Finding Vacancies in a Defected Crystal Lattice

George Bargoud

July 13, 2011

Abstract

This program finds vacancies in a defected crystal. This is useful for analysis of various structural properties of the crystal. In this paper the workings of the algorithms used to find the vacancies will be described as well as guidelines for extension and modification and a quick outline of certain important functions.

Contents

1 Important Functions

These are functions that are important to the finding of vacancies. A small description is given here for reference when they are named later.

1.1 extend

`crystal_t * extend(crystal_t crystal);`
This function takes in a crystal and extends it by a small amount on each side. Any atoms that are closest to the bounding faces are copied over to the adjacent parallelipipeds. This ensured that the atoms behave as if they are in a continuous structure rather than a single shape surrounded by nothing.

1.2 voronoi

`vector_t * voronoi(crystal_t crystal,atom_t atom,int * c);`
Determines the Voronoi polyhedron surrounding `atom` in `crystal`. `*c` is where the number of planes is stored. This is done by first determining which planes make up the faces of the Voronoi polyhedron. Each plane is represented by a vector with `atom` at the origin and with the plane orthogonal to it at it's endpoint. The vectors point from `atom` and end at one of the atoms in `crystal`. A new vector $\vec{v}$ is stored in the array `planes` if

$$\forall \vec{p} \in \texttt{planes}$$
$$\vec{v} \cdot \vec{p} < \vec{p} \cdot \vec{p}$$

At the same time, to ensure that any vectors in `planes` that are not supposed to are weeded out:

$$\forall \vec{p} \in \texttt{planes}$$
$$\vec{v} \cdot \vec{p} > \vec{v} \cdot \vec{v} \implies \vec{p} \notin \texttt{planes}$$

1.2.1 Ric

`double Ric(crystal_t crystal,atom_t atom);`
Finds the radius of the inscribed sphere of the Voronoi polyhedron. This is the magnitude of the shortest vector in the array returned by `voronoi`.

1.3 vertices

`vector_t * vertices(crystal_t crystal,atom_t atom,int * c);`
Finds the vertices of the voronoi polyhedron surrounding `atom`. First the planes are found with `voronoi` and then for each set of three planes, the point of intersection is found using the following formula:

$$\forall\ \vec{u}, \vec{v}, \vec{w} \in \texttt{planes}$$
$$\vec{P} = \frac{||\vec{u}||^2(\vec{v} \times \vec{w}) + ||\vec{v}||^2(\vec{w} \times \vec{u}) + ||\vec{w}||^2(\vec{u} \times \vec{v})}{\vec{u} \cdot (\vec{v} \times \vec{w})}$$

Each vertex is then checked against the other plane vectors to make sure that it is actually a vector in the voronoi shell, this is done in the same way that vectors are tested when creating the shell.

1.3.1 Rcc

`double Rcc(crystal_t crystal,atom_t atom);`
The radius of the curcumsphere of the voronoi shell is equal to the distance to the farthest point on it. Because the farthest point is a vertex, first the vertices are found using the `vertices` function, then their magnitudes are compared to find the farthest one.

1.4 bringin

```
void bringin(atom_t * atom,crystal_t crys);
```
If the atom is not within the bounds of the Bravais lattice of the crystal, then it will be moved along the Bravais vectors and brought within the bounds. This is useful for drift to allow for wraparound and therefore an easier time finding the best position.

1.5 drift

e `atom_t drift(atom_t atom,crystal_t crystal, double (*func)(crystal_t,atom_t), double d,double dmin);`

Find the local maxima of `func` in the crystal starting at the point where `atom` is and moving in steps starting with `d` times the atom's radius and shrinking by a factor of 10 each time an approximate maxima is found until the step size is `dmin`. First direction is determined by calling `grad` which returns an approximate unit vector of the gradient of `func` at this point. The atom is moved a distance in that direction and this loop continues until either 10000 iterations have occured or the new value of `func` is lower than the old one meaning that a local maximum has been reached. After that, the distance is divided by 10 and the local maximum is looked for now with better resolution. This is repeated until the distance is less than `dmin`. The following functions are the ones that `drift` is used on in the program:

1.5.1 closest

```
double closest(crystal_t crystal,atom_t atom);
```
Finds the distance to the closest atom in the crystal as measured from the edge of the atom's sphere. This can be negative if the edges overlap.

1.5.2 r_c_ratio

```
double r_c_ratio(crystal_t crystal,atom_t atom);
```
Returns the ratio of `closest_c` (the distance to the closest atom, measured from the center) to `Rcc` (see above). This value must be greater than 1 for the atom's position to be allowed.

2 Filling Algorithms

Every filling algorithm so far first uses `extend` to get an extended version of the crystal and through it treat edges the same way as the middle.

2.1 gridfill

This algorithm creates a grid with the axes of the Bravais vectors and then calls `drift` using the points on the grid as starting points. Each cell in the grid is $1.95\times$`radius` across in the direction of each Bravais vector. After `drift` has been called with a starting resolution of 0.1 and ending at 10^{-10}, the atom is kept if it does not overlap with any other atoms and then drifted using `r_c_ratio`. It is kept after this if `Rcc` is greater than `closest_c`. When multithreaded, each thread is assigned a different section of the crystal and they all send the atoms that they found to the thread of rank 0 at the end.

3 Adding New Filling Algorithms

In this section are rules and conventions used in the filling algorithms. Because of the way `fill.h` and `fill.c` are structured, `fill.o` can be replaced at linking with any file that contains a `funcmap` array and `#include "gapslib.h"` for external variables and useful functions, typedefs and macros.

3.1 Types

The filling algorithms take in one argument and have no return value. The argument is a pointer to the crystal which is to be filled. The function prototypes for the filling algorithms are in `fill.h`.

3.2 funcmap

```
struct {
    char * name;
    fill_t func;
} funcmap[] = {
    {"gridfill"  ,gridfill  },
    {NULL,NULL}
};
```

In addition to the function prototypes, `fill.h` contains an array of structs called `funcmap`. The struct in the array consists of a `char *` and a `fill_t` field. `fill_t` is defined in `gapslib.h` as

```
typedef void (*fill_t)(crystal_t * crystal);
```

which is a function pointer to a filling algorithm. It should have an entry for each filling algorithm and another entry consisting of `{NULL,NULL}` to signal the end of the array. The string is the name by which the algorithm is designated on the command line and the pointer is a pointer to that algorithm. The default is the first element in the array.

3.3 Multithreading

If the program is run with multiple threads using `mpiexec`, the thread with rank 0 will receive the pointer to the crystal and all others will have a `NULL` in stead. This means that at the end, the thread with rank 0 is the one to which all vacancies should be sent.

4 Documentation

Documentation for individual functions can be found in short form in the header files and in longer form in the `.c` files. A brief overview of what can be found in each file is located in `README` and for help with usage, a man page is included. To preview the man page before installation, use the command "`make man`". After installing, use "`man 1 gaps`".