arrays for individual particles in the correct order.

5.2 Format 1

Writing in format 1 works the same as above, but you need different function values. Also you need to make sure the blocks are in the order gadget expects them to be!

```
write_header(filename, header, snap_format=1)
write_block(filename,  pos,    snap_format=1)
write_block(filename,  vel,    snap_format=1)
write_block(filename,  id,     snap_format=1)
```

CHAPTER 6

Unit Conversion

GadJet.jl now uses Unitful.jl and UnitfulAstro.jl to store the unit conversion factors with actual units in place. You can convert the internal units of Gadget into cgs units by defining the object `GadgetPhysicalUnits`:

```
GU = GadgetPhysicalUnits(l_unit::Float64=3.085678e21, m_unit::Float64=1.989e43, v_
↪unit::Float64=1.e5;
                         a_scale::Float64=1.0, hpar::Float64=1.0,
                         γ_th::Float64=5.0/3.0, γ_CR::Float64=4.0/3.0, xH::Float64=0.
↪76)
```

where the keyword arguments are:

- `a_scale::Float64 = 1.0`: Cosmological scale factor of the simulation. Can be passed with the header `h` as `h.time`.
- `hpar::Float64 = 1.0`: Hubble constant as 'little h'. Can be passed with header `h` as `h.h0`.
- `γ_th::Float64 = 5.0/3.0`: Adiabatic index of gas.
- `γ_CR::Float64 = 4.0/3.0`: Adiabatic index of cosmic ray component.
- `xH::Float64 = 0.76`: Hydrogen fraction of the simulation, if run without chemical model.

This returns an object of type `GadgetPhysicalUnits` with the following properties:

```
struct GadgetPhysicalUnits

    x_cgs::typeof(1.0u"cm")          # position in cm
    v_cgs::typeof(1.0u"cm/s")        # velocity in cm/s
    m_cgs::typeof(1.0u"g")           # mass in g

    t_s::typeof(1.0u"s")             # time in sec
    t_Myr::typeof(1.0u"Myr")         # time in Myr

    E_cgs::typeof(1.0u"erg")         # energy in erg
    E_eV::typeof(1.0u"eV")           # energy in eV
```

(continues on next page)

(continued from previous page)

```
    B_cgs::typeof(1.0u"Gs")          # magnetic field in Gauss

    rho_cgs::typeof(1.0u"g/cm^3")    # density in g/cm^3
    rho_ncm3::typeof(1.0u"n_e")      # density in N_p/cm^3

    T_K::typeof(1.0u"K")             # temperature in K

    P_th_cgs::typeof(1.0u"Ba")       # thermal pressure in Ba
    P_CR_cgs::typeof(1.0u"Ba")       # cosmic ray pressure in Ba

end
```

To convert, say positions of gas particles from a cosmological simulation to physical units you can use:

```
h      = read_header(filename)

pos    = read_snap(filename, "POS", 0)

GU     = GadgetPhysicalUnits(a_scale=h.time, hpar=h.h0)

pos .*= GU.x_cgs
```

If you have different units than the standard Gadget ones you can call the object cunstructor with different values

```
GU = GadgetPhysicalUnits(your_l_unit, your_m_unit, your_v_unit; kwargs...)
```

Converting the units can then be done with Unitful.jl and UnitfulAstro.jl. So if you want to convert the position units from the default `cm` to `Mpc` you can do this as:

```
using Unitful
using UnitfulAstro

pos = read_snap(filename, "POS", 0)
pos = @. pos * GU.x_cgs |> u"Mpc"
```

If you want to get rid of the units, for example if you need basic datatypes again for a function you can use the funtion `ustrip`:

```
pos = ustrip(pos)
```

6.1 Primitive unit type

If you want to have the same functionality, but without using `Unitful.jl` you can construct a similar object:

```
GU = GadgetPhysical(l_unit::Float64=3.085678e21, m_unit::Float64=1.989e43, v_
↪unit::Float64=1.e5;
                    a_scale::Float64=1.0, hpar::Float64=1.0,
                    γ_th::Float64=5.0/3.0, γ_CR::Float64=4.0/3.0, xH::Float64=0.76)
```

This uses the same conversions, but leaves out the actual unit strings.

CHAPTER 7

Riemann Solvers

GadJet.jl provides a number of exact riemann solvers. So far these are for

- Sod shock, pure hydro
- Sod shock, with CR acceleration

7.1 Setup

To get the exact solution to a Sod shock you first need to set up the inital conditions. You can do this with the helper function `RiemannParameters` that contains all parameters for all possible configurations:

```
RiemannParameters(;rhol::Float64=1.0, rhor::Float64=0.125,       # density left and␣
↪right (L&R)
                   Pl::Float64=0.0,   Pr::Float64=0.0,           # pressure L&R
                   Ul::Float64=0.0,   Ur::Float64=0.0,           # internal energy L&R
                   P_cr_l::Float64=0.0, P_cr_r::Float64=0.0,     # CR pressure L&R
                   E_cr_l::Float64=0.0, E_cr_r::Float64=0.0,     # CR energy L&R
                   Bl::Array{Float64,1} = zeros(3),              # B-field left
                   Br::Array{Float64,1} = zeros(3),              # B-field right
                   Mach::Float64=0.0,                            # target Mach number
                   t::Float64,                                   # time of the solution
                   x_contact::Float64=70.0,                      # position of the␣
↪contact discontinuity along the tube
                   γ_th::Float64=5.0/3.0,                        # adiabatic index of␣
↪the gas
                   γ_cr::Float64=4.0/3.0,                        # adiabatic index of␣
↪CRs
                   Pe_ratio::Float64=0.01,                       # ratio of proton to␣
↪electron energy in acceleration
                   thetaB::Float64=0.0,                          # angle between␣
↪magnetic field and shock normal
                   theta_crit::Float64=(π/4.0),                  # critical angle for␣
↪B/Shock angle efficiency
```

(continues on next page)

(continued from previous page)

```
                    dsa_model::Int64=-1,                    # diffuse shock␣
↪acceleration model
                    xs_first_guess::Float64=4.7)            # first guess of the␣
↪resulting shock compression
```

To set up a standard Sod shock you need to supply it with pressure/energy values for left and right state, or with pressure/energy values for the left state and a target Mach number.

A minimal working version would be, for a shock with Mach 10, at time = 1.5:

```
par = RiemannParameters(Ul=100.0, Mach=10.0, t=1.5)
```

This returns a parameter object for a pure hydro Sod shock:

```
mutable struct SodParameters

    rhol::Float64           # denisty left
    rhor::Float64           # density right
    Pl::Float64             # pressure left
    Pr::Float64             # pressure right
    Ul::Float64             # internal energy left
    Ur::Float64             # internal energy right
    cl::Float64             # soundspeed left
    cr::Float64             # soundspeed right
    M::Float64              # Mach number
    t::Float64              # time
    x_contact::Float64      # position of the contact discontinuity along the tube
    γ_th::Float64           # adiabatic index of the gas
    γ_exp::Float64          # helper variable
    η2::Float64             # helper variable

end
```

A minimal working version for the solution of the CR shock discussed in Pfrommer+16 (doi:10.1093/mnras/stw2941) would be:

```
par = RiemannParameters(Pl=63.499, Pr=0.1, t=1.5, dsa_model=4)
```

This also returns a parameter object: `SodCRParameters_noCRs` which can be found in cr_sod_shock_noprepopulation.jl .

7.2 Solving the shock

To solve the shock with the given initial condition you just need to call

```
sol = solve(x, par)
```

with par being either of the above mentioned parameter objects, multiple dispatch will take care of the rest.

`x` has to be an array with either sample positions along the tube, or your actual particle positions, to make calculating errors easier. You can also just pass it an array with a single position, if you're only interested in that specific part of the shock (e.g. `x = [86.0]` for the center of the postshock region.)

This will return a solution object depending on which shock you're solving.

For the pure hydro case this is:

```
mutable struct SodHydroSolution
    x::Array{Float64,1}          # array of given positions
    rho::Array{Float64,1}        # array of densities along the tube
    rho4::Float64                # density in postshock region
    rho3::Float64                # density between contact disc. and rarefaction wave
    P::Array{Float64,1}          # array of pressures along the tube
    P34::Float64                 # pressure between shock and rarefaction wave
    U::Array{Float64,1}          # array of internal energies along the tube
    v::Array{Float64,1}          # array of velocities along the tube
    v34::Float64                 # velocity between shock and rarefaction wave
    vt::Float64                  # velocity of rarefaction wave
    vs::Float64                  # shock velocity
    Mach::Float64                # Mach number
end
```

7.3 Utility

A common issue is running into the error DomainError when solving a CR Sod shock. This is due to the definition of the incomplete beta function. You can avoid this by supplying a value for `xs_first_guess`, which is a first guess for the value of the shock compression ratio. In case you don't know the target `xs` (which is the usual case) and are tired of trying different values there's a helper function for that:

```
function find_xs_first_guess(Ul::Float64, Mach::Float64;
                             xs_start::Float64=3.8, delta_xs::Float64=1.e-4,
                             eff_model::Int64=2, thetaB::Float64=0.0)

    [...]
end
```

CHAPTER 8

SPH mapping

8.1 Internal Module

You can map SPH data to a grid using the function:

```
function sphMapping(Pos, HSML, M, ρ, Bin_Quant;
                    param::mappingParameters,
                    kernel::SPHKernel,
                    show_progress::Bool=true,
                    conserve_quantities::Bool=false,
                                        parallel::Bool=true,
                    dimensions::Int=2)

        [...]

end
```

8.1.1 Setup

To map the data you need to define the mapping parameters via the `mappingParameters` object:

```
par = mappingParameters(xlim=[xmin, xmax], ylim=[ymin, ymax], zlim=[zmin, zmax],
                        Npixels=200)
```

Instead of Npixels you can also give the keyword argument `pixelSideLength` if you prefer to define your image that way.

You also need to choose the kernel you used in the simulation. I implemented the following ones:

```
k = Cubic()
k = Quintic()
k = WendlandC4()
k = WendlandC6()
```

8.1.2 Mapping

With the setup done you can now map (e.g.) density of your data using the function above as:

```
image = sphMapping(x, hsml, m, rho, rho, param=par, kernel=k)
```

Replacing the second `rho` with any other quantity would map that quantity of course. Please note: This function doesn't do any unit conversion for you, so you need to convert to the desired units beforehand. See the chapter on unit conversion for usage.

Image now contains a 2D array with the binned data and can easily be plotted with `imshow()` from any plotting package of your choosing.

Per default the keyword `parallel = true` causes the run to use multiple processors. For this you need to start julia with `julia -p <N>` where `<N>` is the number of processors in your machine.

8.1.3 Conserved quantities

With the latest release you can map the particles to a grid while also conserving the particle volume, following the algorithm described in Dolag et. al. 2006 (https://ui.adsabs.harvard.edu/link_gateway/2005MNRAS.363...29D/doi:10.1111/j.1365-2966.2005.09452.x).

This is switched off by default, but is slightly more expensive than simple mapping. If you don't want to use it simply call the mapping function with `conserve_quantities=false`.

CAUTION: This is currently broken and under development!

8.2 External Programs

GadJet.jl provides helper function for two external sph mapping Codes: Smac and P-Smac2.

8.2.1 P-Smac2

P-Smac2 by Julius Donnert (https://github.com/jdonnert/Smac2) is an advanced mapping code for a multitude of different quantities. To run a mapping and plotting loop from a Julia script you need to update the parameter files on the fly. The function `write_smac2_par` provides this functionality.

```
write_smac2_par(x, y, z,
                euler_angle_0, euler_angle_1, euler_angle_2,
                xy_size, z_depth, xy_pix::Int64,
                input_file, output_file, path,
                effect_module::Int64=0, effect_flag::Int64=0)
```

8.2.2 Smac

Smac is a SPH mapping Code by Klaus Dolag and others. The implementation is described in Dolag et al. 2005 (https://ui.adsabs.harvard.edu/link_gateway/2005MNRAS.363...29D/doi:10.1111/j.1365-2966.2005.09452.x)

Smac isn't public unfortunately. So these functions are mainly for my personal use. If you do have access to Smac, here's a reference to what you can do.

GadJet.jl provides some functions to read the binary output of Smac, as I personally prefer that over the FITS output. To get the binary format you need to set `FILE_FORMAT = 1` in the parameter file.

Reading image information

If you set `FILE_HEADER = 1` in the Smac parameter file you can read the information of the image header into a `Smac1ImageInfo` object like so:

```
info = read_smac1_binary_info(filename)
```

The `Smac1ImageInfo` object contains the following information

```
struct Smac1ImageInfo

    snap::Int32                 # number of input snapshot
    z::Float32                  # redshift of snapshot
    m_vir::Float32              # virial mass of halo
    r_vir::Float32              # virial radius of halo
    xcm::Float32                # x coordinate of image center
    ycm::Float32                # y coordinate of image center
    zcm::Float32                # z coordinate of image center
    z_slice_kpc::Float32        # depth of the image in kpc
    boxsize_kpc::Float32        # xy-size of the image in kpc
    boxsize_pix::Float32        # xy-size of the image in pixels
    pixsize_kpc::Float32        # size of one pixel in kpc
    xlim::Array{Float64,1}      # x limits of image
    ylim::Array{Float64,1}      # y limits of image
    zlim::Array{Float64,1}      # z limits of image
    units::String               # unitstring of image

end
```

Reading the image

The image itself can be read with

```
image = read_smac1_binary_image(filename)
```

This will return an `Array{Float32,2}` with the pixel values. You can pass this to any imshow function of your favorite plotting package.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search