

cgmdode

A C library for coarse-grained macromolecular dynamics with Open Dynamics Engine

User Manual

version 0.3

Bertrand Caré
bertrand.care@gmail.com

<http://bitbucket.org/bcare/cgmdode-hg>

Table of Contents

I - Introduction.....	3
I.1 - Origins of cgmdode.....	3
I.2 - What is cgmdode's purpose ?.....	3
I.3 - Overview of cgmdode design.....	3
I.4 - Should I use cgmdode for my simulation ?.....	4
I.5 - License.....	5
II - Installation.....	6
II.1 - Requirements.....	6
II.2 - Installation from sources.....	6
III - Building a simulation.....	7
III.1 - Importing cgmdode into your code.....	7
III.2 - cgmdode initialization sequence.....	7
III.3 - Creating physical objects.....	9
III.4 - Assigning interaction sites to physical objects.....	10
III.5 - Assigning physical objects to thermostats.....	12
III.6 - Linking physical objects using mechanical joints.....	15
III.7 - Running the simulation.....	18
III.8 - Cleaning up.....	18
IV - API Reference.....	19
IV.1 - Basic data types.....	19
IV.2 - Simparameters functions.....	19
IV.3 - Simcontext functions.....	22
IV.4 - Geometry data functions.....	22
IV.5 - Physobject functions.....	24
IV.6 - Interaction class functions.....	27
IV.7 - Thermostat class functions.....	28
IV.8 - Articulation functions.....	29
IV.9 - Additionnal features.....	30

I - Introduction

I.1 - Origins of cgmdode

The development of cgmdode started in the Theoretical Physics of Condensed Matter lab in Paris, by the Multiscale Modelling of Living Matter group. It was initially intended as a way to facilitate reusability and centralize the different coarse-grained simulation codes produced by the group. The library stemmed from the simulations of single molecule experiments with DNA and optic/magnetic tweezers, and the simulations of biological macromolecular assemblies such as microtubules and capsids. Among the models and simulation tools for this type of systems, there is "scale gap" : very performant tools exist for the microscopic scale, mainly full-atom, molecular dynamics software, but they consume a lot of computing resources. At the macroscopic scale, the physical properties of such systems is generally abstracted away in mean-field analytical models, whereas these properties might be crucial. The coarse-graining approach is aimed at filling this gap with a mesoscopic scale. At this scale, the behavior of large biological systems composed of thousands of dynamically self-assembling large subunits can be somewhat efficiently simulated. cgmdode is aimed at facilitating the development of simulations of such models.

The other reason for cgmdode's creation is that the coarse-grained simulations of biological systems developed by the M3V group typically outsourced the physics to the rigid body dynamics engine ODE. The consequence was that there was a lot of boilerplate code required to implement the simulation softwares that needed to be rewritten for every new biological system simulated. So I (Bertrand Caré) decided to write a library that does this once and for all that was really simple to use, and hopefully fast.

I.2 - Coarse-graining using a rigid body dynamics engine

cgmdode, or Coarse-Grained Macromolecular Dynamics with Open Dynamics Engine, is a C library aimed at providing data structures and functions for simulating populations of macromolecules. In cgmdode, a macromolecule is represented as a rigid body with an arbitrary geometry and a distribution of interaction sites that attract/repulse sites of neighbouring macromolecules. cgmdode also allow the user to define statistical physics thermostats : they determine how the macromolecules positions fluctuate. In other words, cgmdode is a coarse-grained molecular dynamics engine.

The fundamental principle behind cgmdode is that in order to build a coarse-grain model of a macromolecule, the distribution of mass and the distribution of interacting sites can be decoupled and treated separately (this principle is developed in the Concepts section). In addition, the general process of building a coarse-grained model can also be separated between what is model-dependent and what is model-independent (this is also developed in the Concepts section). For instance, the shape, mass, and distributions of interaction sites of your macromolecule all depend on the model you came up with, on what you decided to include in your coarse-grained model or not. Similarly, the nature of the interaction potentials between the interaction sites you defined are also model-dependent. But there are parts of your simulation that will always follow the same routine, regardless of the interaction potentials you chose or the positions of the interaction sites in your macromolecule : you have to compute the force applied by one interaction site on each other site that it interacts with, you have to integrate the laws of motion, compute collisions, and so on. cgmdode also allows the user to define mechanical joints between rigid bodies, which was originally needed for models DNA developed by the M3V group.

So what is the aim of cgmdode ? the aim of cgmdode is **not** to provide a collection of built-in interaction potentials and thermostats (which are model-dependent), but rather to take care of everything that is model-independent. Basically, you tell cgmdode the shape, mass, distribution and nature of interaction sites of your macromolecules, the interaction potentials, the kind of thermostat you need -- but you have to write all of these components yourself ! -- and cgmdode takes care of what is common to every coarse-grained simulation : calculating forces and integrating the law of motions according to the components you wrote.

I.3 - Overview of cgmdode design

cgmdode is essentially a wrapper for Open Dynamics Engine with a builtin simulation loop that includes a generic interface for long-distance interaction computations and statistical mechanics thermostats. The following figure compares the generic design of a simulation where a client program uses a physics engine (like ODE) with a simulation where the client program uses cgmdode.

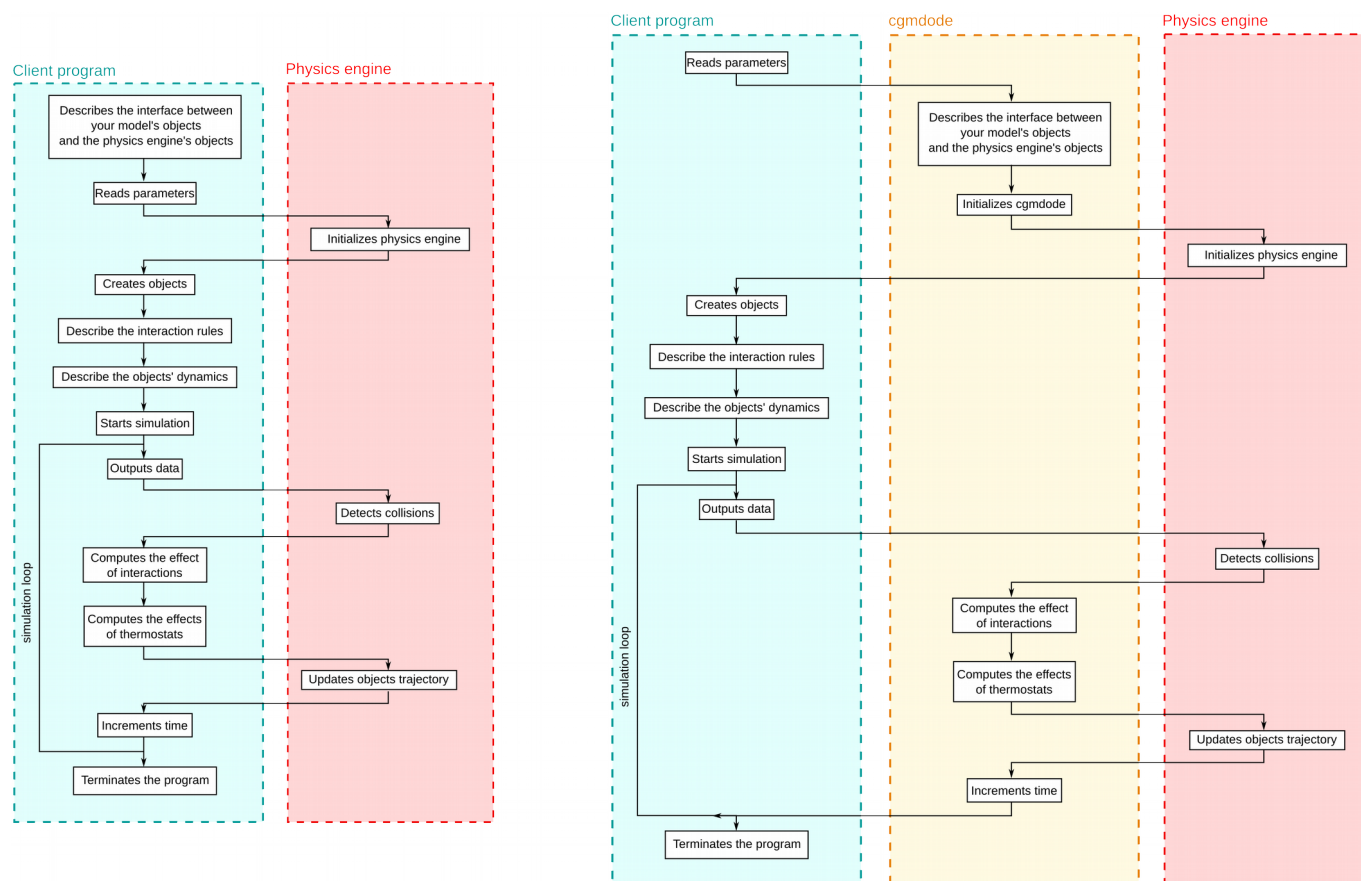


Fig. 1 - (left) an example of simulation where a client program uses a physics engine to model a system. (right) the same simulation when using cgmdode as the interface with the physics engine.

I.4 - Should I use cgmdode ?

Numerous excellent coarse-grained molecular dynamics engines already exist (ESPResSO MD, OpenMD, LAMMPS, DLPOLY ...) and may also fit your needs if you are interested in using cgmdode. The answer to whether or not you should use cgmdode depends on several considerations.

- **What kind of geometrical shapes do you need to simulate ?**

If you just need individual spheres, then cgmdode is probably not for you (but it might !). One particularity of cgmdode is that objects can be of any shape (including trimeshes). However if you need cylinders, boxes, or complex solid geometrical shapes, then cgmdode might suit your needs.

- **Do you need articulated joints and holonomic constraints ? How do you want collisions between rigid bodies to be treated ?**

In ODE, and therefore in cgmdode, collisions are treated as mechanical constraints and solved at each time step to find a separating force between two colliding objects. If you just need hard sphere potentials to recreate excluded volume you can have them in cgmdode if you write them yourself, but other libraries might have them built-in and might be easier to use. However, in cgmdode, built-in mechanical joints are available. So you can create ball-in-socket, universal, hinge, and many other joints very easily, which might not be present in classical coarse-graining libraries.

- **Do you need full-atom models ?**

If yes, then cgmdode is probably not made for you. The core principle of cgmdode is that the mass distribution (i.e. distributions of atoms) of a molecule and its interaction sites (e.g. distribution of charges) can be decoupled and coarse-grained separately without losing (too much) physical relevance. In full-atoms simulations, both the distribution of atoms and charges (and other physico-chemical properties) are tightly coupled. We recommend other tools with already-implemented force fields, potentials, and standard thermostats, as none are shipped with cgmdode.

- **Do you need a ready-to-use executable or do you need to an API ?**

cgmdode does not provide scripting capabilities nor any out-of-the-box executable that produces output : it's a library. You have to write your own program and load your own parameters, using functions and data structures provided by cgmdode and linking against it. cgmdode just makes the process of writing such a program easier.

- **Do you need an additional layer of complexity above the physics of your system ?**

If yes, you might be interested in cgmdode. It was originally made for simulating biological macromolecular systems, that is, living systems. The API was therefore written with the idea in mind that cgmdode should provide the user with the freedom to implement complex interaction rules. These rules can even possibly change throughout the evolution of the simulated system. This is why cgmdode gives you a generic interface for writing thermostats and interaction potentials, and no builtin for these objects. cgmdode might be particularly suitable for models where the evolution of the system does not solely depend on its initial conditions.

I.5 - License

cgmdode is licensed under the LGPL v3 (<https://www.gnu.org/copyleft/lesser.html>)

cgmdode uses Open Dynamics Engine (<http://ode-wiki.org>) which is released under the LGPL v2.

cgmdode uses SGLIB (<http://http://sglib.sourceforge.net/>).

cgmdode uses the GNU Scientific Library (<https://www.gnu.org/software/gsl/>) released under the GPL v3.

II - Installation

II.1 - Requirements

cgmdode is designed to run on a GNU/Linux-like distribution. To install it, you will need :

GNU Scientific Library ≥ 1.15

<http://www.gnu.org/software/gsl/>

ODE ≥ 0.13

<http://ode-wiki.org/>

GNU build system (Autotools) and pkg-config.

They are generally available as the packages autotools-dev and pkg-config for major GNU/Linux distributions.

II.2 - Installation from sources

You will also require a C compiler (we recommend gcc).

You can get the sources from the Bitbucket repository : <http://bitbucket.org/bcare/cgmdode-hg>

Or in your terminal console if you have mercurial :

```
$> hg clone https://bitbucket.org/bcare/cgmdode-hg
```

Then in the source folder :

```
$> ./autogen.sh
$> ./configure
$> make
$> make install
```

`./autogen.sh` needs to be run once, generally just after having fetched the sources. It creates the autotools files required for installation depending on your system.

If you chose to install cgmdode system-wide, you will need administrator rights for the `make install` command.

You can use `./configure --help` to see what compilation and installation options are available.

III - Building a simulation

This section takes you through the basic steps of building a cgmdode simulation. This simple tutorial was written so that people less familiar with physics engine libraries, statistical physics or simulation techniques should still understand how to use cgmdode. Therefore, we will not review here the advanced aspects of scientific rigid bodies simulations, statistical mechanics, or structural biology. This is a quick start tutorial that should be suitable for anyone with a significant grasp of basic geometry and the C language (for example undergrad students). The most advanced programming concept required for using cgmdode is function pointers.

What are the basic steps for creating a simulation with cgmdode ? First you will have to import cgmdode into your code and set the compilation options in order to link against cgmdode. Then you may begin writing your simulation program. In order to use cgmdode, your program will have to initialize cgmdode internal data. Next, you will have to create simulated objects, or "physical objects", decorate these objects with interaction sites, and attribute them to thermostats. Once these steps are done, you will be able to actually start the simulation loop and let cgmdode take care of the computations.

III.1 - Importing cgmdode into your code

Include cgmdode depending on your installation options. For a standard installation, your C program should include :

```
#include <cgmdode/cgmdode.h>
```

When compiling your program, you must specify to the compiler and the linker where to find cgmdode. For a standard installation and gcc, it should look like this in a terminal console :

```
$> gcc -I/installation_prefix/include myprogram.c -o myprogram.out  
-L/installation_prefix/lib -lcgmdode -lode -lgs1 -lgs1cblas -lm
```

installation_prefix is the prefix you provided when you called ./configure during cgmdode installation. If you did the standard installation (./configure ; make ; sudo make install) without changing the default options, the compilation command should be :

```
$> gcc myprogram.c -o myprogram.out -lcgmdode -lode -lgs1 -lgs1cblas -lm
```

Note that the options will also depend on where the GNU Scientific Library and ODE are installed on your system.

With the pkg-config command, you can get all the required compilation options :

```
$> pkg-config --cflags cgmdode # required include folders  
$> pkg-config --libs cgmdode # required libs
```

III.2 - cgmdode initialization sequence

In every program using cgmdode, you first need to execute a few steps in a precise order before actually creating physical objects and defining interactions and thermostats. cgmdode first has to initialize its internal objects and data according to the parameters you provide, then it has to initialize the physics engine, i.e. ODE. Simulation parameters such as the simulation timestep, the size of the world, and parameters of the physics engine are stored in a `simparameters` data structure. cgmdode uses a context design pattern for the interface between the user program and cgmdode internal data. This context is a `simcontext` data structure.

Every cgmdode simulation starts with creating a `simparameters` object and a `simcontext` object. The sequence is :

- Allocate memory for a `simcontext` and a `simparameters` data structure.
- Create the `simparameters` structure with `simparams_create`.
 - set the random number generator seed (see section IV.10.C).
 - set ODE-specific parameters (solver type, collision parameters, etc.)
 - set the simulated medium parameters (size, boundary conditions, etc.)
- Create the `simcontext` object and attach the `simparameters` object to the `simcontext` object using `simcontext_create`.

- Initialize the simulation context using `simcontext_init`.

Here is the code of a standard initialization sequence :

```

/*
   Allocate memory for a simparameters struct.
*/
simparameters * simparams = malloc(sizeof(*simparams));

/*
   Allocate memory for a simcontext struct.
*/
simcontext * simcon = malloc(sizeof(*simcon));

/*
   Initialize simparams :
*/
simparams_create(simparams);
simparams_set_seed(simparams, prng_seed);
simparams_set_ode_params(simparams,
    qstep_nbiters, /* nb of iterations for ODE quickstep solver */
    qstep_w,      /* relaxation parameter for the quickstep solver */
    erp_glob,     /* Error Reduction Parameter (ERP) */
    cfm_glob,     /* Constraint Force Mixing parameter (CFM) */
    ode_step,     /* simulation timestep */
    lin_damp,     /* linear velocity damping */
    ang_damp,     /* angular velocity damping */
    max_corrvel,  /* maximum collision correction velocity */
    max_avel,     /* maximum angular velocity */
    max_contacts, /* maximum collision contact points */
    contact_mode, /* collision parameters */
    contact_mu,   /* contact friction coefficient (tangent) */
    contact_mu2,  /* contact friction coefficient (normal) */
    contact_bounce, /* collision restitution parameter */
    contact_bounce_vel, /* minimum collision relative velocity threshold */
    stepper_type, /* solver type */
    min_object_size, /* minimum simulated object size */
    max_object_size); /* maximum simulated object size */

simparams_set_medium_params(simparams,
    boundary_type, /* boundary conditions */
    world_Lx,     /* world size along x axis */
    world_Ly,     /* world size along y axis */
    world_Lz,     /* world size along z axis */
    max_grid_size, /* maximum nb of cells in space discretization grid */
    min_cell_size); /* minimum size of cell in space discretization grid */

/*
   Initialize simcon :
*/

simcontext_create(simcon, simparams, some_user_defined_data_pointer);
simcontext_init(simcon);

/*
   That's it !
   You are all set, you can now define
   physical objects, interaction classes,
   thermostats and so on from here !

   -- your code below --
*/

```

This set of instructions is required for any program using `cgmdode`, and must be executed in this precise order (the variable values may vary of course). You may want to refer to the API reference (section IV) for detailed information about these functions' parameters, in particular for `simparams_set_ode_params`.

Please note that **you must not modify simparams once initialization has been done.**

An other important notice : parameters of physics simulations are generally scaled before being used in the program, so that their numerical values are comparable, and ideally around 1. It is up to the user to scale the physical quantities fed to cgmdode.

III.3 - Creating physical objects

Physical objects (`physobject`) are created by calling the function `physobject_create`. This function sets up the internal representation of the physical object in cgmdode and attaches data to the specified simulation context. A physical object is the combination of two things : a rigid body (i.e. a mass distribution attached to a local referential) and optionally a geometric shape determining the volume occupied by the physical object.

In cgmdode, a physical object can be either :

- Just a rigid body with no geometry or volume
(forces act on the object, but it can pass through other objects)
- Just a geometric shape with no mass distribution
(an immobile obstacle unaffected by external forces, but it collide with other objects)
- The combination of a rigid body (forces act on the object) and a geometric shape (it collides with other objects).

In either cases, before creating the physical object, you need to instantiate an intermediary data structure determining the geometry data of the object. Even if you don't plan on allowing collisions for the object, and even if the rigid body you want to create will be a punctual mass, the geometry data will be used by cgmdode to determine the mass distribution of your object. So **geometry data is always required.**

So how do you instantiate the geometry data ? cgmdode provides a set of functions to create basic geometric primitives plus the possibility to load triangular meshes (see API reference section IV.5).

For example, to create a box of size 10x5x2.5, you need to allocate memory for a `geometry_data` structure, then initialize the `geometry_data` object with the sizes of the box.

```
geometry_data* box_geomdata = malloc(sizeof(*box_geomdata));

geomdata_create_box(box_geomdata,
    10.0,      /* box size along the local referential x axis */
    5.0,      /* box size along the local referential y axis */
    2.5);     /* box size along the local referential z axis */
```

Then you need to allocate memory for the `physobject` that will represent your simulated box, and instantiate it using the geometry data object for the box.

```
physobject* pobj1 = malloc(sizeof(*pobj1));

physobject_create(pobj1,
    42,          /* the user-defined ID of the object */
    box_geomdata, /* the geometry_data of the object */
    0x0,         /* some pointer to user-defined additional data (here none) */
    simcon);     /* the simcontext object to which the object will be attached */

/*
    Note: You can create several instances of a physobject
    using the same geometry_data object.
    Once the physobjects are created, the original geometry_data object
    is no longer required (it has been copied internally).
    You have to free the geometry_data object you created
    (but internal copies created by cgmdode will be freed
    by cgmdode, you don't have to worry about it).
*/
free(box_geomdata); /* it was copied into pobj1 */
```

At this stage, the box was just attached to the simulation context, and its internal representation initialized. If you want your box to react to forces applied onto it and obey Newton's laws, you must initialize its dynamical properties (as a body e.g. a mass distribution). The following code initializes the inertia matrix of an object so that its total mass is equal to 13.0 mass units :

```
t_real box_mass = 13.0;
physobject_init_body(pobj1, box_mass, INIT_MASS_TOTAL);

/*
   If instead of the total mass of the body,
   you want to initialize the mass using the volume of the object
   and a density (so that total_mass = Volume x density), you may use :

t_real box_density = 1.0 ;
physobject_init_body(pobj1, box_density, INIT_MASS_DENSITY);

   The volume is automatically determined by cgmdode using the
   geometry_data you provided when creating the physobject.

*/
```

The function computes the inertia matrix based on the geometry data specified when `physobject_create` was called.

Independently, if you want an object to be able to collide with other objects, and prevent these others objects to penetrate it, you must initialize its geometrical properties (i.e. the volume occupied by the object that cannot be occupied by another object) :

```
physobject_init_geometry(pobj1);
```

In cgmdode, a physical object can have a body and/or a geometry, but not necessarily both. This is determined by whether or not you called `physobject_init_body`, `physobject_init_geometry`, or both. **But you have to call at least one of the two.**

If you initialize the geometry but not the dynamical body of an object, then external forces will have no effect on the object, but it will still collide with other moving objects (it will act as an immobile obstacle).

If you initialize the dynamical body but not the geometry of an object, it will be able to move around and interact with other objects through interaction sites, but it will never collide with anything (it will pass through other objects).

If you initialize both properties, then external forces will affect the object's trajectory, and so will collisions with other objects.

Only after a physobject has been properly created and initialized can you assign arbitrary forces on it, or set its position and velocity, or assign interaction sites and thermostats to it.

III.4 - Assigning interaction sites to physical objects

You can add punctual interaction sites to your physobjects by using the `interaction_class` / `interaction_site` system. The first step is to create an `interaction_class` object. You must create one `interaction_class` object for each type of interaction you need in your simulation. Only interaction sites belonging to the same interaction class will interact.

III.4.A - Creating an interaction class

This object will store data common to all `interaction_site` objects that belong to the `interaction_class` : the external parameters and the function that gives the amplitude of the interaction force between two interaction sites of the same class depending on their distance. This force is assumed to be reciprocal : if two sites `isite1` and `isite2` interact with each other, then the amplitude of force applied by 2 on 1 is the opposite of the amplitude of the force applied by 1 on 2. By convention, a negative amplitude gives an attractive force (interaction sites are pulled together), and a positive amplitude a repulsive force (sites are pushed away).

At each time step, for each pair of interaction sites of the same interaction class, cgmdode will take the force amplitude function you defined and use it to compute the interaction force between the pair of interaction sites, and apply this force on the bodies to which the interaction sites belong. So you will never have to call the force amplitude function yourself : cgmdode will do it for you automatically, but you do have to write it and tell cgmdode where to find it with a function pointer.

Force amplitude function

So first you need to create a force amplitude function. This function gives the amplitude of the force that an interaction site `isite1` applies to an interaction site `isite2` that belong to the same class as `isite1`. All force amplitude functions must have a specific prototype so that cgmdode can use them. In cgmdode you need to write your own force amplitude function, there are no default function.

This prototype is :

```
t_real force_amplitude(t_real ,
                      interaction_site* ,
                      interaction_site* ,
                      interaction_class* );
```

Here is an example of force amplitude function :

```
t_real force_coulomb(t_real distance, interaction_site* isite1, interaction_site* isite2,
interaction_class* iclass)
{
    t_real force_amplitude_1_on_2 = 0.0;

    /*
     * compute a force_amplitude that isite1
     * applies on isite2 depending on :
     * the distance between isite1 and isite2
     * the parameters of the interaction sites and
     * the parameters of the class these sites belong to
     */

    force_amplitude_1_on_2 = (isite1->params[0]*isite2->params[0]) \
        / (iclass->params[0]*distance*distance) ;

    return force_amplitude_1_on_2;
}
```

Within the code of your function, you have access to various data so that you can define precisely your interaction force. In particular :

data	type	description
distance	t_real	the euclidean distance between the two interaction sites <code>isite1</code> and <code>isite2</code> .
<code>isite1->params</code> <code>isite2->params</code>	t_real*	The parameters that are specific of the interaction site {1,2} («internal» parameters)
<code>iclass->params</code>	t_real*	The parameters that are specific of the interaction class to which the interaction sites belong («external» parameters)

In the example function above, you can think of the function as the Coulomb Force interaction, where `isite1->params[0]` is the electrical charge of `isite1`, `isite2->params[0]` is the electrical charge of `isite2`, and `iclass->params[0]` is $4\pi\epsilon_0$.

In addition to this data, you also have access to the data that you set as the userdata field of both `isite1` and `isite2`, as well as `iclass`, when you created them. You can use them in the force amplitude functions you define to create any interaction behavior you desire.

Note that you can also access the physobjects containing `isite1` and `isite2` within the amplitude function, using the `isite1->physobj` and `isite2->physobj` pointers. You can call `cgmdode` functions on these physobjects, although **you must not modify their position or their orientation**. A lot of behaviors are available for the amplitude function, as you may for instance return 0.0 for the force amplitude, and decide to apply forces and/or torques yourself. This allows for a lot more behaviors than simply applying forces on the interaction sites.

Interaction class initialization

To create an `interaction_class` object, you must use the `interclass_create` function. For example, to create the Coulomb interaction force reusing the force amplitude function we defined earlier, and attach it to a `simcon` simulation context :

```
interaction_class* iclass_coulomb = malloc(sizeof(*iclass_coulomb));

t_real* coulomb_params = malloc(1*sizeof(*coulomb_params));
coulomb_params = 4.0 * 3.14159 * 8.85e-12 ; // 4*pi*eps_0

interclass_create(iclass_coulomb,
    17,           // int : user-defined id of the interaction class
    coulomb_params, // t_real* : array of parameters of the class
    1,           // uint : nb of parameters in coulomb_params
    &force_coulomb, // void* : function pointer to the force amplitude function
    0x0,         // void* : userdata pointer
    simcon);     // simcontext* : simulation context

free(coulomb_params); // no longer need as parameters are copied internally.
```

III.4.B - Adding interaction sites to a physobject

Once you have your `interaction_class` object, you can add interaction sites belonging to this interaction class to a given physobject by invoking `physobject_add_intersite` :

```
/*
    parameters specific to the site
    just one : its electrical charge
*/
t_real* isite_params = malloc(1*sizeof(*isite_params));
isite_params[0] = 1e-2 ;

physobject_add_intersite(obj, // physobject* : the target physobject
    0,           // int : id of the interaction_site
    iclass_coulomb, // interaction_class* : interaction class of the site
    0., 0., 0., // t_real : x,y,z of the site in the local physobject referential
    1,           // uint : size of isite_params
    isite_params, // t_real* : array containing the intersite parameters
    0x0);        // void* : userdata pointer

free(isite_params);
```

The example above add an interaction site to the `obj` physobject that corresponds to an electrical charge (undergoing Coulomb forces applied by neighboring electrical charges) at the center of the physobject's local referential.

If you want to be able to remove / add interaction sites on-the-fly during the simulation, be sure to assign a unique `id` for each interaction site that belongs to the same physobject since the function `physobject_remove_intersite` doesn't work properly with duplicate ids. Note however that `cgmdode` completely ignores user-specified ids and uses its own independent indexing method during simulation.

III.5 - Assigning physical objects to thermostats

In a typical statistical mechanics simulation, a thermostat ensures that the temperature of the simulated system, that is the average kinetic energy of particles, remains constant. This is achieved by correcting the velocity of all particles at each timestep, according to one among many available algorithms. `cgmdode` generalizes this concept for coarse-grained objects. You can define rules that dictates the dynamics of each object at each timestep, i.e. how the motion of the object will change. For example, the rules could be "at each time step, slow down the translational velocity of all objects proportionally to a friction coefficient, and apply a random force to each object".

The role of the `thermostat_class` / `thermostat_data` system is to provide you with a simple way to create any thermostat, and is similar to the `interaction_class` / `interaction_site` system.

III.5.A - Creating a thermostat class

A thermostat class is mainly defined by the following components :

- a thermostat function that will be applied to each physobject assigned to the thermostat class at each time step
- external parameters that are common to all physobjects assigned to the same thermostat class.

In order to create a `thermostat_class`, you first need to define its thermostat function.

Thermostat function

All thermostat functions must match the following prototype :

```
void therm_function(physobject* , thermostat_data* , thermostat_class* );
```

At each time step, this function will be called for all physobjects assigned to the thermostat class. Your code will never have to call this function itself explicitly, cgmdode will do it automatically. But you do have to write the function and tell cgmdode where to find it with a function pointer.

Here is an example of thermostat function :

```

void thermfunc_friction_fluctuation(physobject* pobj,
                                   thermostat_data* thermdata,
                                   thermostat_class* thermclass)
{
    /*
     * This thermostat function simply applies
     * a correction force on pobj that depends on
     * a translational drag and a translational fluctuation
     * (non-specific so defined in thermclass)
     * and also a coefficient defined specifically for pobj
     * (specific so defined in thermdata)
     */

    /* fetch pobj's current linear velocity :
     * lvel[0] : x velocity in world referential
     * lvel[1] : y velocity in world referential
     * lvel[2] : z velocity in world referential
     */
    t_real lvel[3];
    physobject_get_linear_vel(po, lvel);

    /*
     * fetch the thermostat class parameters :
     */
    t_real friction = thermclass->params[0];
    t_real fluctuation = thermclass->params[1];

    /*
     * fetch the coefficient that
     * is specific of pobj (stored in thermdata)
     */
    t_real coeff = thermdata->params[0];

    /*
     * create a random vector
     * whose components are drawn
     * in a N(0,1) normal distribution
     */
    t_real ranvec[3];
    ranvec[0] = ran_gaussian(0.0, 1.0);
    ranvec[1] = ran_gaussian(0.0, 1.0);
    ranvec[2] = ran_gaussian(0.0, 1.0);

    /*
     * apply the correction force to pobj
     */
    physobject_add_force(pobj,
                        -friction*coeff*lvel[0] + coeff*fluctuation*ranvec[0],
                        -friction*coeff*lvel[1] + coeff*fluctuation*ranvec[1],
                        -friction*coeff*lvel[2] + coeff*fluctuation*ranvec[2]);
}

```

Within the function, you have access to thermostat parameters that are specific to the physobject through `thermdata`, and access to parameters that are non-specific and common to all physobjects assigned to the thermostat class through `thermclass`.

Thermostat class initialization

Now that you have your thermostat function, you can create the thermostat class :

```

/*
    non-specific parameters
    thermclass_params[0] : friction
    thermclass_params[1] : fluctuation
*/
t_real* thermclass_params = malloc(2*sizeof(*thermclass_params));
thermclass_params[0] =1.0;
thermclass_params[1] =2.0;

thermostat_class* example_therm = malloc(sizeof(*example_therm));
thermclass_create(example_therm,
    21,                // int : thermostat class user-defined id
    &thermfunc_friction_fluctuation, // void* : function pointer
    2,                // uint : nb of thermclass params
    thermclass_params, // t_real* : thermostat class params
    0x0,              // void* : userdata pointer
    simcon);           // simcontext* : simulation context

free(thermclass_params); // copied internally, no longer needed.

```

III.5.B - Setting a physobject's thermostat data

You can assign a `pobj` physobject to a thermostat class `thermclass_example` by using `physobject_set_thermdata` :

```

/*
    thermostat parameters that
    are specific of pobj
*/
t_real* thermdata_params = malloc(1*sizeof(*thermdata_params));
thermdata_params[0] = 0.5; /* coeff in thermfunc_friction_fluctuation */

physobject_set_thermdata(pobj, /* physobject* : the object */
    thermclass_example, /* thermostat_class* : destination thermclass */
    1, /* uint : nb of thermdata params */
    thermdata_params, /* t_real* : thermdata params */
    0x0); /* void* : userdata pointer */

free(thermdata_params); /* copied internally, no longer needed. */

```

III.6 - Linking physical objects using mechanical joints.

cgmdode provides a way to bind physobjects with mechanical joints. These joints restrict the relative motion of two connected physobjects. Depending on the joint type and parameters, cgmdode (through ODE) will automatically apply the corrections so that the motion of connected physobjects satisfies the constraint imposed by their connecting joint. Such joints are manipulated in cgmdode using `articulation` objects. The complete specifications are detailed in section IV.9. In this example, we will attach two already existing cylindric physobjects with a hinge joint parallel to their main axis.

III.6.A - Creating an articulation between two physobjects

Every joint you want to create requires its own `articulation` object. The two connected physobjects must be properly created and placed in the correct relative orientation / position before you attach them with an articulation. Here we will demonstrate how to create two cylinders linked by a hinge. The two cylinders will be placed on top of each other, their flat faces touching and their main axes aligned. The hinge will be set so that the two cylinders will remain parallel and their relative distance will stay the same, but they will be able to rotate freely around their common aligned axis.

So let's first create an articulation named `hinge_art` into the `simcon` simulation context :

```

articulation* hinge_art = malloc(sizeof(*hinge_art));

articulation_create(hinge_art,      /* articulation* : the articulation object */
                    1,              /* int : user-defined articulation id */
                    JOINT_HINGE,    /* int : cgmdode joint type */
                    0x0,            /* void* : callback function pointer */
                    0x0,            /* void* : userdata pointer */
                    simcon);        /* simcontext* : simulation context */

```

The callback function pointer lets you set a callback to a function that will be called at each timestep for this articulation. See API reference section IV.9 for more details. We don't need one here.

The next step is to attach the physobjects with the articulation. It must be done before setting the articulation parameters. In the following snippet we will create two cylinders as physobjects and attach them with hinge_art.

So let's create the cylinders :

```

t_real cylinder_radius = 1.0 ;
t_real cylinder_length = 5.0 ;
t_real cylinder_mass = 1.0 ;

/*
   Creates two identical cylinders.
   We'll use the same geometry_data
   for the cylinders.
*/

geometry_data* cylinder_shape = malloc(sizeof(*cylinder_shape));
geomdata_create_cylinder(cylinder_shape, cylinder_radius, cylinder_length);

physobject* cylinder_1 = malloc(sizeof(*cylinder_1));
physobject* cylinder_2 = malloc(sizeof(*cylinder_2));

physobject_create(cylinder_1, 1, cylinder_shape, 0x0, simcon);
physobject_create(cylinder_2, 2, cylinder_shape, 0x0, simcon);

/*
   For the joint to work, the cylinders' dynamical properties
   must be initialized :
*/
physobject_init_body(cylinder_1, cylinder_mass, INIT_MASS_TOTAL);
physobject_init_body(cylinder_2, cylinder_mass, INIT_MASS_TOTAL);

```

Then let's place them in the right orientation and position and bind them with the hinge :


```

/*
    We want to create a hinge joint between these two objects.
    We will set the hinge so that the two cylinders main axes
    are parallel (their local z-axis).
    We will place cylinder_1 on top of cylinder_2,
    and make the joint so that they freely rotate
    around their main axis.

    By default, when cgmdode creates a cylinder,
    its main axis is the local z-axis of the physobject reference frame,
    and it is aligned with the z-axis of the world reference frame.
    The origin of the local frame of reference is
    at the center of the cylinder.

    So if we set the position of cylinder_1 at (0,0,0),
    and the position of cylinder_2 at (0,0,cylinder_length),
    they will be in the correct relative orientation and position.
*/

physobject_set_position(cylinder_1, 0.0 , 0.0 , 0.0);
physobject_set_position(cylinder_2, 0.0 , 0.0 , cylinder_length);

/*
    We attach the physobjects :
*/

articulation_attach_physobjects(hinge_art, cylinder_1, cylinder_2);

```

III.6.B - Setting articulation parameters

Now we need to set the joint parameters. A hinge joint requires two parameters : the coordinates of an anchor, and the vector determining the axis of free rotation.

The anchor is given as an (x,y,z) world coordinates. When it is set, cgmdode computes the coordinates of this same global anchor point in the local reference frames of both physobjects. When the joint is created, the distance between the two resulting local anchor points is 0. cgmdode will make sure during the simulation that this distance remains 0 by applying corrections on the physobjects trajectory.

The hinge axis is also given in world coordinates. These two parameters combined will create a hinge at the anchor point, aligned along the hinge axis, between the two cylinders.

How do you set joint parameters in cgmdode ? Although there are several types of joints each having different kinds of parameters, a single cgmdode function fulfill this role : [articulation_set_params](#). See section IV.9.C on how to use it. The first argument is the articulation. The second is a constant stating the kind of parameter, usually named ART_PARAM_<parameter name>. The third argument is an array of [t_real](#) containing the values of the parameter to be set. See API reference section IV.9.C for more details.

For our example, this gives :

```

/* the anchor point in world coords */
t_real anchor_coords[3];
anchor_coords[0] = 0.0 ; // x
anchor_coords[1] = 0.0 ; // y
anchor_coords[2] = 2.5 ; // z

/* the hinge axis vector */
t_real hinge_axis[3];
hinge_axis[0] = 0.0 ; // x
hinge_axis[1] = 0.0 ; // y
hinge_axis[2] = 1.0 ; // z

articulation_set_params(hinge_art, ART_PARAM_ANCHOR, anchor_coords);
articulation_set_params(hinge_art, ART_PARAM_AXIS, hinge_axis);

```

III.7 - Running the simulation

Once you have created all the physical objects you need, set their interaction sites, assigned them to thermostat classes, and connected them with articulations, you can advance the simulation by calling :

```
simulation_step(simcon);
```

That's it. Time will be incremented by the value `ode_step` you specified, interactions will be computed and resulting forces applied, thermostats will be applied, collisions will be detected and the trajectories of all physobjects corrected accordingly. Constraints imposed by articulations will also be solved and physobjects trajectories affected accordingly. If you required periodic boundary conditions, physobjects positions will be updated and corrected accordingly.

Between two time steps you can :

- modify physobjects positions.
- modify physobjects velocities (although ODE manual recommends **not** to do it)
- add and/or remove articulations between physobjects.
- add and/or remove physobjects from the simulations
- add and/or remove interaction sites
- apply forces and torques on physobjects.
- forbid / allow collisions between two specific individual physobjects
- forbid / allow interactions between two specific individual physobjects

If you destroy an interaction class, you must first be sure to destroy all interaction sites that belong to this class. Same thing for thermostat classes and thermostat data.

Also remember that `simparameters` **must not** be modified after the initialization sequence.

III.8 - Cleaning up

Most `cgmdode` data type has its destructor. You have to free the memory you allocated :

- you manage the memory for `physobject`, `geometry_data`, `thermostat_class`, `interaction_class`, `articulation`, `simcontext` and `simparameters` objects.
- `cgmdode` manages the memory for `interaction_site` and `thermostat_data` objects.

If you want to clean up everything at once, just call :

```
simcontext_destroy(simcon);
simparams_destroy(simparams);

free(simcon);
free(simparams);
```

The order matters ! You must not destroy `simparams` before `simcon`. The main reason behind this is that you might want to reuse the same `simparameters` object for another, fresh `simcontext` object, so there is no reason to destroy `simparams` when destroy `simcon`. The other reason is that `simcontext_destroy` needs `simparams` to know what needs to be cleaned.

IV - API Reference

IV.1 - Basic data types

The following types are wrappers for ODE data types and/or directly imported from ODE that are used in cgmdode :

```
#define t_real dReal
/*
    dReal
    the floating point type of your ODE installation,
    either float or double)
*/
```

```
#define t_matrix3x3 dMatrix3
/*
    dMatrix3 is dReal[3*4]
    row-major order,
    the last column is ignored and is here for alignment
*/
```

```
#define t_vec3 dVector3
/*
    dVector3 is the ODE type dReal[4],
    the last cell is here for alignment
*/
```

```
/*
    dMass : mass data for ODE bodies
    (directly imported from ODE)
*/
typedef struct dMass
{
    dReal mass; //total mass of the rigid body
    dVector3 c; //center of gravity position in body frame (x,y,z)
    dMatrix3 I; //3x3 inertia tensor in body frame, about POR
}dMass;
```

```
typedef dReal[4] dQuaternion ;
/*
    quaternion (w,x,y,z)
    defined in ODE
*/
```

IV.2 - Complex data types

The internal fields of cgmdode objects should generally not be accessed or modified directly, apart a few exceptions described below that may be useful. There are other complex data structures used internally, they should never be used by the user.

The complete structure types are declared in the `<install prefix>/include/cgmdode/cgmdode_simobjects.h` headers.

IV.2.A - Stored user-defined ids and data

The `physobject`, `articulation`, `interaction_class`, `interaction_site`, and `thermostat_class` structures all have a `int id` field and a `void* userdata` field that can be set by the user when the object is created. These fields also can be directly read and written. The `id` field of an `interaction_site` should be treated carefully though : when the user removes an interaction site with the function `physobject_remove_intersite`, cgmdode searches for the interaction site using the `id`.

The `simcontext` structure and the `simparameters` structure have a `userdata` field that can be set by the user when they are created, and can also be directly modified or read during the simulation.

IV.2.B - Encapsulated ODE objects

The field `b_id` of a `physobject` structure is the `dBodyID` of the `physobject`. The field `g_id` of the `physobject` structure corresponds to its `dGeomID`. They are 0 if the ODE body / geom initialization was not required by the user (see section IV.6.B).

Important : the proper functioning of `cgmdode` requires that the `userdata` field of `dGeomID g_id` is not modified during the simulation. We recommend to use the `userdata` field of the `physobject` structure instead.

The `simcontext` structure field `world_id` is a `dWorldID` giving access to ODE's world object. The `simcontext` structure field `space_id` is a `dSpaceID` giving access to ODE's collision space object.

IV.3 - Simparameters functions

Simulation parameters are stored by `cgmdode` using a `simparameters` object.

IV.3.A - Creation / destruction

```
void simparams_create(simparameters* simparams);  
void simparams_destroy(simparameters* simparams);
```

IV.3.B - Setting engine parameters

```
void simparams_set_seed(simparameters* simparams, unsigned long seed);
```

`simparams_set_seed` saves the user-defined `seed` that will be used to initialize the pseudo random number generator (PRNG) shipped with `cgmdode` (see section IV.10.C). **This function does not initialize the PRNG**, the function `simcontext_init` does it. This means that PRNG-related functions are only available **after** `simcontext_init` has been called.

```
void simparams_set_ode_params(simparameters* simparams,  
    int qstep_nbiters,  
    t_real qstep_w,  
    t_real erp_glob,  
    t_real cfm_glob,  
    t_real ode_step,  
    t_real lin_damp,  
    t_real ang_damp,  
    t_real max_corrvel,  
    t_real max_avel,  
    int max_contacts,  
    int contact_mode,  
    t_real contact_mu,  
    t_real contact_mu2,  
    t_real contact_bounce,  
    t_real contact_bounce_vel,  
    int stepper_type,  
    t_real min_object_size,  
    t_real max_object_size);
```

`simparams_set_ode_params` sets the parameters of ODE, the physics engine used by `cgmdode`. All parameters must be provided, even if they will not be used given the user-defined options (such as the stepper type or the contact mode).

Here is a table giving the type, description, and range of all parameters :

type and name	description	range
<code>int qstep_nbiters</code>	number of ODE's SOR constraint solver iterations (used if <code>stepper_type == USER_STEP_QUICK</code>)	≥ 1
<code>t_real qstep_w</code>	relaxation parameter for ODE's SOR constraint solver (used if <code>stepper_type == USER_STEP_QUICK</code>)	> 1.0
<code>t_real erp_glob</code>	Error reduction parameter. See ODE User Manual, rule of thumb : between 0.2 and 0.8.	[0.0 ; 1.0]
<code>t_real cfm_glob</code>	Constraint force mixing parameter. see ODE User Manual, rule of thumb : $1e-10$ if double precision float, $1e-5$ if single precision float.	≥ 0.0
<code>t_real ode_step</code>	time step of ODE numerical integrator for motion.	> 0.0
<code>t_real lin_damp</code>	Linear damping. All linear velocities will be multiplied by $(1 - \text{lin_damp})$ at each time step.	real
<code>t_real ang_damp</code>	Angular damping. Same as linear damping but with angular velocities.	real
<code>t_real max_corrvel</code>	Maximum velocity that contacts (collisions) are able to generate on objects.	≥ 0.0
<code>t_real max_avel</code>	Maximum angular velocity that objects can reach.	≥ 0.0
<code>int max_contacts</code>	Maximum number of contact joints that two colliding objects can generate.	≥ 0
<code>int contact_mode</code>	Physical approximation used when handling collisions.	$\neq 0$, see * below
<code>t_real contact_mu</code>	Friction coefficient on the tangent direction when two objects are colliding.	≥ 0.0
<code>t_real contact_mu2</code>	Friction coefficient on the normal direction when two objects are colliding (only used if <code>dContactMu2</code> is set in <code>contact_mode</code>)	≥ 0.0
<code>t_real contact_bounce</code>	Restitution parameter for bouncy collisions.	≥ 0.0
<code>t_real contact_bounce_vel</code>	Relative velocity above which two colliding objects will bounce away from each other.	≥ 0.0
<code>int stepper_type</code>	Type of constraint solver and numerical integrator used in ODE.	$\neq 0$, see ** below
<code>t_real min_object_size</code>	Size of the smallest object you plan on simulating.	> 0.0 ***
<code>t_real max_object_size</code>	Size of the biggest object you plan on simulating.	$> \text{min_object_size}$ ***

* must a combination of at least one among : `dContactMu2`, `dContactFDir1`, `dContactBounce`, `dContactSoftERP`, `dContactSoftCFM`, `dContactMotion1`, `dContactMotion2`, `dContactSlip1`, `dContactSlip2`, `dContactApprox1_1`, `dContactApprox1_2`, `dContactApprox1`. See ODE User Manual for details. A good minimal working default is `dContactBounce` (elastic collisions depending on parameter `contact_bounce`).

** must be either `USER_STEP_QUICK` (cgmdode will use ODE's SOR constraint solver with [dWorldQuickStep](#)) or `USER_STEP_NORMAL` (cgmdode will use ODE's LCP constraint solver with [dWorldStep](#)). See ODE User Manual for details on these methods.

*** In cgmdode, collision detection is done by ODE using a mutli-resolution hash table (see Hash Space in ODE Manual). `min_object_size` and `max_object_size` are respectively used to set the minimum hash level and maximum hash level of the hash table. You must ensure that all colliding physobjects you plan on simulating will have sizes between `min_object_size` and `max_object_size`.

IV.3.C - Settings medium parameters

```
void simparams_set_medium_params(simparameters* simparams,
                                int boundary_type,
                                t_real world_Lx,
                                t_real world_Ly,
                                t_real world_Lz,
                                unsigned int max_grid_size,
                                t_real min_cell_size);
```

`simparams_set_medium_params` sets the parameters of the medium in which your simulation takes place. Here are the parameter descriptions and expected values :

parameter	description	range
<code>int boundary_type</code>	The type of boundary condition. If you choose periodic boundary conditions, you should avoid using articulations.	INIT_BOUNDARY_PERIODIC or INIT_BOUNDARY_NONE or INIT_BOUNDARY_WALLS
<code>t_real world_Lx</code>	Size of medium along the x axis.	> 0.0
<code>t_real world_Ly</code>	Size of medium along the y axis.	> 0.0
<code>t_real world_Lz</code>	Size of medium along the z axis.	> 0.0
<code>unsigned int max_grid_size</code>	Maximum width of the spatial grid used.	>= 2
<code>t_real min_cell_size</code>	Minimum size that cgmdode can use for a spatial grid cell.	>= 0.0

cgmdode discretizes the medium into a grid of rectangular cells in which physobjects will be placed. Each physobject is assigned a cell of the spatial grid that corresponds to where it is located. When calling `simparams_set_medium_params`, the function will compute automatically the largest size of the spatial grid so that it's smaller or equal to `max_grid_size` (max_grid_size^3 cells total), but with cells of size at least `min_cell_size`. In order to compute the interaction forces acting on a given physobject `pobj`, cgmdode looks for interaction sites of other physobjects located in the 26 neighbouring cells (+1 / -1 on each direction x,y,z) + the cell `pobj` currently occupies. The consequence is that interaction sites of two physobjects only see each other if the physobjects are in adjacent cells.

Therefore you must choose `min_cell_size` and `max_grid_size` depending on two factors :

- the maximum range for any interaction you want to create in your simulation (`max_inter_range`)
- the size of the biggest object you need to simulate (`max_obj_size`).

Setting `min_cell_size > 2 x (max_inter_range + max_obj_size)` guarantees that all pairs of interaction sites located closer than `max_inter_range` will be treated.

If you don't want to set an effective maximum range for interactions, then you need to set `max_grid_size` to 2 : all available space will be discretized in 2x2x2 cells, which are all adjacent to each other. That way all possible interactions are guaranteed to be treated at each time step.

You can impose the spatial grid size by setting `min_cell_size` at 0.0. By doing so, the function will take `max_grid_size` as the spatial grid size.

This spatial grid is used for computing interactions and virtual images when periodic boundary conditions are required. It is not used for collision detection.

See section IV.10.D for important notes on periodic boundary conditions and reflective boundary conditions.

IV.4 - Simcontext functions

IV.4.A - Creation / destruction / Initialization

```
void simcontext_create(simcontext* simcon, simparameters* simparams, void* userdata);
void simcontext_destroy(simcontext* simcon);
```

Memory for `simcon` and `simparams` must be allocated beforehand. `simparams` must have been created and filled

with parameters. `userdata` lets the user attach arbitrary data to `simcon`.

```
void simcontext_init(simcontext* simcon);
```

IV.4.B - Simulation execution

```
void simulation_step(simcontext* simcon);
```

`simulation_step` advances the simulation by a time increment `ode_step` specified when `simparams_set_ode_params` was called.

IV.4.C - Convenience functions

```
interaction_class* simcontext_get_from_userid_interclass(simcontext* simcon, int userid);
```

This function returns the pointer to the interaction class that was given the id `userid` by the user.

IV.5 - Geometry data functions

IV.5.A - Simple geometric primitives

```
void geomdata_create_point(geometry_data* gdata);
void geomdata_create_sphere(geometry_data* gdata, t_real radius);
void geomdata_create_box(geometry_data* gdata, t_real lx, t_real ly, t_real lz);
void geomdata_create_cylinder(geometry_data* gdata, t_real radius, t_real length);
void geomdata_create_plane(geometry_data* gdata,
                           t_real a, t_real b, t_real c, t_real d);
void geomdata_destroy(geometry_data* gdata)
```

`gdata` must be already allocated by the user. These functions initialize `gdata` properly. It can be then passed to `physobject_create`. All shapes are created so that the origin of the local referential is the center of mass of the corresponding volume as if it were a homogeneous solid body. `geomdata_create_cylinder` creates a z-axis aligned cylinder by default (if you want to change the axis, you will need to rotate the `physobject`). `geomdata_create_plane` creates geometry data for a plane of equation $a*x + b*y + c*z = d$ in world coordinates.

IV.5.B - Trimesh geometry

These functions allow the creation of trimesh collision shapes.

```
void geomdata_create_trimesh(geometry_data* gdata,
                             t_real* vertices_coords,
                             unsigned int nb_vertices,
                             unsigned int* faces_indices,
                             unsigned int nb_faces);
```

`geomdata_create_trimesh` creates the internal representation for a trimesh given a set of vertices and triplets of indices describing the faces. The parameters must follow this scheme :

`vertices_coords` is an array of `t_real` storing the coordinates of each vertex : [`v1_x`, `v1_y`, `v1_z`, ..., `vi_x`, `vi_y`, `vi_z`, ...]. So coordinates `x,y,z` of the vertex `i` are respectively given by :

```
vertices_coords[3*i+0] // (= coord x of vertex i)
vertices_coords[3*i+1] // (= coord y of vertex i)
vertices_coords[3*i+2] // (= coord z of vertex i)
```

`faces_indices` is an array of unsigned int giving the indices of vertices of a given face. So each face `j` is stored as the triplet :

```
faces_indices[3*j+0] // (= index of vertex m)
faces_indices[3*j+1] // (= index of vertex n)
faces_indices[3*j+2] // (= index of vertex p)
```

Each item of the triplet contains the index of one vertex constituting the face (corresponding to the index `i` in the `vertices_coords` array).

cgmdode outsources the computation the trimesh mass properties to ODE. This computation only works if all of the normals to the trimesh's faces point outward the trimesh. The orientation of the face (outward/inward the trimesh) is determined by the order of the vertices indices in the face triplet.

Within a face triplet, vertex indices must be ordered following the counter-clockwise convention., aka the Right Hand Rule :

Let M, N and P be the vector coordinates of the vertices m, n and p. MN is the vector from M to N and MP the vector from M to P. By cgmdode convection, the vector given by the cross product $MN \times MP$ points towards the outside of the face.

```
void geomdata_trimesh_center_on_com(geometry_data* gdata, t_real* translation);  
void geomdata_trimesh_align_principal_axes(geometry_data* gdata,  
                                           t_matrix3x3 rotation_matrix);
```

`geomdata_trimesh_center_on_com` computes the coordinates of the center of mass of the trimesh (as if it represents a solid object with homogeneous mass density), and then translates the trimesh vertices so that the center of mass is at (0,0,0).

You can only use `geomdata_trimesh_align_principal_axes` on trimeshes that are already centered on their center of mass (i.e. you should call `geomdata_trimesh_center_on_com` beforehand).

`geomdata_trimesh_align_principal_axes` computes the inertia moments of the trimesh (as if it represents a solid object with homogeneous mass density), and then rotates the trimesh vertices so that the inertia moments are aligned on the x, y and z axes. The moment of inertia tensor of the resulting physobject body will therefore be a diagonal matrix (actually with close-to-zero off-diagonal terms).

`geomdata_trimesh_align_principal_axes` writes the rotation matrix it used to rotate the original trimesh into `rotation_matrix`, so the user can examine it afterwards.

IV.5.C - Utility functions

```
void geomdata_copy(geometry_data* from, geometry_data* to);  
t_real geomdata_distance_of_farthest_point(geometry_data* geomdata);  
void geomdata_scale_lengths(geometry_data* geomdata, t_real scale_l);
```

`geomdata_copy` creates an independent copy of a geometry data (memory must be allocated for `to`).

`geomdata_distance_of_farthest_point` returns the euclidean distance of the farthest point from the local origin contained in the volume defined by a geometry data (note that it also works for trimeshes).

`geomdata_scale_lengths` scales the geometry data by `scale_l` in place (i.e. it divides all lengths of the geometry data by `scale_l`).

IV.6 - Physobject functions

IV.6.A - Creation / destruction

```
void physobject_create(physobject* physobj,  
                      int id,  
                      geometry_data* geomdata,  
                      void* userdata,  
                      simcontext* simcon);  
void physobject_destroy(physobject* physobj);
```

Memory for `physobj` must be allocated before calling `physobject_create` (and freed when no longer needed). `geomdata` must be an already allocated and properly created geometry data object. `id` is a user-defined id not used by cgmdode internally. `userdata` is a pointer to user-defined data, a null pointer can be passed if this is not required.

IV.6.B - Initializing dynamical et geometrical properties

```
void physobject_init_body(physobject* physobj, t_real mass_val, int init_type);  
void physobject_init_body_custom_mass(physobject* physobj, dMass * custom_mass);
```

physobject_init_body initializes internal objects representing the mass distribution of the physical object. It uses the geometry data provided when **physobject_create** was called. If **init_type** is **INIT_MASS_TOTAL**, then the function assigns an inertia matrix to the object so that its total mass equals **mass_val**. If **init_type** is **INIT_MASS_DENSITY**, then the functions assigns an inertia matrix to the object so that its density equals **mass_val**.

physobject_init_body_custom_mass let the user assign a custom, arbitrary **dMass** object to the physobject (you may refer to ODE manual to know how to build a valid ODE **dMass** object.)

```
void physobject_init_geometry(physobject* physobj);
```

physobject_init_geometry initializes internal collision objects representing the geometry of **physobj** as specified by the geometry data provided when created with **physobject_create**.

IV.6.C - Getting position, orientation, velocities and mass

```
const t_real* physobject_get_position(physobject* physobj);  
const t_real* physobject_get_rotation_matrix(physobject* physobj);
```

These functions return a pointer to the internal objects storing the **physobject*** position and rotation matrix, so they cannot not be modified.

```
void physobject_get_quaternion(physobject* physobj, t_real* res);  
void physobject_get_linear_vel(physobject* physobj, t_real* res);  
void physobject_get_angular_vel(physobject* physobj, t_real* res);  
void physobject_get_angular_vel_relative(physobject* physobj, t_real* res);
```

res must be properly allocated beforehand, as these functions copy the quaternion / linear velocity 3-vector / angular velocity 3-vector into it. **physobject_get_angular_vel_relative** returns the angular velocity expressed in the physobject frame of reference.

```
void physobject_get_mass(physobject* physobj, dMass * res_mass);
```

physobject_get_mass copies the **dMass** object associated with the **physobject*** into **res_mass**, which must be allocated beforehand.

IV.6.D - Transforming coordinates

```
void physobject_from_world_pos_to_relative(physobject* physobj, t_real px,  
t_real py, t_real pz, t_real* res);  
void physobject_from_relative_pos_to_world(physobject* physobj, t_real px, t_real py, t_real  
pz, t_real* res);
```

These functions transform the coordinates of a point between the world referential and a physobject local referential.

```
void physobject_from_world_vector_to_relative(physobject* physobj, t_real vx,  
t_real vy, t_real vz, t_real* res);  
void physobject_from_relative_vector_to_world(physobject* physobj, t_real vx,  
t_real vy, t_real vz, t_real* res);
```

These functions transform a vector expressed in the world referential to a physobject local referential (and reciprocally).

IV.6.E - Setting position, orientation, velocities

```
void physobject_set_position(physobject* physobj, t_real x, t_real y, t_real z);  
void physobject_set_quaternion(physobject* physobj, dQuaternion q);  
void physobject_set_rotation_matrix(physobject* physobj, const t_matrix3x3 R);
```

```
void physobject_set_linear_vel(physobject* physobj,  
                               t_real lvel_x,  
                               t_real lvel_y,  
                               t_real lvel_z);  
void physobject_set_angular_vel(physobject* physobj,  
                                t_real avel_x,  
                                t_real avel_y,  
                                t_real avel_z);
```

Most of these functions are in fact wrappers for ODE's rigid body functions and have the same form. They do handle the special cases of physobjects without a body or without a geometry. Velocity setters have no effect on bodiless physobjects.

IV.6.F - Adding forces and torques

```
void physobject_add_force(physobject* physobj, t_real fx, t_real fy,  
                          t_real fz);  
void physobject_add_relative_force(physobject* physobj, t_real fx, t_real fy,  
                                  t_real fz);  
void physobject_add_force_at_relative_point(physobject* physobj,  
                                             t_real fx, t_real fy, t_real fz,  
                                             t_real px, t_real py, t_real pz);  
void physobject_add_force_at_point(physobject* physobj,  
                                   t_real fx, t_real fy, t_real fz,  
                                   t_real px, t_real py, t_real pz);  
void physobject_add_relative_force_at_relative_point(physobject* physobj,  
                                                      t_real fx, t_real fy, t_real fz,  
                                                      t_real px, t_real py, t_real pz);  
void physobject_add_relative_force_at_point(physobject* physobj,  
                                             t_real fx, t_real fy, t_real fz,  
                                             t_real px, t_real py, t_real pz);  
void physobject_add_torque(physobject* physobj, t_real tx, t_real ty, t_real tz);  
void physobject_add_relative_torque(physobject* physobj, t_real tx, t_real ty, t_real tz);
```

In these function names the prefix **relative** means that the coordinates/components are expressed in the body local referential, otherwise they are expressed in the world referential (this applies to **point** as well as **force** and **torque**). These functions have no effect on bodiless physobjects.

IV.6.G - Adding and removing interaction sites

```
void physobject_add_intersite(physobject* physobj,  
                              int id,  
                              interaction_class* interclass,  
                              t_real relcx, t_real relcy, t_real relcz,  
                              unsigned int nb_params,  
                              t_real* params,  
                              void* userdata);  
int physobject_remove_intersite(physobject* physobj, int id);  
void physobject_destroy_all_intersites(physobject* physobj);
```

physobject_add_intersite adds an interaction site of the specified **interclass** **interaction_class**, at the coordinates (**relcx**, **relcy**, **relcz**) expressed in the body local referential. **id** must be specified and must be unique : **in a given physobject, two interaction sites must not have the same id value**. The function copies the contents of **params** (which must be a **t_real[nb_params]** array) internally, so the original params array is not longer required once the interaction site has been created.

physobject_remove_intersite destroys all internal data related to the interaction site identified by **id** belonging to **physobj**. The function returns 0 if no interaction site with the specified **id** was found, a non-null value otherwise. If several interaction sites have the same id within **physobj**, one of them is removed (but it cannot be determined which one). This function does not delete data pointed by **userdata**.

physobject_destroy_all_intersites destroys all the interaction sites of a **physobject**.

IV.6.H - Setting thermostat data

```
void physobject_set_thermdata(physobject* physobj,
                             thermostat_class* thermclass,
                             unsigned int nb_params,
                             t_real* params,
                             void* userdata);
```

physobject_set_thermdata sets the thermostat data associated with **physobj**. The contents of the **t_real[nb_params]** array **params** will be copied internally so that **params** can be freed after having called the function. If previous thermostat data was already assigned to **physobj**, it will be lost.

IV.6.I - Detailed collision / interaction management

These functions will enable/disable the effects of collisions between two **physobjects**, or between one **physobjects** and every other one (including world boundaries). All collisions are enabled by default, so **physobject_allow_collision_between** is not required when creating new **physobjects**.

```
void physobject_forbid_collision_between(physobject* pobj1, physobject* pobj2);
void physobject_allow_collision_between(physobject* pobj1, physobject* pobj2);
void physobject_allow_all_collisions(physobject* physobj);
```

Note that if you specified a custom collision callback (see section IV.10.B), cgmdode will not know whether a collision should be ignored or treated. Therefore these functions will have no effect if you use a custom collision callback.

```
void physobject_forbid_interaction_between(physobject* pobj1, physobject* pobj2);
void physobject_allow_interaction_between(physobject* pobj1, physobject* pobj2);
void physobject_allow_all_interactions(physobject* physobj);
```

Same but for interactions (defined by interaction sites / interaction classes).

IV.7 - Interaction class functions

IV.7.A - Creation / destruction

```
void interclass_create(interaction_class* interclass,
                      int id,
                      t_real* params,
                      unsigned int nb_params,
                      void* interaction_amplitude,
                      void* userdata,
                      simcontext* simcon);

void interclass_destroy(interaction_class* interclass);
```

`params` is an array of `t_real` containing external parameters : parameters that are common to every interaction sites of this class. `nb_params` gives the number of such parameters. These are copied internally, so the `params` array can be freed right after `avec` created the interaction class object.

IV.7.B - Interaction amplitude function

In order to create an `interaction_class` object, you also need to provide a pointer `interaction_amplitude` to a function that will give the amplitude of the interaction force between two interaction sites of the same class. This function must have the prototype :

```
t_real interaction_amplitude(t_real ,
                           interaction_site* ,
                           interaction_site* ,
                           interaction_class* );
```

You have to write the code of this function yourself. This function is called whenever two physobjects containing interaction sites of this class are close (the maximum distance is determined by `min_cell_size`, see section IV.3.C). This function must return the amplitude of the force that will be applied by `cgmdode` on both interacting physobjects. The force will be applied on the rigid bodies at the coordinate of the interaction sites (not at the center of mass). The force applied on the first physobject will be the opposite of the force applied on the second physobject. It's up to you to write the code that gives the amplitude of the force.

Inside the interaction amplitude function, you have access to the function arguments determined by the prototype :

<code>t_real distance :</code>	the euclidean distance between interaction sites when the function was called
<code>interaction_site* isite1</code>	the <code>interaction_site</code> object on the first physobject
<code>interaction_site* isite2</code>	the <code>interaction_site</code> object on the first physobject
<code>interaction_class* iclass</code>	the <code>interaction_class</code> object corresponding to <code>isite1</code> and <code>isite2</code>

Through these arguments, you have indirectly access to :

<code>isite1->physobj</code> and <code>isite2->physobj</code>	the physobjects containing the interaction sites
<code>iclass->simcon</code>	the whole <code>simcontext</code> object
<code>iclass->params</code>	the external parameters of the interaction (e.g. physical constants, etc.)
<code>isite1->params</code> and <code>isite2->params</code>	the internal parameters of the interaction (e.g. charge, polarity, etc.)

Additionally, you have access to any `userdata` pointer you specified when creating any of these objects. Finally, you can call `cgmdode` functions inside the `interaction_amplitude` code, **provided that you do not modify the position of the physobjects and that you do not destroy the interaction sites**. Note however that you may do so between two `simulation_step` calls.

Therefore, you can use all these elements inside the `interaction_amplitude` function to precisely describe the behavior you expect when two physobjects interacts with eachother through interaction sites. For instance, you may decide that `interaction_amplitude` always return 0.0, so no force will be applied on the physobjects, but instead the function might itself apply a torque on the physobjects, set a trigger, modify user-specified data, change the physobjects masses, etc.

IV.8 - Thermostat class functions

IV.8.A - Creation / destruction

```
void thermclass_create(thermostat_class* thermclass,
                      int id,
                      void* therm_function,
                      unsigned int nb_params,
                      t_real* params,
                      void* userdata,
                      simcontext* simcon);

void thermclass_destroy(thermostat_class* thermclass);
```

`params` is an array of `t_real` containing external parameters : parameters that are common to the dynamics of all physobjects that will be assigned this thermostat class. `nb_params` gives the number of such parameters. These are copied internally, so the `params` array can be freed right after having created the thermostat class object.

IV.8.B - Thermostat function

A thermostat class requires a pointer to a function that determines what happens to physobjects assigned to the thermostat class at each timestep. This function must follow the prototype :

```
void therm_function(physobject* , thermostat_data* , thermostat_class* );
```

This function will be called on every physobject assigned to the corresponding thermostat class, at every timestep. You have to write the code yourself. Inside the thermostat function, you have access through its arguments to :

<code>physobject* pobj :</code>	the physobject currently acted on.
<code>thermostat_data* thermdata :</code>	the thermostat data assigned to pobj by <code>physobject_set_thermdata</code> .
<code>thermostat_class* thermclass :</code>	the thermostat class object.

These objects also indirectly give you access to :

<code>thermclass->params :</code>	thermostat parameters common to all physobjects assigned to the thermostat (e.g. medium viscosity, temperature, etc.)
<code>thermdata->params :</code>	thermostat parameters specific to <code>pobj</code> (e.g. translational rotation friction, etc.)
<code>pobj->simcon</code> or <code>thermclass->simcon</code>	the whole simcontext object.

Unlike the interaction amplitude function system, `cgmdode` does not take the return value of the thermostat function to affect the physobjects (since the thermostat function return type is `void`). The thermostat function itself must call the `cgmdode` functions that will affect the physobjects (for instance `physobject_add_force` and `physobject_add_torque`).

IV.9 - Articulation functions

IV.9.A - Creation / destruction

```
void articulation_create(articulation* artobj,
                        int id,
                        int type,
                        void* userdata,
                        void* callback,
                        simcontext* simcon);

void articulation_destroy(articulation* artobj);
```

Articulation types available for the argument **type** (see ODE manual and below for articulation parameters) :

JOINT_FIXED	Fixed joint : the relative orientation and position of the two attached physobjects will remain the same.
JOINT_BALL	Ball-in-socket joint
JOINT_HINGE	Hinge joint
JOINT_HINGE2	Double hinge joint : two hinges (with different axes) connected in series.
JOINT_UNIVERSAL	Universal joint : two perpendicular rotation axes.
JOINT_PRISMROT	Prismatic-rotoid joint : one slider axis + one rotation axis
JOINT_DBALL	Double ball-in-socket : maintains a fixed distance between two points on two different bodies, no constraint on their relative rotation.

the **callback** function pointer can be used to make cgmdode call a user-defined function at each time step. If you don't need a callback, pass a null pointer. The prototype of the callback function must be :

```
void callback(articulation* );
```

The callback will be called at each timestep, passing the articulation object to the callback using the **articulation* artobj** argument. Through this object, you have access to :

artobj ->obj1	the first physobject attached to the articulation
artobj ->obj2	the second physobject attached to the articulation
artobj ->userdata	pointer to user-defined data
artobj ->simcon	the whole simcontext object.

IV.9.B - Physobject attribution

These functions attach and detach physobjects using an already created articulation object.

```
void articulation_attach_physobjects(articulation* artobj,
                                     physobject* pobj1,
                                     physobject* pobj2);

void articulation_detach_physobject(articulation* artobj, physobject* physobj);
```

You may attach two physobjects together, or attach one physobject "with the world". To do the latter, you can pass a null pointer as **pobj2**. Attaching the physobjects does not sets the internal data required for maintenance of the joint yet. You will still need to use **articulation_set_params** to enable the joint (see below). You do need to attach the physobjects with the joint before setting the joint parameters.

IV.9.C - Articulation parameters setting

There is only one function to set the parameters for any joint type.

```
void articulation_set_params(articulation* artobj, int param_type, t_real* params);
```

Important : physobjects must already have been placed in the correct relative orientation and position, and attach to the articulation, before setting the anchors and axes of articulations.

Each joint requires specific parameters. For instance, a ball-in-socket joint only requires an anchor coordinate, but a hinge joint requires the triplet describing the axis around which rotation is allowed. The following table sums the different joint parameters and how to set them using **articulation_set_params** :

Articulation type	int param_type	t_real* params
JOINT_FIXED	ART_PARAM_ANCHOR	[x, y, z] : anchor position in world coordinates
JOINT_BALL	ART_PARAM_ANCHOR	[x, y, z] : anchor position in world coordinates
JOINT_HINGE	ART_PARAM_ANCHOR	[x, y, z] : anchor position in world coordinates
	ART_PARAM_AXIS or ART_PARAM_AXIS_1	[a1x, a1y, a1z] : axis 1 in world coordinates
JOINT_HINGE2	ART_PARAM_ANCHOR	[x, y, z] : anchor position in world coordinates
	ART_PARAM_AXIS_1	[a1x, a1y, a1z] : axis 1 in world coordinates
	ART_PARAM_AXIS_2	[a2x, a2y, a2z] : axis 2 in world coordinates
JOINT_UNIVERSAL	ART_PARAM_ANCHOR	[x, y, z] : anchor position in world coordinates
	ART_PARAM_AXIS_1	[a1x, a1y, a1z] : axis 1 in world coordinates
	ART_PARAM_AXIS_2	[a2x, a2y, a2z] : axis 2 in world coordinates
JOINT_PRISMROT	ART_PARAM_ANCHOR	[x, y, z] : anchor position in world coordinates
	ART_PARAM_AXIS_1	[a1x, a1y, a1z] : slider axis in world coordinates
	ART_PARAM_AXIS_2	[a2x, a2y, a2z] : rotation axis in world coordinates
JOINT_DBALL	ART_PARAM_ANCHOR_1	[x,y,z] : anchor in body1 in world coordinates
	ART_PARAM_ANCHOR_2	[x,y,z] : anchor in body2 in world coordinates
	ART_PARAM_DISTANCE	set the distance to be kept between anchors manually (instead of the distance at initialization)
Common to all types (optional)	ART_PARAM_ERP	[erp_value] : error reduction parameter for the joint (see ODE manual)
	ARP_PARAM_CFM	[cfm_value] : constraint force mixing for the joint (see ODE manual)

ART_PARAM_CFM and ART_PARAM_ERP have by default the values specified by `simparams_set_ode_params`. All other parameters have the default values specified by ODE (see ODE manual).

For instance, to properly set a hinge articulation, you will need two successive calls of [articulation_set_params](#) :

```
articulation_set_params(artobj, ART_PARAM_ANCHOR, anchor_coords);
articulation_set_params(artobj, ART_PARAM_AXIS, axis_vector);
```

IV.10 - Additional features

IV.10.A - Geometry helpers

These functions are available to perform basic geometric operations on vectors, matrices and quaternions.

Basic vector operations :

```
t_real vector_norm(t_real* vec);
void vector_lincomb_3d(t_real a, t_real* vec1, t_real b, t_real* vec2, t_real* res);
void cross_product_3d(t_real* res, t_real* vec1, t_real* vec2);
t_real dot_product_3d(t_real* vec1, t_real* vec2);
```

Rotation functions :


```
void rotate_vector_using_matrix(t_real* res, t_real* rotmat, t_real* vect);
void rotate_vector_using_quaternion(t_real* vec, t_real* quat, t_real* res);
t_real angle_between_vectors_3d(t_real* vec1, t_real* vec2);
```

The two following functions can be used to express coordinates from/to the world from/to a specific relative frame of reference :

```
void world_to_relative_using_quaternion(t_real* neworigin,
                                       t_real* quat,
                                       t_real x, t_real y, t_real z,
                                       t_real* res);

void relative_to_world_using_quaternion(t_real* obj_pos,
                                       t_real* obj_quat,
                                       t_real x, t_real y, t_real z,
                                       t_real* res);
```

IV.10.B - Custom collision management

cgmdode has a default, builtin interface with the ODE collision system, i.e. a collision callback called by ODE's function `dSpaceCollide` when two geometries are close to eachother. However, if you are familiar with the ODE collision system, cgmdode gives you a way to write your own ODE collision callback. Beware though that it might be cumbersome.

You can set it using the following function :

```
void simcontext_set_custom_collision_callback(simcontext* simcon,
                                             void* custom_callback);
```

The `custom_callback` should point to a function complying with ODE specifications (see the collision section of the ODE manual and the `nearCallback` function). The prototype of the callback should be :

```
void collision_callback(void* data, dGeomID geom1, dGeomID geom2);
```

cgmdode will pass the simcontext object pointer cast to `void*` using the `data` argument. The userdata field of the `geom1` and `geom2` objects are automatically set by cgmdode to the corresponding `physobject*` `pobj1` and `pobj2` pointers. You can use ODE's `dGeomGetData(geom1)` and cast the result to `physobject*` to access them. Once you have proper `physobject*` pointers, you can access ODE's objects using `pobj1->b_id` for the `dBodyID` and `pobj1->g_id` for the `dGeomID` (idem for `pobj2`).

Additionally, you can cast the `data` pointer to `simcontext*` to acces the simcontext object. Through this `simcontext*` pointer, you can access the ODE dynamic world using `((simcontext*)(data))->world_id` (which is a `dWorldID` object). You can also access the collision hashspace `((simcontext*)(data))->space_id` (which is a `dSpaceID` object).

You also have access to cgmdode's simulation parameters using `((simcontext*)(simcon))->simparams` (this internal data structure stores the ODE parameters you specified using `simparams_set_ode_params`).

Here is a snippet to sum it up :


```

/* custom callback */
void collision_callback(void* data, dGeomID geom1, dGeomID geom2)
{
    /* fetch cgmdode objects :*/

    simcontext* simcon = (simcontext*)data;
    simparameters* simparams = simcon->simparams;

    physobject* pobj1 = (physobject*)dGeomGetData(geom1);
    physobject* pobj2 = (physobject*)dGeomGetData(geom2);

    /* ODE objects :*/

    dSpaceID ode_space = simcon->space_id;
    dWorldID ode_world = simcon->world_id;

    dBodyID body1 = pobj1->b_id;
    dBodyID body2 = pobj2->b_id;

    /* the rest of your callback ... */
}

```

Note that if you write your own collision callback, you will need to find a way to keep track of the contact joints created during the simulation, because you must destroy them after each time step. cgmdode keeps track of contact joints using a `dJointGroupID` object in the `simcontext` structure named `contact_joints`. You can use `simcon->contact_joints` to keep track of the contact joints. If you do so, cgmdode will automatically destroy the `contact_joints` group after each timestep, you won't have to do it yourself.

To get more details on how to write your callback, look at the function `collision_callback_generic` in the source file `cgmdode_dynamics.c`

IV.10.C - Random number generation

cgmdode uses the GNU Scientific Library (GSL) pseudorandom number functions. The core pseudorandom number generator is the Mersenne Twister `gsl_mt_19937`. It is initialized by `simcontext_init`, which also requires that `simparams_set_seed` was called beforehand. After those two steps are executed, random number generation is available. You are free to use your own random number generation code.

```

unsigned int ran_int(unsigned long max);

t_real ran_uniform(t_real min, t_real max);

t_real ran_gaussian_01();

t_real ran_gaussian(t_real mean, t_real sigma);

```

`ran_uniform` computes a `t_real` uniform variate in the interval `[min ; max]` using the GSL function `ran_gsl_flat`. `ran_gaussian_01` computes a `t_real` Gaussian variate of standard deviation 1 and mean 0. `ran_gaussian` computes a `t_real` Gaussian variate of standard deviation `sigma` and mean `mean`. Both functions use the GSL `ran_gsl_ziggurat` function, i.e. the Marsaglia-Tsang ziggurat method.

IV.10.D - Caveats regarding boundary conditions

cgmdode automatically manages different boundary conditions at the limit of your simulation box. These are set when you call `simparams_set_medium_params` (see section IV.3.C). There are a few caveats regarding the `INIT_BOUNDARY_<boundary type>` options.

*Articulations **should not be used** with periodic boundary conditions*

With periodic boundary conditions, cgmdode ensure that a `physobject` crossing the world's boundaries will automatically reappear on the symmetric, opposite boundary. Interactions at distance between two `physobjects` located close of two opposites sides of the simulation box are computed and executed. Collisions between two `physobjects` located close to two opposites sides are also treated. cgmdode achieves this feature by maintaining virtual copies of `physobjects` that are close to the simulation box boundaries. This process is transparent for the user, the same simulation can run with any boundary conditions without having to change a single line of code, except the `simparams_set_medium_params` arguments.

However, the case of a physobject connected to another one by an articulation that crosses a boundary whereas its jointed counterpart does not is not treated by cgmdode.

This means that when periodic boundary conditions are enabled, you must not use articulations in your simulation.

If you still do it anyway, the simulation will probably explode if a physobject connected to another crosses the boundary. This is because cgmdode will correct the position of the boundary-crossing physobject so that it reappears at the other side, but will not modify the position of the other non-crossing physobject. This will introduce a huge error in the joint, and therefore ODE will try to make right with a tremendous correction force that will almost certainly result in a kinetic explosion.

The spatial discretization grid when the simulation box is limitless.

Here is a note regarding INIT_BOUNDARY_NONE. With this option, the physobjects of your simulation may go as far away from their starting position and from each other. However, when you invoke `simparams_set_medium_params`, cgmdode asks for explicit spatial boundaries for the simulation. Besides, cgmdode uses a finite spatial discretization grid to speed up the computation of interactions at distance, even in the case where INIT_BOUNDARY_NONE is set. So what happens when physobjects get very far away from the (0,0,0) point of your simulation ?

The spatial discretization grid is centered at (0,0,0). The grid has discrete sizes C_i , C_j , C_k (number of cells) in the three spatial dimensions. The sizes (C_i , C_j , C_k) are set by `simparams_set_medium_params`, using the `max_grid_size` and `min_cell_size` that you specified to compute grid sizes that maximize the number of cells. Regardless of the boundary condition, cgmdode automatically adds an outer layer of cells around the spatial grid (so the actual grid sizes are (C_i+2, C_j+2, C_k+2)). This outer layer is the virtual space in which virtual copies are set when periodic boundary conditions are enabled. But when no boundary conditions are set, any physobject going further than the spatial limits you specified is assigned to the outer layer of the spatial discretization grid. This ensures that the spatial discretization grid covers a virtually infinite volume, but computing interactions is still fast near the (0,0,0) point.