Rocpack
User's Documentation

November, 2013

Guilherme Amadio

Contents

1 Introduction

Rocpack is a particle packing code developed at University of Illinois to simulate the microstructure of solid rocket propellants. Rocpack packs particles based on two different packing algorithms. For spheres, Rocpack uses the consolidated Lubachevsky–Stillinger [1] packing algorithm. For polyhedra, Rocpack uses a new algorithm developed specifically for the task. This new algorithm is a hybrid between the Lubachevsky–Stillinger and Adaptive Shrinking Cell [2] packing algorithms. In both algorithms, infinitesimal particles are distributed randomly within a given container and are then allowed to grow until the desired packing fraction is reached or until the packing is jammed. When two particles are about to collide, measures are taken to prevent any intersections.

Rocpack can produce packs of polydisperse spheres and convex polyhedra. Polyhedra are defined in external files. Rocpack comes with many files for common shapes, including all the platonic and archimedean solids, as well as a few extra shapes for pyramids, prisms, etc. Rocpack can create particles with both fixed size as well as with continuous size distributions.

In the current version of Rocpack, it is possible to pack spheres inside a box (with solid or periodic walls), a cylinder, an annulus, a sphere, or a torus. Polyhedra can be packed only inside a periodic box for the moment, but additional boundary types will be implemented in the future. Figure 1 shows packings of spheres inside all supported boundary types. Figure 2 shows a few packs with polyhedral shapes that come with Rocpack. For more details about the packing algorithm used in Rocpack, please take a look at the file `rocpack.pdf` in the `doc` directory that has been distributed with the source code.

2 Installing Rocpack

After unpacking the source code into a suitable location, the user needs to type the commands below in order to build and install Rocpack. First, build and install the main packing code, by simply typing

```
$ make
$ make install
```

Rocpack's build system has not been migrated to use autotools yet, so the installation of auxiliary tools needs to be done manually, according with what the user needs. First, we need to clean the object files from the current compilation, since some code is shared between the two different packing algorithms:

```
$ make clean
```

Then, we can build the Lubachevsky–Stillinger packing code for spheres and other tools as needed:

```
$ make pack-ls
$ make rdf
$ make pack2bin
$ make pack2mesh
```

After everything has been compiled, we need to move the binaries to a location where our shell will find them. If `$HOME/bin` is not in your `$PATH`, I recommend you to add it, so that you can store your local binaries there and run them from anywhere. Please refer to the manual page of your shell (either `man bash` or `man tcsh`, most commonly) for instructions on how to add a directory to your search path.

```
$ mkdir $HOME/bin
$ cp pack-ls rdf pack2bin pack2mesh $HOME/bin
```

This is all that is needed for the installation. If you run into any problems, make sure that you have OpenGL and gd installed, as these two libraries are needed by Rocpack for rendering and creating screenshots. OpenGL support can optionally be turned off by undefining `USE_OPENGL` in `config.h`.

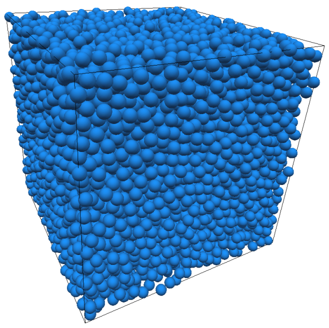
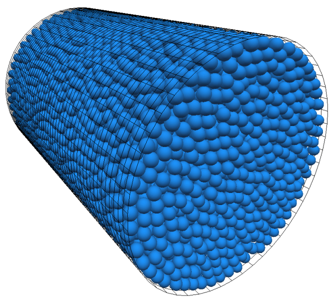
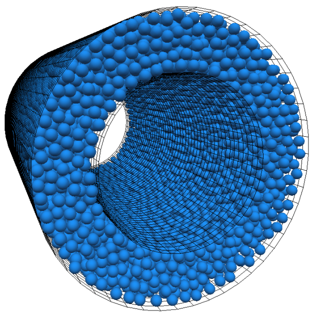
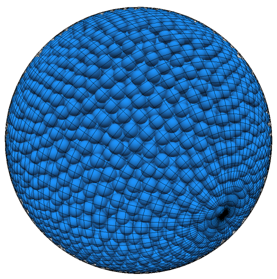
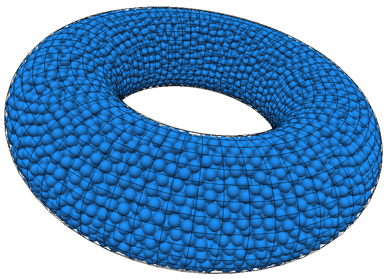

(a) Periodic Box

(b) Cylinder

(c) Annulus

(d) Sphere

(e) Torus

Figure 1: Packs of spheres generated by Rocpack for different boundary types.

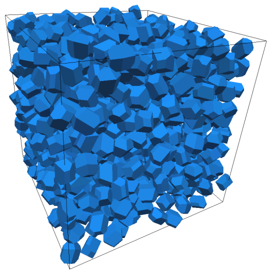
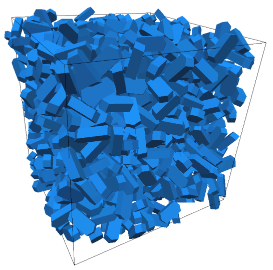
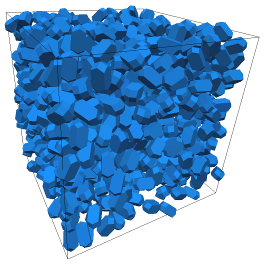
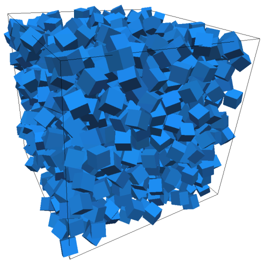
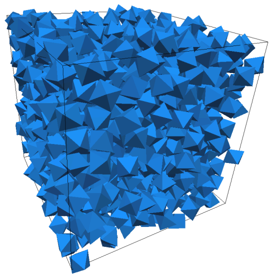
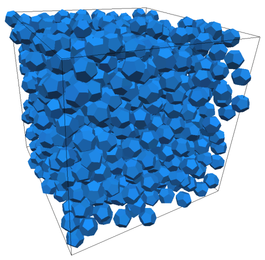
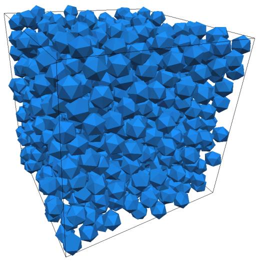
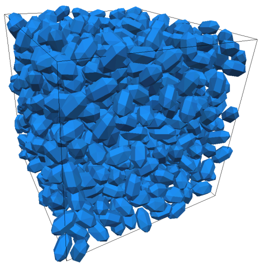

(a) HMX

(b) ADN

(c) PETN

(d) Cubes

(e) Octahedra

(f) Dodecahedra

(g) Icosahedra

(h) Snub Prisms

Figure 2: Packs of polyhedra generated by Rocpack.

3 Running Rocpack

Rocpack is a command line tool with a simple OpenGL interface to present the results of each run. It is possible to watch Rocpack's packing progress in real time, but it is not recommended, as it will slow down packing speed considerably. However, real time rendering is very useful during development, for finding and fixing bugs. Rocpack uses a simple textual input file format described in the next section.

3.1 Input File Format

Rocpack's input file format can be subdivided into four main sections: settings, shape definitions, boundary definition, and particle definitions. Additionally, settings and shapes defined in external files may be imported at the beginning of an input file. Imported files cannot, however, contain any boundary or particle definitions, otherwise there will be a conflict with the definitions in the input file.

3.1.1 Settings

The settings section consists of commands of the form `set parameter = value;` At this early stage, Rocpack only supports setting a few parameters, however. A list of parameters accepted by Rocpack is presented below:

`seed` This parameter is used to set the random seed used by the code. It should be set to an integer number. If set, consecutive runs with the same file will yield the same results. If unset, the default is for the code to use a random seed based on time.

`growth_rate` This parameter can be any positive number, and it determines how fast particles will grow during simulations. All particle sizes in the input file are rescaled such the the largest particle will grow with a rate as set by this parameter. If not set, this parameter defaults to 1.0. It is recommended that the user keeps this parameter between 0.1 and 10.0.

`packing_fraction` This parameter can have a value from 0.0 to 1.0 and sets the desired packing fraction for the run. If the user does not specify a packing fraction, a default value of 1.0 is used. Rocpack's only termination condition is by reaching the desired packing fraction. However, the user can stop the run at any time by pressing `Control-c`. Rocpack will catch the user's intention to terminate the program and write an output file with the current packing information. The user can then use the same output file as a new input file to continue the run from where it was interrupted.

`packing_rate_min` If the the rate of increase in packing density per unit time falls below the value of this parameter, the run is stopped (final result is saved to output file).

`max_runtime` This parameter can be used to set a maximum running time after which the run is stopped (final result is saved to the output file).

hgrid_levels Sets the number of levels of the hierarchical cell grid. Overrides `max_hgrid_levels` if it is also set.

max_hgrid_levels Sets the maximum number of levels that the hierarchical grid may have. Actual value is still computed dynamically based on the size ratio between largest and smallest particles.

window_width Sets the width in pixels of the rendering window.

window_height Sets the height in pixels of the rendering window.

window_size Sets both the width and height of the rendering window.

These are the only parameters that Rocpack uses at the moment. This section will grow as more parameters become available to control the packing process.

3.1.2 Shape Definitions

Rocpack has only spheres as a built-in shape at the moment. Polyhedral shapes must be described either inline in the input file or in external files that are then imported into an input file using the `import` command. The format for a shape definition is very simple. To start a shape definition, the user uses the `shape` keyword, followed by a name for that shape, and the shape definition within braces. The shape definition is given simply by a list of vertices, followed by a list of faces indexed by the list of vertices, starting at zero. The example below shows the definition for a cube:

```
shape cube {
        vertices {
                <-0.5, -0.5,  0.5>, < 0.5, -0.5,  0.5>,
                <-0.5,  0.5,  0.5>, < 0.5,  0.5,  0.5>,
                <-0.5,  0.5, -0.5>, < 0.5,  0.5, -0.5>,
                <-0.5, -0.5, -0.5>, < 0.5, -0.5, -0.5>
        }

        faces {
                [0, 1, 3, 2], [2, 3, 5, 4], [4, 5, 7, 6],
                [6, 7, 1, 0], [1, 7, 5, 3], [6, 0, 2, 4]
        }
}
```

Rocpack comes with 35 ready to use shapes distributed in the `shapes` directory. Some scripts are also provided to facilitate converting shapes from other file formats to be used with Rocpack. Currently, two scripts are provided in the `scripts` folder to convert files in the OFF and OBJ formats.

The website polyHédronisme (http://levskaya.github.io/polyhedronisme) is a good place to create new shapes using Conway's notation for polyhedra. To create a new shape, head over to polyHédronisme, type in the Conway notation for the shape you'd like to create, then save that shape as an OBJ file. For example, the Conway notation **eP3**, for an exploded triangular prism, is a good shape for an elongated crystal. After you save the shape file, run the conversion script on it as shown below:

```
$ obj2shape polihedronisme_eP3.obj exploded_prism
```

This will create a file called `exploded_prism.shape` in Rocpack's format. If the shape name is not provided, the file name is used to give a name to the shape; in this case the shape would be called `polyhedronisme_eP3`. Make sure, however, to give a shape name that contains only alphanumeric characters and underscores. Identifiers in Rocpack input files cannot contain dashes. If the code complains about syntax, simply edit the file and give the shape another name. Also, make sure that you only work with convex shapes. Although it's possible to pack concave shapes generated this way, the intersection algorithm in Rocpack will only consider the convex hull of any shape defined in an external file, so you won't be able to reach high packing fractions.

Converting shapes in the OFF file format is very similar. The command

```
$ off2shape my_shape.off
```

will create a file named `my_shape.shape` that contains the definition for the newly created `my_shape` shape in Rocpack's format. This script is provided to allow conversion of randomly shaped polyhedra produced with QHull (http://qhull.org). Since generation of random shapes using QHull can be a bit complicated, another script is provided with Rocpack to make generation of random polyhedral shapes easy. The script is called `shapegen` and is located in the `scripts` directory, along with the other scripts. To create a random polyhedron from the convex hull of a point distribution containing, say, 30 points, simply type

```
$ shapegen my_shape 30
```

This command will create a `my_shape.shape` file with the resulting shape. You need to have QHull installed to be able to use this script.

Along with polyhedra, Rocpack can also pack cylinders, disks, and ellipsoids. The declaration of these shapes is different than the syntax for polyhedra. For ellipsoids, the shape is declared as

```
shape AP { ellipsoid { A B C } }
```

where A, B, and C are the sizes of each axis. Usually $A = 1.0$, while B and C are less than 1. Cylinders are declared in a similar way, but the parameters are the radius and height, respectively:

```
# R = 1.0 and H = 5.0
shape rod  { cylinder { 1.0 5.0 } }
# R = 1.0 and H = 0.1
shape disk { cylinder { 1.0 0.1 } }
```

Shape definitions do not have to reside in separate files. Shapes can be all put into a single file and imported from the input file, or they can be placed at the beginning of the input file, after the parameter settings. For modeling AP particles with a distribution of ellipsoidal shapes, a file can be created with all ellipsoidal shapes, and the appropriate number of each shape can be calculated from experimental data. An example input file for coarse AP particles is provided in the `examples` directory, along with a file containing all of the ellipsoid shape definitions in the `shapes` directory.

3.1.3 Boundary Types

After the settings and shape definitions, the user can define a boundary shape in which to pack the particles. Each boundary type has its own input syntax. In general terms, the boundary definition is of the form

```
boundary { TYPE PARAMETERS }
```

where `TYPE` is the type of boundary, and the parameters depend on each specific boundary. Each boundary type is described below, along with its input parameters. All parameters are mandatory when defining boundaries.

Box `boundary { box width height depth [solid|periodic] }`
Defines a box boundary with either solid or periodic walls and the given dimensions. The dimensions are mandatory. Box boundaries have solid walls by default.

Cylinder `boundary { cylinder radius height }`
Defines a cylindrical boundary with solid walls with given radius and height.

Annulus `boundary { annulus inner_radius outer_radius height }`
Defines an annular boundary with given inner and outer radii and height.

Sphere `boundary { sphere radius }`
Defines a spherical boundary with given radius.

Torus `boundary { torus major_radius minor_radius }`
Defines a toroidal boundary with given parameters. There is currently no test to determine if the torus doesn't have self-intersections, so the code may yield unexpected results with torii that self-intersect.

If an input file does not contain any boundary specification, a periodic box of unit dimensions is used by default, as this is the most common case.

It's important to remember that when using the `pack` binary to pack polyhedra, only a periodic box is supported at the moment. Other boundary types will produce garbage, as they are not properly implemented yet. If you'd like to make packs on solid boundaries, make sure to use the `pack-ls` binary, for the Lubachevsky-Stillinger packing code. However, please note that collisions against boundaries are computed from the bounding sphere, so it's not possible to pack particles very closely, unless you are packing only spheres. If you need to pack polyhedra in the other boundary types, please wait until they are properly implemented in the new code.

3.1.4 Creating Particles for a Packing

After the optional sections for settings, shape and boundary definitions, the user must specify in this last section what particles should be created for packing. Particles can be created as a group or one by one, altought the syntax for creating particles one by one is only really used to save packs in a format that can be read back in to continue runs. To create particles, the `create` command is used. The syntax of the create command is shown below:

```
create N SHAPE [size SIZE_DEFINITION tag TAG color R G B];
```

Let's now explain what each part of the command means. The command above would create `N` particles of shape `SHAPE`. These particles would be created according to a either a constant size or size distribution, would be tagged by tag number `TAG` and would be shown in the OpenGL interface with a color that has red, green, and blue levels given by `R`, `G`, `B`. The color levels are numbers that vary from 0.0 to 1.0. If a color is given as `color 1.0 0.0 0.0`, for example, the particles would be completely red. If the color is not specified, the program uses a blue hue as the default. In addition, the user can specify `color random` to let Rocpack pick a random color for that particle group. The tag number can be any positive integer number, as it's used in post processing only, when calculating statistics or radial distribution functions. It's recommended, though, to use a sequence beginning from 1. A user would tag each material on a solid propellant, for example, with a different number so that later he could have statistics calculated by material, independently of how many particle sizes and types are used to describe that material. The size definition can be either just a number, in which case all particles in that group will have the same size, or it can be the name of a size distribution, followed by the parameters for that distribution. Currently, the following distributions are implemented in Rocpack:

Uniform `size uniform A B`
: Creates particles with sizes uniformly chosen in the interval $[A, B]$.

Gaussian `size gaussian mean sigma`
: Creates particles with a gaussian distribution with mean `mean` and standard deviation `sigma`.

Gamma `size gamma apha beta`
: Creates particles with a gamma distribution with the given parameters.

Cauchy `size cauchy a b`
: Creates particles with a Cauchy distribution with the given parameters.

Lognormal `size lognormal mu sigma`
: Creates particles with a lognormal distribution with the given parameters.

Weibull `size weibull a b`
: Creates particles with a Weibull distribution with the given parameters.

It's important to mention, though, that distributions like these may create particles at very small sizes that may make the runs very slow. So, for that reason, all distributions will reject particles that are smaller than 0.001 at the moment. This behavior may be changed in the future. New distributions can be easily added to the code.

Please visit, e.g., http://en.wikipedia.org/wiki/List_of_probability_distributions for more information about the distributions.

3.2 Using Rocpack's OpenGL Interface

Rocpack has a simple OpenGL interface that allows the user to visualize the packing either during the run, or at the end of the run. The view of the packing can be rotated by clicking and dragging with the mouse. Some keyboard shortcuts are also available. They are shown in the list below:

i/o, z/Z, +/- Zoom in and out

s, p Take a screenshot / print screen to a file (output goes to a file called `screenshot.png` in the same directory as the program is running).

Esc, q, Q Quit.

The OpenGL window can also be resized with the mouse.

3.3 Command Line Options

Rocpack has a few command line options that control output and viewing of the results on the screen. To get a list of available options, run `pack -help`:

```
Usage: pack [OPTIONS] INPUT_FILE

        -h      --help
        -V      --version
        -v      --verbose
        -q      --quiet
        -d      --draw
        -D      --draw-final
        -f      --fraction      fraction
        -l      --levels        levels
        -o      --output        output
        -s      --seed          seed
```

The options are pretty self-explanatory. The verbose and quiet options control output of progress information to the terminal. The default is for the program to be verbose and show progress both of what the current packing fraction is as well as a percentage of the desired packing fraction. When invoked with the `-d` option, Rocpack will show a real time rendering of the packing process on the screen. Usually, the user will prefer to use the `-D` option to show a rendering of the result only at the end of the run. To save the output to a file, the user must pass a file name to the `-o` option. For example, the following command will create a packing with particle volume fraction of 0.5, show the result on the screen at the end, and save the run to a file:

```
$ pack -D -f 0.5 -o output.pack input.pack
```

The order of the arguments does not matter. The precedence of parameters is such that anything in the command line will override what is in the input file (e.g. the packing fraction as set inside the input file is overriden by the one set in the command line). This is the most typical behavior in UNIX systems, so Rocpack follows these conventions for consistency.

3.4 Continuing a Previously Stopped Run

Output files produced by Rocpack are usable as input files without any changes. When an output file is used as input, packing will resume from where it stopped before. If you have a packing fraction set in the input file, it will not go beyond it. That way, you can use `pack -D` on an output file to view it on screen using OpenGL. You can then take a screenshot or just rotate the pack to see around it.

To stop a run before reaching the specified packing fraction, the user can simply press `Control-c`, which will send a termination signal to the process and the code will stop packing and write the output file and show the result on the screen if those options were passed in. The output file can be used to resume a run as normal. Colors and tags will remain the same in a continuation run and a random color will remain assigned to the same value.

It is also possible to use an output file to create a packing with lower packing fraction from an output file with high packing fraction. The positions and orientations of the particles will be the same as in the higher packing fraction configuration, but the code will stop when the particles grow enough to fill the new packing fraction defined either in the input file or at the command line. This can be used, for example, to create packings in which particles are guaranteed to have small gaps between them. Soon, it will be possible to make packings respecting a given margin between particles set by a parameter in the input file. This margin is currently set to a very small value inside the code to prevent intersections due to floating point rounding and truncation errors.

4 Running Rocpack's Auxiliary Tools

Rocpack comes with a few auxiliary tools for analyzing the results and computing the statistical properties of a packing. We present each tool in a section below.

4.1 Converting a Packing into a Binary Mesh

One of the tools that comes with Rocpack is the `pack2mesh` tool. This program can convert the output file from a Rocpack run into a binary mesh format suitable for being read by finite difference codes, as well as by Rocstat, our auxiliary code to compute two- and three-point statistical functions of microstructures. The `pack2mesh` tool can also be used to create PNG image files of slices through the packings.

To convert an output file from Rocpack called `pack.out` into a binary mesh file called `pack.am` (the `.am` extension stands for Amira, since the files are in Amira's mesh format), a user would issue the following command:

```
$ pack2mesh -l 1024 pack.out pack.am
```

The `-l` (or `-layers`) option is used to tell the code in how many layers it should subdivide the longest dimension of the packing. The default value is 1024 layers. Another option, `-S` (or `-scale`) can be given to the program that will make it ignore the scaling coming from the input file and scale the pack using that factor instead. If a packing is already made, but you used the wrong size in the input file, this option can be useful. The scaling parameter should be set to the bounding radius of the largest particle in the packing.

A similar tool, named `pack2bin` converts rocpack output files to binary files that can be easily imported into Fortran codes. An example code for reading a file converted with this tool is provided in the `examples` directory. The command line arguments for `pack2bin` are similar to those of `pack2mesh`.

Creating slices of packings is done with `pack2mesh`. In this case, you need to specify in what plane to cut the pack. The plane is specified by setting an axis to a constant, e.g. using `x=0.3` will cut a pack in the yz-plane at 30% from the left. For example, to get a cut of the packing in the middle of it's height, a user would issue the command:

```
$ pack2mesh -l 1024 --slice y=0.5 pack.out pack.png
```

Figure 3 shows an example image created with the command above, for a packing of 1000 spheres at 0.5 packing fraction.

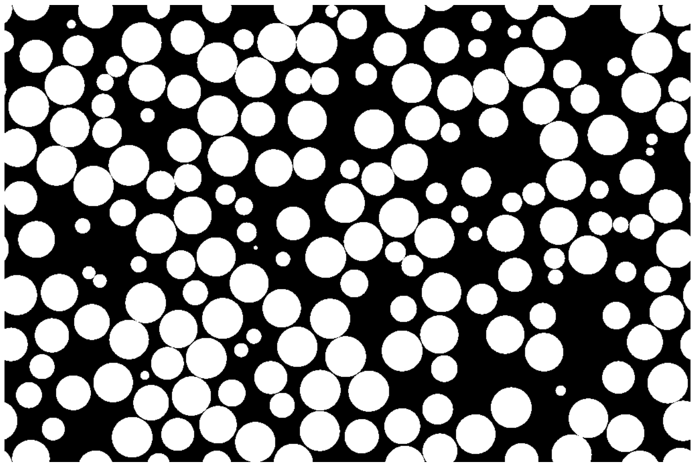


Figure 3: Cross section image of a packing generated with `pack2mesh`

4.2 Computing 2-Point and 3-Point Statistics

Along with Rocpack, we also developed a code to compute two- and three-point statistical functions of microstructures. This tool uses the Amira binary mesh format. Amira (www.amira.com) is a 3D image processing software that we use for post processing rocket propellant sample scans carried out at the University of Illinois. The binary mesh format used by Amira is also the output file format of the X-Ray tomography machine at Beckman Institute. Therefore, it has become our choice of format as well.

Compiling the code is trivial, so we will not cover it in a section of its own. Simply type `make install`, and you'll be done. You will get a `rocstat` binary, and a script called `rocplot`.

The binary `rocstat` can be used to show information about a binary mesh, such as packing fraction, resolution, etc, to compute statistics, to view the pack on the screen, slice by slice, and to create text files that contain slices through the pack. Usage of the code is simple. Running `rocstat -h` will show all the available options and a few examples of how to run the code. We will simply reproduce this output below:

```
rocstat - calculate statistics and create slices of pack files

  Usage: rocstat [options] file.am

  Options:

  -2  calculate two-point statistics
  -3  calculate three-point statistics
  -h  print help and exit
  -i  print info about input file
  -p  calculate packing fraction
  -s  create a slice file
  -v  view file on the screen
  -H  create hotspot locations inside particles (fraction is in
    the range of 0 to 100).
  -B  select the number of particle size bins (default = unset)
  -b  select the number of bins for the statistics calculations
    (default = 100)
  -d  maximum distance in microns for statistics calculations
    (default = 1/2 of the average of the pack dimensions)
  -n  number of random points per bin
    (default = 1000000 for 2-point and 200000 for 3-point statistics).

  Examples:

    Second order statistics:

      $ rocstat -2 -b 50 -d 200 -n 30000 -o DCPD.sij DCPD.am
    Third order statistics:

      $ rocstat -3 -b 100 -d 150 -o DCPD.sijk DCPD.am

    Packing fraction:

      $ rocstat -p DCPD.am

    Create an xy slice file for z=50

      $ rocstat -s xy:50 DCPD.am > slice.dat

    The slice types that the program recognizes are xy, xz and yz,
    always followed by a colon and the number of the slice in the
    missing variable, e.g., yz:10 for a yz slice and x=10.
```

The only thing that the user should be alert to is that the syntax for creating slice files and the output format of the slices are different than those of `pack2mesh`.

The output of `rocstat` is always dimensional, and the output format is very simple. For two-point statistics, the output file has the columns

```
r S00 S01 S11
```

where `r` are the distances between the two points. For three point statistics, the output columns are

```
r1 r2 S000 S001 S010 S011 S100 S101 S110 S111
```

where `r1` and `r2` are the two distances between the central point and the other two points. The angle is constant and is given as a command line parameter.

Rocstat was written a couple of years ago, so it still does not have support for using material tags as set in Rocpack. It can only tell particles from matrix at the moment. However, support for tagging by material will be added soon.

Files created with `rocstat` can be plot by the script `rocplot`, which is written in Python using matplotlib. It is a convenient tool to take a look at the results quickly, but it may also be used to save the plot to a png or a pdf file. To plot results with `rocplot`, the user can issue simple commands as

```
$ rocplot file.sij
$ rocplot file.sij S11
$ rocplot file.sijk
$ rocplot file.sijk S101
```

The first command will show all two-point functions, the second will plot only S_{11} from the given file, and similarly for the three-point statistics.

Rocstat is not distributed with Rocpack. I you need to use it, please request a copy of the source code to either Guilherme Amadio or Tom Jackson.

5 Computing Radial Distribution Functions

The last tool covered in this manual is `rdf`, which is used to compute radial distribution functions. Since this tool was created before Rocpack's current version, it uses the old output file format from D. S. Stafford's code for packing spheres. This format is very simple, so instead of changing the code, a simple script is provided with Rocpack to create these input files. The input file should have a particle per line, with the following columns:

```
x y z r tag
```

where `x`, `y`, and `z` are the coordinates of the centroid of the particle, `r` is the bounding radius, `g` is the growth rate (unused by `rdf`, so you can put anything in this column), and `tag` is the tag number. To compute a radial distribution function, the pack must be in a cubic box of unit dimensions. The boundaries can be either solid or periodic. The output of `rdf` is usually unscaled, however, there is an option to change the scale to the bounding radius of the larger particle. The options to `rdf` are

```
Usage: rdf [options] input_file

  Options:
  -b --bins BINS  set number of bins to BINS (default: 1024)
  -d --max-dist DIST  calculate g(r) in the interval [0:DIST]
                      (default: L/2 for periodic, L/4 otherwise)
  -h --help   print help and exit
  -o --output FILE  save output to FILE (default: stdout)
  -p --periodic   use algorithm for a periodic pack (default: false)
  -s --scaled   output r as r/R, where R = max(ri, rj) (default: false)
  -t --tag  i[:j] calculate g_ii(r) or, if j is set, g_ij(r) (default: 0)
  -v --verbose    show a progress meter of the calculation (default: false)
```

Below are a few examples of how to run `rdf`

```
$ rdf -b 1000 -v -o particles.rdf particles.in
$ rdf -v -p -o particles.rdf particles.in
$ rdf -p -s -t 1:2 -o particles.rdf particles.in
$ rdf -s -d 5 -t 3 -o particles.rdf particles.in
```

In the first example, the radial distribution function g_{00} will be computed with 1000 bins, and the program will show progress information in the terminal. In the second example, defaults are used on a periodic pack, also showing progress. The third example shows how to compute g_{12} instead of the default g_{00}, and the output is scaled by the larger radius between particles tagged with 1 or 2. Finally, the last example shows how to compute the scaled g_{33} only to a maximum distance of 5 radii of particles tagged as 3. The distance defaults to half the width of the pack for periodic packs and a quarter of the width of the pack for packs in boxes with solid walls.

6 Tutorial for a Simple Run

Now that we have described both the input file format and the command line and OpenGL interfaces, we provide here an example of a full run with a few particle shapes and assigned different tags and colors, as well as using the new size distributions. Our input file will contain the shape definitions or imports first. We will use spheres, HMX crystals and a new shape called snub triangular prism, defined as `snub_prism3` in an external file. We will use external shape files, but it's up to the user what his preferences are. There is no difference between defining a shape inline in the input file or importing an external file. It's also possible to keep all shapes in a single file and import that in every run. In our case, we will use the files `hmx.shape`, and `snub_prism3.shape`. Since we are packing polyhedra, we need to use a periodic box, as the other boundaries are currently unsupported for polyhedra. The beginning of our input file is then:

```
set packing_fraction = 0.5;

import ”hmx.shape”
import ”snub_prism3.shape”

boundary { box 1.0 2.0 1.0 periodic }
```

Now we need to create the particles that we want to pack. Let's make their colors random to make things easy, and let's use a gaussian distribution of spheres, a uniform distribution of HMX crystals and all the same size for the other particles. Our full input file is then:

```
set packing_fraction = 0.5;

import ”hmx.shape”
import ”snub_prism3.shape”

boundary { box 1.0 2.0 1.0 periodic }

create 200 sphere size gaussian 1.0 0.05 color random;
create 200 hmx    size uniform  0.6 1.4  color random;
create 200 snub_prism3 size 0.7 color random;
```

We can now run the code and see the results on the screen with the following command:

```
$ pack -D -o output.pack example.pack
```

After the run finishes, we can press `s` on our keyboard to save a screenshot of the run. Figure 4 shows the result. This run took about 10 seconds to finish on my own computer.

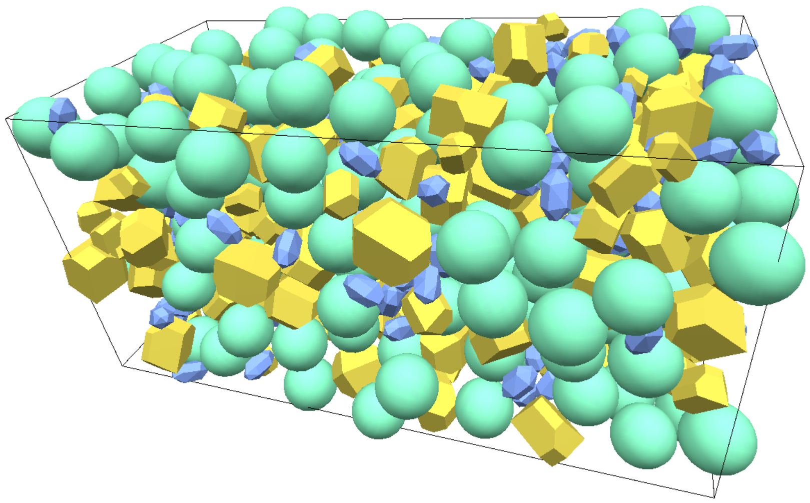


Figure 4: Result of our example run.

References

[1] B. D. Lubachevsky and F. H. Stillinger, Geometric properties of random disk packings, *Journal of Statistical Physics* vol **60** (1990), pages 561–583.

[2] S. Torquato, and Y. Jiao, Dense packings of polyhedra: Platonic and Archimedean solids, *Phys. Rev. E* **80**, 041104 (2009).