Voro++: a three-dimensional Voronoi cell library in C++

Chris H. Rycroft *

January 15, 2009

Contents

1 Introduction **1**

2 Additional code features **3**

3 Getting started and compiling the code **4**

4 Examples of the code **4**

5 Command-line utility **6**
5.1 Command-line arguments 6
5.2 File input and output 6
5.3 Basic command-line options 7
5.4 Command-line options for walls 8

6 Code structure **8**
6.1 The voronoicell class 8
6.1.1 Internal data representation 9
6.2 The container class 10
6.3 Wall computation 11
6.4 Extra functionality via the use of templates 12

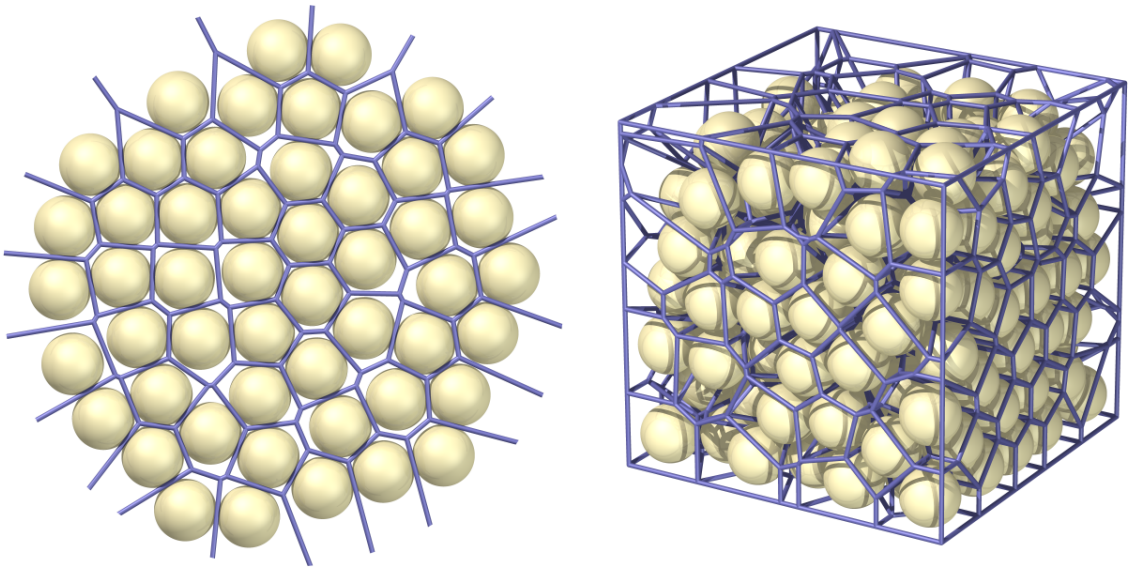
7 Licensing **13**

8 Acknowledgments **14**

1 Introduction

Voro++ is a open source software library for the computation of the Voronoi tessellation, originally proposed by Georgy Voronoi in 1907 [9]. For a set of points in a domain, the tessellation is defined by associating a cell of space to each point, that corresponds to the

*Lawrence Berkeley National Laboratory, Berkeley, CA 94720



(a) (b)

Figure 1: (a) A small sample Voronoi tessellation for a group of particles in two dimensions. The blue lines partition the domain into cells of space that are closer to one particle than any other. The blue lines are the perpendicular bisectors between neighboring particles. (b) A three-dimensional Voronoi tessellation for a small particle packing in a cube. The blue lines show the edges of the Voronoi cells, and were computed using this software package. (Images created with *POV-Ray* [3].)

section of the domain which is closer to that point than any other (Fig. 1). The Voronoi diagram perfectly partitions the domain, and it has a myriad of applications across science in problems that involve allocating space between a group of objects. For a complete discussion, the reader should refer to the book by Okabe *et al.* [6].

Not surprisingly, there are already several mature software projects that compute the Voronoi tessellation. The software package *QHull* [4] can compute Voronoi diagrams in arbitrary numbers of dimensions, making use of an indirect projection method; Matlab's Voronoi routines make use of this package. Another program is *Triangle* [5], which is most well-known for mesh generation via the Delaunay triangulation, but it also computes the Voronoi tessellation. However, this code is specific to two-dimensional computations.

Voro++ makes use of an alternative method of computation, and it aims to be most suited to research problems in materials science, physics, and engineering that frequently involve large systems of particles, often with non-standard boundary conditions. Three key design features are:

- **Cell-based computations** – Some of the existing codes compute the Voronoi diagram as a single object: given a set of points, they will return the complete mesh that divides those points into cells. However, in physical applications it is more natural to associate a single Voronoi cell with each particle, and compute it individually. This perspective makes it easier to compute just a subset of Voronoi cells in a packing, or to tailor the computation to handle special cases and complex boundary conditions. It makes it

straightforward to compute cell-based statistics, such as cell volumes, or the number of faces per cell.

- **Three-dimensional calculation** – Increasingly, particle simulation studies are carried out in three dimensions, and Voro++ is specifically tailored to this case.
- **C++ architecture** – The code is written in object-oriented C++, allowing it to be easily modified and incorporated into other programs. It can carry out calculations using a mix of periodic and non-periodic boundary conditions, and has a general class mechanism for handling different types of walls.

2 Additional code features

As well as carrying out the standard Voronoi tessellation, Voro++ has a number of other features:

- **Radical Voronoi tessellation** – for polydisperse particle systems it is often useful to consider the radical Voronoi tessellation, that weights the cell boundaries according to the relative radii of the particles.
- **Neighbor list computation** – Voro++ can optionally compute neighbor information, storing a list of neighboring particles that created each face of a Voronoi cell.
- **Walls and complex boundary conditions** – Voro++ supports both periodic and non-periodic boundary conditions. It also makes use of an extensible class system for handling boundary conditions due to walls. Plane, spherical, cylinder, and conical walls are supported, but further wall types can be added as derived C++ classes. Curved wall surfaces are approximated using planes.
- **Tolerant algorithms and degenerate vertex support** – Carrying out accurate floating-point calculations is problematic on many systems, and on many popular processors, truncation errors can occur at any time, when numbers are moved from registers to memory. This makes it very hard to guarantee inequalities such as $a < b$ as both a and b may change at any time. Voro++ takes the approach that if two numbers are within a small numerical tolerance, then those numbers are treated as being equal. In the cell-construction process, this can lead to the creation of vertices of arbitrary order, that Voro++ directly supports, allowing it to have perfect representations of shapes such as octahedrons, and icosahedrons.
- **Extensive documentation and examples** – Every routine in the source code is documented, and there are numerous examples provided online of how to use the code. A reference manual is automatically generated from the source code using the *Doxygen* [1] system.
- **Simple extension to parallel computation** – Since each Voronoi cell is computed independently, it is straightforward to parallelize the code to a multicore architecture.

Operating system	Compiler
Mac OS version 10.5 (Leopard)	gcc, version 4.0.1
Mac OS version 10.4 (Tiger)	gcc, version 4.0.0
Cygwin on Microsoft Windows XP	gcc, version 3.4.4
Red Hat Linux 2.4.21	gcc, version 3.2.3
Red Hat Linux 2.6.18	gcc, version 4.1.1
Red Hat Linux 2.6.9	gcc, version 4.2.0
Red Hat Linux 2.6.9	Portland C compiler, version 7.2-3

Table 1: A list of operating systems on which all the Voro++ example programs and the command-line utility have been compiled with no errors or warnings.

3 Getting started and compiling the code

Voro++ is written as platform-independent C++ code, and can be compiled on a variety systems. The latest version of the source code can be obtained as a gzipped tar file from the downloads page on the website at `http://math.lbl.gov/voro++/download/`. The top level directory contains a "README" file describing the code layout.

At present there is no general build system, and the code needs to be compiled directly. The file "config.mk" can be used to configure the compilation. By default, the GNU C++ compiler is used with the configuration flags `-Wall -ansi -pedantic -O3` to provide a high level of optimization and to list as many warnings as possible. Typing `make` in the top level directory should compile the code examples and the command-line utility. The program has been successfully compiled with a variety of architectures and a compilers, shown in table 1. After compilation, the command-line utility should appear in the "bin" directory, and usage of this utility is discussed in more detail in section 5. The example codes are well-documented online, and they are summarized in section 4. Some of the example codes generate output that can be read using the free plotting program *gnuplot* [2] and the freeware raytracer *POV-Ray* [3].

The source code can be freely modified and incorporated into other programs. It is extensively documented, and the software package *Doxygen* [1] is used to generate a complete C++ class reference manual. A LaTeXversion of the manual is provided in the "latex" directory, and an HTML version is provided in the "html" directory. The most up-to-date HTML version of the reference manual is also available on the project website at `http://math.lbl.gov/voro++/doc/refman/`. The *Doxygen* package is freely available, but only needs to be installed if the user wishes to regenerate the manuals.

4 Examples of the code

Perhaps the easiest method of learning the code structure is to look through the examples that are provided in the "examples" directory of the distribution. These codes are all fully commented, and further discussion of each is available online at `http://math.lbl.gov/voro++/examples/`. A list of examples with brief descriptions is given below:

- **Constructing a single Voronoi cell** (basic/single_cell.cc)
  This example introduces the voronoicell class, that represents a single Voronoi cell as a convex polyhedra, represented by a set of vertices and a table of edges. A simple Voronoi cell is built by considering a small set of neighboring particles.

- **The Platonic solids** (basic/platonic.cc)
  This makes use of the voronoicell class to construct the five Platonic solids using plane cuts, outputting the results to text files.

- **The Voronoi diagram for random points in a cube** (basic/random_points.cc)
  The container class is introduced for representing a simulation region. Twenty particles are introduced at random, and a Voronoi cell is constructed for each.

- **Importing particle positions from a text file** (basic/import.cc)
  This makes use of the import() function, to load a particle packing from a text file into the container class. The Voronoi cells are constructed and output using both *POV-Ray* and *gnuplot* formats.

- **A cylindrical particle packing** (walls/cylinder.cc)
  The wall classes are introduced, and used to calculate Voronoi cells in a cylinder.

- **Voronoi cells in a tetrahedron** (walls/tetrahedron.cc)
  Several planar wall objects are used to construct a Voronoi tessellation of some random points in a tetrahedron.

- **Using the cone wall object to create a frustum** (walls/frustum.cc)
  Two planar wall objects and a conical wall object are used to create a frustum (a truncated cone). The volume of the Voronoi cells is tested against the exact frustum volume to test the plane approximation to the curved wall surface.

- **The region that can cut a Voronoi cell** (extra/cut_region.cc)
  This demonstrates the use of the plane_intersects() routine to examine the region of space where an additional particle could be placed to cut a Voronoi cell.

- **Cutting a cell by a grid of points in box** (extra/box_cut.cc)
  This demonstrates the region that can be influenced by a rectangular box of particles.

- **Constructing a superellipsoid** (extra/superellipsoid.cc)
  This is a variation of the single Voronoi cell example that computes a complicated cell approximating a superellipsoid of the form $x^4 + y^4 + z^4 = r^4$.

- **The radical Voronoi tessellation** (radical/radical.cc)
  This compares a radical Voronoi tessellation (carried out with the container_poly class) with a standard Voronoi tessellation (carried out with a container class) using two small sample packings in cube of side length 6.

- **Degenerate vertices** (degenerate/degenerate.cc)
  This demonstrates the ability of the code to handle degenerate vertices with order greater than 3, that occur when plane cuts intersect existing vertices.

- **A complicated degenerate vertex example** (degenerate/degenerate2.cc)
  Many plane cuts are applied by rotating around specific axes to create a shape with many vertices of high order.

- **A timing study using a perl script** (timing/timing_test.pl)
  This perl script can be used to test the speed of the code, by repeatedly compiling and running it with different parameters.

5 Command-line utility

The Voro++ distribution contains a command-line utility that carry out many standard Voronoi calculations. It can be compiled by typing `make` in the top level directory of the distribution, and will appear in the "bin" directory. It can read text files of particle systems, and output versions with Voronoi cell volume appended. The program has the following syntax:

```
voro++ [options] <length_scale> <x_min> <x_max> <y_min>
                 <y_max> <z_min> <z_max> <filename>
```

5.1 Command-line arguments

`<length_scale>` This number should be set to a typical particle length scale in the system, and it is use to configure the code for maximum efficiency. Using a typical particle diameter in the system usually works well. Tuning the value may result in slightly different performance.

`<x_min>` **and** `<x_max>` The minimum and maximum x coordinates of the box.

`<y_min>` **and** `<y_max>` The minimum and maximum y coordinates of the box.

`<z_min>` **and** `<z_max>` The minimum and maximum z coordinates of the box.

`<filename>` The input file containing a list of particles and numerical ID labels.

5.2 File input and output

The input file should have entries on separate lines with the following format:

```
<Numerical ID label> <x> <y> <z>
```

When the command imports the particles, any which lie outside the container geometry are ignored. The program then computes Voronoi cells for all the particles, and generates an output file using the same filename but with a ".vol" suffix, that has the following entries:

```
<Numerical ID label> <x> <y> <z> <Voronoi cell volume>
```

By default, the command assumes non-periodic boundary conditions. The particles in the output file may be ordered differently to those in the input file.

5.3 Basic command-line options

The utility accepts the following options:

`-g` If this option is specified, then an additional output file is generated with the ".gnu" extension, which contains a description of all the cells in a format that can be viewed using *gnuplot* using the splot command. *Caution:* For large input files, the *gnuplot* output file will be extremely large, so this option is best used on smaller systems.

`-h` **or** `--help` This option prints out a summary of the command syntax and the available options.

`-n` This option turns on the neighbor tracking procedure. In each line of the output file, a list of the numerical ID labels is appended that corresponds to the neighboring particles that created each face of the current particle's Voronoi cell. The list can contain negative numbers. For the non-periodic case these correspond to when the particles have faces created by walls. The numbers -1 to -6 correspond to the minimum x, maximum x, minimum y, maximum y, minimum z, and maximum z walls respectively. For periodic boundary conditions, negative numbers correspond to the cases when a face of the Voronoi cell is created by the periodic image of the current particle.

`-p` Make the container periodic in all three coordinate directions.

`-px` Make container periodic in the x direction.

`-py` Make container periodic in the y direction.

`-pz` Make container periodic in the z direction.

`-r` Carry out a Voronoi tessellation for a polydisperse particle arrangement using the radical Voronoi tessellation. For this case, an extra column is required in the input file, that contains the particle radii. The radii are also included in the output file.

5.4 Command-line options for walls

In addition, a number of options can be used to specify wall objects. Walls are implemented by applying extra plane cuts during the cell construction process. At present, four wall types are supported:

`-wc <x1> <x2> <x3> <x4> <x5> <x6> <x7>` Add a cylindrical wall object, where (x_1, x_2, x_3) is a point on the cylinder axis, (x_4, x_5, x_6) is a vector along the cylinder axis, and x_7 is the cylinder radius.

`-wo <x1> <x2> <x3> <x4> <x5> <x6> <x7>` Add a conical wall object, with apex at (x_1, x_2, x_3), axis along (x_4, x_5, x_6), and half angle x_7 (specified in radians).

`-ws <x1> <x2> <x3> <x4>` Add a spherical wall object, centered on (x_1, x_2, x_3), with radius x_4.

`-wp <x1> <x2> <x3> <x4>` Add a plane wall object, with normal (x_1, x_2, x_3), and displacement x_4.

Each wall is accounted for using a single approximating plane – see the cylinder and frustum examples for a complete discussion of this. If neighbor information is requested using the `-n` option, then the walls are numbered sequentially, starting at -7 and decreasing.

6 Code structure

The code is structured around two main C++ classes. The voronoicell class contains all of the routines for constructing a single Voronoi cell. It represents the cell as a collection of vertices that are connected by edges, and there are routines for initializing, making, and outputting the cell, which are discussed in more detail in subsec. 6.1. The container class represents a three-dimensional simulation region into which particles can be added. The class can then carry out a variety of Voronoi calculations by computing cells using the voronoicell class, and this is discussed in more detail in subsec. 6.2. The container class also has a general mechanism using virtual functions to implement walls, which is covered in subsec. 6.3. To implement the radical Voronoi tessellation and the neighbor calculations, two class variants called voronoicell_neighbor and container_poly are provided by making use of templates – this is discussed in subsec. 6.4.

6.1 The voronoicell class

The voronoicell class represents a single Voronoi cell as a convex polyhedron, with a set of vertices that are connected by edges. The class contains a variety of functions that can be used to compute and output the Voronoi cell corresponding to a particular particle. The command init() can be used to initialize a cell as a large rectangular box. The Voronoi cell can then be computed by repeatedly cutting it with planes that correspond to the perpendicular bisectors between that particle and its neighbors.

This is achieved by using the plane() routine, which will recompute the cell's vertices and edges after cutting it with a single plane. This is the key routine in voronoicell class. It begins by exploiting the convexity of the underlying cell, tracing between edges to work out if the cell intersects the cutting plane. If it does not intersect, then the routine immediately exits. Otherwise, it finds an edge or vertex that intersects the plane, and from there, traces out a new face on the cell, recomputing the edge and vertex structure accordingly.

Once the cell is computed, it can be drawn using commands such as draw_gnuplot() and draw_pov(), or its volume can be evaluated using the volume() function. Many more routines are available, and are described in the online reference manual.

6.1.1 Internal data representation

The voronoicell class has a public member p representing the number of vertices. The polyhedral structure of the cell is stored in the following arrays:

- pts[] – an array of floating point numbers, that represent the position vectors $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{p-1}$ of the polyhedron vertices.
- nu[] – the order of each vertex $n_0, n_1, \ldots, n_{p-1}$, corresponding to the number of other vertices to which each is connected.
- ed[][] – a table of edges and relations. For the ith vertex, ed[i] has $2n_i+1$ elements. The first n_i elements are the edges $e(j,i)$, where $e(j,i)$ is the jth neighbor of vertex i. The edges are ordered according to a right-hand rule with respect to an outward-pointing normal. The next n_i elements are the relations $l(j,i)$ which satisfy the property

$$e(l(j,i),e(j,i)) = i.$$

  The final element of the ed[i] list is a back pointer used in memory allocation.

In a very large number of cases, the values of n_i will be 3. This is because the only way that a higher-order vertex can be created in the plane() routine is if the cutting plane perfectly intersects an existing vertex. For random particle arrangements with position vectors specified to double precision this should happen very rarely. A preliminary version of this code was quite successful with only making use of vertices of order 3 [7]. However, when calculating millions of cells, it was found that this approach is not robust, since a single floating point error can invalidate the computation. This can also be a problem for cases featuring crystalline arrangements of particles where the corresponding Voronoi cells may have high-order vertices by construction.

Because of this, Voro++ takes the approach that it if an existing vertex is within a small numerical tolerance of the cutting plane, it is treated as being exactly on the plane, and the polyhedral topology is recomputed accordingly. However, while this improves robustness, it also adds the complexity that n_i may no longer always be 3. This causes memory management to be significantly more complicated, as different vertices require a different number of elements in the ed[][] array. To accommodate this, the voronoicell class allocated edge

memory in a different array called mep[][], in such a way that all vertices of order k are held in mep[k]. If vertex i has order k, then ed[i] points to memory within mep[k]. The array ed[][] is never directly initialized as a two-dimensional array itself, but points at allocations within mep[][]. To the user, it appears as though each row of ed[][] has a different number of elements. When vertices are added or deleted, care must be taken to reorder and reassign elements in these arrays.

During the plane() routine, the code traces around the vertices of the cell, and adds new vertices along edges which intersect the cutting plane to create a new face – additional details of this process are discussed in Ref. [7]. The values of $l(j,i)$ are used in this computation, as when the code is traversing from one vertex on the cell to another, this information allows the code to immediately work out which edge of a vertex points back to the one it came from. As new vertices are created, the $l(j,i)$ are also updated to ensure consistency. To ensure robustness, the plane cutting algorithm should work with any possible combination of vertices which are inside, outside, or exactly on the cutting plane.

Vertices exactly on the cutting plane create some additional computational difficulties. If there are two marginal vertices connected by an existing edge, then it would be possible for duplicate edges to be created between those two vertices, if the plane routine traces along both sides of this edge while constructing the new face. The code recognizes these cases and prevents the double edge from being formed. Another possibility is the formation of vertices of order two or one. At the end of the plane cutting routine, the code checks to see if any of these are present, removing the order one vertices by just deleting them, and removing the order two vertices by connecting the two neighbors of each vertex together. It is possible that the removal of a single low-order vertex could result in the creation of additional low-order vertices, so the process is applied recursively until no more are left.

6.2 The container class

The container class represents a three-dimensional rectangular box of particles. The constructor for this class sets up the coordinate ranges, sets whether each direction is periodic or not, and divides the box into a rectangular subgrid of regions. Particles can be added to the container using the put() command, that adds a particle's position and an integer numerical ID label to the corresponding region. Alternatively, the command import() can be used to read large numbers of particles from a text file.

The key routine in this class is compute_cell(), which makes use of the voronoicell class to construct a Voronoi cell for a specific particle in the container. The basic approach that this function takes is to repeatedly cut the Voronoi cell by planes corresponding neighboring particles, and stop when it recognizes that all the remaining particles in the container are too far away to possibly influence cell's shape. The code makes use of two possible methods for working out when a cell computation is complete:

- **Radius test** – if the maximum distance of a Voronoi cell vertex from the cell center is R, then no particles more than a distance $2R$ away can possibly influence the cell. This a very fast computation to do, but it has no directionality: if the cell extends a

long way in one direction then particles a long distance in other directions will still need to be tested.

- **Region test** – it is possible to test whether a specific region can possibly influence the cell by applying a series of plane tests at the point on the region which is closest to the Voronoi cell center. This is a slower computation to do, but it has directionality.

Another useful observation is that the regions that need to be tested are simply connected, meaning that if a particular region does not need to be tested, then neighboring regions which are further away do not need to be tested.

For maximum efficiency, it was found that a hybrid approach making use of both of the above tests worked well in practice. Radius tests work well for the first few blocks, but switching to region tests after then prevent the code from becoming extremely slow, due to testing over very large spherical shells of particles. The compute_cell() routine therefore takes the following approach:

1. Initialize the voronoicell class to fill the entire computational domain.
2. Cut the cell by any wall objects that have been added to the container.
3. Apply plane cuts to the cell corresponding to the other particles which are within the current particle's region.
4. Test over a pre-computed worklist of neighboring regions, that have been ordered according to the minimum distance away from the particle's position. Apply radius tests after every few regions to see if the calculation can terminate.
5. If the code reaches the end of the worklist, add all the neighboring regions to a new list.
6. Carry out a region test on the first item of the list. If the region needs to be tested, apply the plane() routine for all of its particles, and then add any neighboring regions to the end of the list that need to be tested. Continue until the list has no elements left.

The compute_cell() routine forms the basis of many other routines, such as store_cell_volumes() and draw_cells_gnuplot() that can be used to calculate and draw the cells in the entire container or in a subdomain.

6.3 Wall computation

Wall computations are handled by making use of a pure virtual wall class. Specific wall types are derived from this class, and require the specification of two routines: point_inside() that tests to see if a point is inside a wall or not, and cut_cell() that cuts a cell according to the wall's position. The walls can be added to the container using the add_wall() command, and these are called each time a compute_cell() command is carried out. At present, wall types

for planes, spheres, cylinders, and cones are provided, although custom walls can be added by creating new classes derived from the pure virtual class. Currently all wall types approximate the wall surface with a single plane, which produces some small errors, but generally gives good results for dense particle packings in direct contact with a wall surface. It would be possible to create more accurate walls by making cut_cell() routines that approximate the curved surface with multiple plane cuts.

The wall objects can used for periodic calculations, although to obtain valid results, the walls should also be periodic as well. For example, in a domain that is periodic in the x direction, a cylinder aligned along the x axis could be added. At present, the interior of all wall objects are convex domains, and consequently any superposition of them will be a convex domain also. Carrying out computations in non-convex domains poses some problems, since this could theoretically lead to non-convex Voronoi cells, which the internal data representation of the voronoicell class does not support. For non-convex cases where the wall surfaces feature just a small amount of negative curvature (*eg.* a torus) approximating the curved surface with a single plane cut may give an acceptable level of accuracy. For non-convex cases that feature internal angles, the best strategy may be to decompose the domain into several convex subdomains, carry out a calculation in each, and then add the results together. The voronoicell class cannot be easily modified to handle non-convex cells as this would fundamentally alter the algorithms that it uses, and cases could arise where a single plane cut could create several new faces as opposed to just one.

6.4 Extra functionality via the use of templates

C++ templates are often presented as a mechanism for allowing functions to be coded to work with several different data types. However, they also provide an extremely powerful mechanism for achieving *static polymorphism*, allowing several variations of a program to be compiled from a single source code. Voro++ makes use of templates in order to handle the radical Voronoi tessellation and the neighbor calculations, both of which require only relatively minimal alterations to the main body of code.

The main body of the voronoicell class is written as template named voronoicell_base. Two additional small classes are then written: neighbor_track, which contains small, inlined functions that encapsulate all of the neighbor calculations, and neighbor_none, which contains the same function names left blank. By making use of the typedef command, two classes are then created from the template:

- voronoicell – an instance of voronoicell_base with the neighbor_none class.
- voronoicell_neighbor – an instance of voronoicell_base with the neighbor_track class.

The two classes will be the same, except that the second will get all of the additional neighbor-tracking functionality compiled into it through the neighbor_track class. Since the two instances of the template are created during the compilation, and since all of the functions in neighbor_none and neighbor_track are inlined, there should be no speed overhead with this construction – it should have the same efficiency as writing two completely separate

classes. C++ has other methods for achieving similar results, such as virtual functions and class inheritance, but these are more focused on *dynamic polymorphism*, switching between functionality at run-time, resulting in a drop in performance. This would be particularly apparent in this case, as the neighbor computation code, while small, is heavily integrated into the low-level details of the plane() routine, and a virtual function approach would require a very large number of function address look-ups.

In a similar manner, two small classes called radius_mono and radius_poly are provided. The first contains all routines suitable for calculate the standard Voronoi tessellation associated with a monodisperse particle packing, while the second incorporates variations to carry out the radical Voronoi tessellation associated with a polydisperse particle packing. Two classes are then created via typedef commands:

- container – an instance of container_base with the radius_mono class.
- container_poly – an instance of container_base with the radius_poly class.

The container_poly class accepts an additional variable in the put() command for the particle's radius. These radii are then used to weight the plane positions in the compute_cell() routine.

It should be noted that the underlying template structure is largely hidden from a typical user accessing the library's functionality, and as demonstrated in the examples, the classes listed above behave like regular C++ classes, and can be used in all the same ways. However, the template structure may provide an additional method of customizing the code; for example, an additional radius class could be written to implement a Voronoi tessellation variant.

7 Licensing

This project is free, open-source software, released through the Lawrence Berkeley Laboratory and the US Department of Energy. It is distributed under a modified BSD license, the full text of which is provided with the code. Any questions about licensing should be directed to the LBL Tech Transfer department.

This project has been written by Chris H. Rycroft, a postdoctoral researcher in applied mathematics at the University of California, Berkeley and the Lawrence Berkeley Laboratory, and was developed as part of research into dense granular flow modeling. If you make use of this software in an academic paper, please consider citing either references [8] or [7]. The first reference contains some of the initial images that were made using a very early version of this code, to track small changes in packing fraction in a large particle simulation. The second reference discusses the use of three-dimensional Voronoi cells, and describes the algorithms that were employed in the early version of this code. Since the publication of the above references, the algorithms in Voro++ have been significantly improved, and a paper specifically devoted to the current code architecture will hopefully be published during 2009.

8 Acknowledgments

This work was supported by the Director, Office of Science, Computational and Technology Research, U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] *Doxygen, a source code documentation generator tool*, `http://www.doxygen.org/`.

[2] *gnuplot, a portable command-line driven interactive data and function plotting utility*, `http://www.gnuplot.info`.

[3] *POV-Ray – The Persistence of Vision Raytracer*, `http://www.povray.org`.

[4] *Qhull code for convex hull, Delaunay triangulation, Voronoi diagram, and halfspace intersection about a point*, `http://www.qhull.org/`.

[5] *Triangle: A two-dimensional quality mesh generator and Delaunay triangulator*, `http://www.cs.cmu.edu/~quake/triangle.html`.

[6] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu, *Spatial tessellations: concepts and applications of Voronoi diagrams*, John Wiley & Sons, Inc., New York, NY, 2000.

[7] Chris H. Rycroft, *Multiscale modeling in granular flow*, Ph.D. thesis, Massachusetts Institute of Technology, 2007.

[8] Chris H. Rycroft, Gary S. Grest, James W. Landry, and Martin Z. Bazant, *Analysis of granular flow in a pebble-bed nuclear reactor*, Phys. Rev. E **74** (2006), 021306.

[9] Georgy Voronoi, *Nouvelles applications des paramètres continus à la theorie des formes quadratiques*, Journal für die Reine und Angewandte Mathematik **133** (1907), 97–178.