Efficient evaluation of interatomic distances in large atomic scale models [Artivle v1.0]

Sébastien Le Roux

Abstract This article describes how lattice mathematical properties can be combined with the 3D space pixelation method to evaluate interatomic distances in large atomic scale 3D models. It aims to be an educative tool for students and researchers who want to develop structural analysis or 3D visualization tools that requires to implement and compute efficiently interatomic bond distances. Examples codes are provided in C, FORTRAN90 and Python.

*For correspondence:
sebastien.leroux@ipcms.unistra.fr (
)

[†]This author is the sole author of this work

1 Introduction

Every student, every researcher in computational material science, has already spent time calculating interatomic distances. This problem is even likely to be the first one computational material scientists will spend some time over during their studies. As simple as the evaluation of a distance in 3D space could seem to be, the complexity of the problem increases considerably when dealing with the periodicity of non-cubic systems, and even more with the search for performance that is driving the analysis and the visualization of atomic scale models with more than tens of thousands of atoms.

This manuscript illustrates how lattice mathematical properties can be combined with the pixelation of the model box approach, to offer both a general, symmetry independent, methodology, and, an extremely efficient implementation of the search for first neighbor atoms.

2 Simulation box, lattice parameters and transformation matrices

Knowledge and understanding of the mathematics of lattice parameters and atomic coordinates is a prerequisite to the general formulation of the calculation of interatomic bond distances in 3D atomic scale models using periodic boundary conditions.

Note that this section can safely be ignored when dealing with non periodic systems.

2.1 Simulation box or lattice parameters

Box, or lattice parameters can be expressed with two different sets of parameters, using:

1. Using the box parameters A, B, C and the associated angles α, β and γ
2. Using the components of the lattice vectors $\vec{a}(a_x, a_y, a_z)$, $\vec{b}(b_x, b_y, b_z)$ and $\vec{c}(c_x, c_y, c_z)$

Then lattice vectors (2.1) can be calculated using box parameters (2.1) with:

$$\begin{bmatrix} A & 0.0 & 0.0 \\ \times \cos\gamma & B \times \sin\gamma & 0.0 \\ C \times \cos\beta & C \times L & C \times L^2 \end{bmatrix} \qquad (1)$$

With:

$$L = \frac{\cos\alpha - \cos\beta \times \cos\gamma}{\sin\gamma} \qquad (2)$$

While box parameters (2.1) can be calculated using lattice vectors (2.1) with:

$$A = |\vec{a}| \quad B = |\vec{b}| \quad C = |\vec{c}| \qquad (3)$$

and:

$$\alpha = \frac{\vec{c} \cdot \vec{b}}{B \times C} \quad \beta = \frac{\vec{a} \cdot \vec{c}}{A \times C} \quad \gamma = \frac{\vec{a} \cdot \vec{b}}{A \times B} \qquad (4)$$

The lattice volume:

$$V = \vec{a} \cdot (\vec{b} \wedge \vec{c}) = \vec{b} \cdot (\vec{c} \wedge \vec{a}) = \vec{c} \cdot (\vec{a} \wedge \vec{b}) \qquad (5)$$

can then be calculated using:

$$V = A \times B \times C \times Z \qquad (6)$$

With:

$$Z = \sqrt{1 - \cos^2\alpha - \cos^2\beta - \cos^2\gamma + 2\cos\alpha \cos\beta \cos\gamma} \qquad (7)$$

Knowledge of these properties is a basic requirement, from there it possible to compute transformation matrices that allow the conversion from Cartesian r to Fractional f coordinates and the conversion from Fractional to Cartesian coordinates. These mathematical tools are extremely useful, if not almost mandatory prerequisites to the calculation, when dealing with non-cubic periodic systems.

2.2 From Cartesian to fractional coordinates

For an atom with Cartesian coordinates (r_x, r_y, r_z), fractional coordinates (f_x, f_y, f_z) can be calculated using:

$$\begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix} = T_f \times \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix} \qquad (8)$$

Where the transformation matrix T_f is defined as:

$$T_f = \begin{bmatrix} \frac{1}{A} & -\frac{\cos\gamma}{A\sin\gamma} & \frac{\cos\alpha\cos\gamma - \cos\beta}{AZ\sin\gamma} \\ 0.0 & \frac{1}{B\sin\gamma} & \frac{\cos\alpha\cos\gamma - \cos\beta}{BZ} \\ 0.0 & 0.0 & \frac{\sin\gamma}{CZ} \end{bmatrix} \qquad (9)$$

2.3 From fractional to Cartesian coordinates

Similarly fractional coordinates (f_x, f_y, f_z) can be converted to Cartesian coordinates (r_x, r_y, r_z) using:

$$\begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix} = T_c \times \begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix} \qquad (10)$$

Where the transformation matrix T_c is defined as:

$$T_c = T_f^{-1} = \begin{bmatrix} A & B\cos\gamma & C\cos\beta \\ 0.0 & B\sin\gamma & C\frac{\cos\alpha - \cos\beta\cos\gamma}{\sin\gamma} \\ 0.0 & 0.0 & \frac{CZ}{\sin\gamma} \end{bmatrix} \qquad (11)$$

# 3 Pixelation of the model box

The idea of pixelation, or partitioning, of the model box illustrated in this section is mandatory to deal efficiently with searching for neighbor atoms in large atomic scale models. Indeed the intuitive way to implement the procedure would be to test every pair $i - j$ of atoms in the model: compute the interatomic distance D_{ij} between i (α) and j (β), and then compared this distance to a cutoff radius $R_{cut}(\alpha, \beta)$, that could appropriately be determined when looking at the radial distribution function $g_{\alpha\beta}(r)$. Then if D_{ij} is smaller or equal to $R_{cut}(\alpha, \beta)$ then atoms i and j are first neighbors, otherwise they are not. For a program which purpose is to render the atomic scale model in 3D space, the result of the analysis would be then to draw, or not, a bond between atoms i and j.

As intuitive and logical as this approach could seem to be, it requires to perform the testing for every pair of atoms in the model. Which, as long as the size of the system remains within the thousand or few thousands of atoms, could work in a seemingly efficient manner. The time order for the entire analysis is then proportional to $\frac{N \times (N-1)}{2}$, with N the total number of atoms in the model. However as the number of atoms increases, time required to performed the entire analysis increases even more dramatically, soon enough reaching a point where the program will likely seem to be completely frozen.

Therefore a step is required to optimize the procedure for large atomic scale models, and that is to distinguish atom(s) that are of interest for the purpose of the calculation from atom(s) that are not. This is done by dividing, or partitioning, the model box in smaller parts, or pixels.

Atomic coordinates will allow to associate an atom to a particular pixel. Then for this particular atom neighbors candidates will only be search for in that same pixel and its immediate surrounding pixel neighbors. The pixel dimension $a = R_{cut}$,
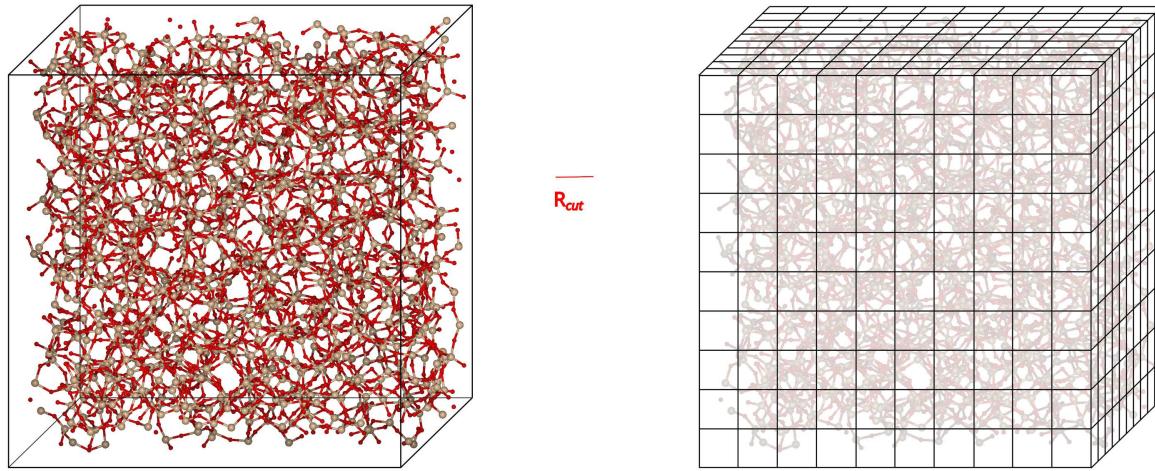
Figure 1. Pixelation of the model box

with R$_{cut}$ is the cutoff radius used to search for first neighbors atoms.

This approach will limit the area of interest to the smallest possible size. Using this methodology the time order for the entire analysis becomes proportional to $N_p \times \frac{N_c \times (N_c-1)}{2}$, with N_p is the total number of pixels in the grid, and N_c is the average number of atom(s) in the pixel and its 26 surrounding neighbor pixels: $N_c <<< N_p << N$.

The idea behind this approach is illustrated in figure 1.

3.1 Without Periodic Boundary Conditions

The number of pixels on each axis, $n_p(x)$, $n_p(y)$ and $n_p(z)$, are calculated using:

$$n_p(axis) = \left\lceil \frac{D_{max}(axis)}{R_{cut}} \right\rceil \qquad (12)$$

Where $D_{max}(axis)$ is the maximum interatomic distance separating two atoms on $axis$, and R$_{cut}$ is the cutoff distance that separates neighbor atoms.

Providing a model box, with parameters A, B, and C, encompassing the entire model could prove useful here, allowing the simplifications:

$$D_{max}(x) = A, \qquad D_{max}(y) = B \qquad \text{and} \qquad D_{max}(z) = C \qquad (13)$$

Otherwise calculations to determine D_{max} for each axis are needed. It is then required to test each pair of atomic coordinates in the model on x, y and z. However as long as only subtractions and min/max comparisons are involved calculation time will remain acceptable.

Then the total number of pixels in the model box, $pixels$, is calculated using:

$$pixels = n_p(x) \times n_p(y) \times n_p(z) \qquad (14)$$

For an atom at with Cartesian coordinates (r_x, r_y, r_z) in the model, corresponding pixel indices in the pixel grid (p_x, p_y, p_z) can be calculated using:

$$p_{axis} = \left\lfloor \frac{r_{axis} - min_{axis}}{R_{cut}} \right\rfloor \qquad (15)$$

With:

$$p_{axis} \in [0, n_p(axis) - 1] \qquad (16)$$

Where min_{axis} in the lowest value for any atomic coordinates in the model on $axis$.

The pixel number for at, between 0 and $pixels - 1$, in entire the pixel grid, P_{id} is calculated using:

$$P_{id}(at) = p_x + n_p(x) \times p_y + \left[n_p(x) \times n_p(y) \right] \times p_z \qquad (17)$$

With:
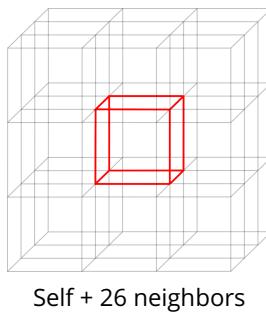
$$P_{id} \in [0, pixels - 1] \qquad (18)$$

First neighbors list is calculated as follow:

1. The pixel position (p_x, p_y, p_z) for every atom (r_x, r_y, r_z) in the model is to be calculated so that each atom can be assigned a pixel number in the grid.

2. Accordingly a list of atom containing pixels is created, the list of atom(s) in each pixel being stored.

3. First neighbor(s) for an atom will then be search for in the pixel this atom belong to and in its surrounding pixel neighbors only, all other pixel(s) being safely ignored.

The list of pixel neighbors is constructed using the mathematical relationships between each pixel index in the grid, two cases must considered:
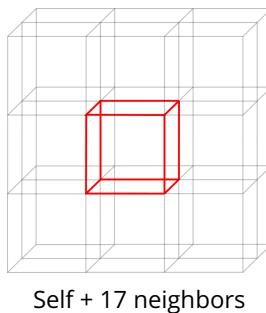
– Pixel inside the pixel grid:
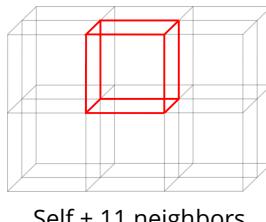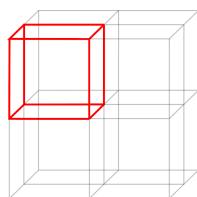
Self + 26 neighbors

– Pixel on the boundary of the pixel grid :
$$p_{axis} = 0 \text{ or } p_{axis} = n_p(axis) - 1$$

Face of the pixel grid

Self + 17 neighbors

Edge of the pixel grid

Self + 11 neighbors

Corner of the pixel grid

Self + 7 neighbors

Pixel neighbors are determined as illustrated in figure 2:

Note that this kind of approach only makes sense if the number of pixels in the grid is high enough so that all pixels are not neighbors.

3.2 With Periodic Boundary Conditions

Non-cubic symmetries make it more complicated to offer a general methodology to deal with periodic systems:

1. Evaluate the number of pixel(s) on each axis, $n_p(x)$, $n_p(y)$ and $n_p(z)$, using equations 12 and 13.

2. Convert atomic Cartesian coordinates to fractional coordinates using $\mathbf{T_f}$.
   Using the transformation to fractional coordinates is the easiest way to compute the distance between atoms in the model. The problem requires to consider the periodicity of the system, and, in the case of non-cubic symmetry transformations, could be tricky when using Cartesian coordinates. Working with fractional coordinates is much easier since in that case corrections are performed simply adding or subtracting multiples of 1.0 on any fractional direction.

   – Convert Cartesian coordinates to fractional coordinates (f_x, f_y, f_z) using Eq. 8.
   – Compute corrected fractional coordinates ($f_{c,x}$, $f_{c,y}$, $f_{c,z}$) inside the model box:
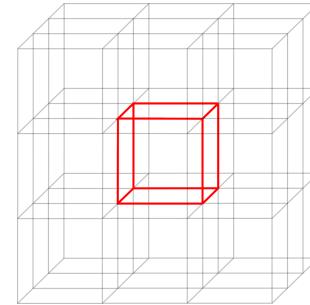
   $$f_{c,axis} = f_{axis} - \lfloor f_{axis} \rfloor \qquad \text{with} \qquad 0 \le f_{c,axis} < 1 \quad (19)$$

3. Pixel positions (p_x, p_y, p_z) are determined using the atom's corrected fractional coordinates ($f_{c,x}$, $f_{c,y}$, $f_{c,z}$):

 $$p_{axis} = \lfloor f_{c,axis} \times n_p(axis) \rfloor \qquad \text{with} \qquad p_{axis} \in [0, n_p(axis) - 1] \quad (20)$$

4. Determine each pixel neighbors:

   – Pixel inside the pixel grid:

Self + 26 neighbors, no PBC transformation required.

$\mathbf{P_{id}+n_{xy}+n_x-1}$ $\mathbf{P_{id}+n_{xy}+n_x}$ $\mathbf{P_{id}+n_{xy}+n_x+1}$

$\mathbf{P_{id}+n_x-1}$ $\mathbf{P_{id}+n_x}$ $\mathbf{P_{id}+n_x+1}$

$\mathbf{P_{id}-n_{xy}+n_x-1}$ $\mathbf{P_{id}-n_{xy}+n_x}$ $\mathbf{P_{id}-n_{xy}+n_x+1}$

$\mathbf{P_{id}+n_{xy}-1}$ $\mathbf{P_{id}+n_{xy}}$ $\mathbf{P_{id}+n_{xy}+1}$

$\mathbf{P_{id}-1}$ $\mathbf{P_{id}}$ $\mathbf{P_{id}+1}$

$\mathbf{P_{id}-n_{xy}-1}$ $\mathbf{P_{id}-n_{xy}}$ $\mathbf{P_{id}-n_{xy}+1}$

$\mathbf{P_{id}+n_{xy}-n_x-1}$ $\mathbf{P_{id}+n_{xy}-n_x}$ $\mathbf{P_{id}+n_{xy}-n_x+1}$

$\mathbf{P_{id}-n_x-1}$ $\mathbf{P_{id}-n_x}$ $\mathbf{P_{id}-n_x+1}$

$\mathbf{P_{id}-n_{xy}-n_x-1}$ $\mathbf{P_{id}-n_{xy}-n_x}$ $\mathbf{P_{id}-n_{xy}-n_x+1}$

Z

Y   X

```
With:
    n_x = number of pixel(s) on x
    n_y = number of pixel(s) on y

And:
    n_xy = n_x X n_y
```
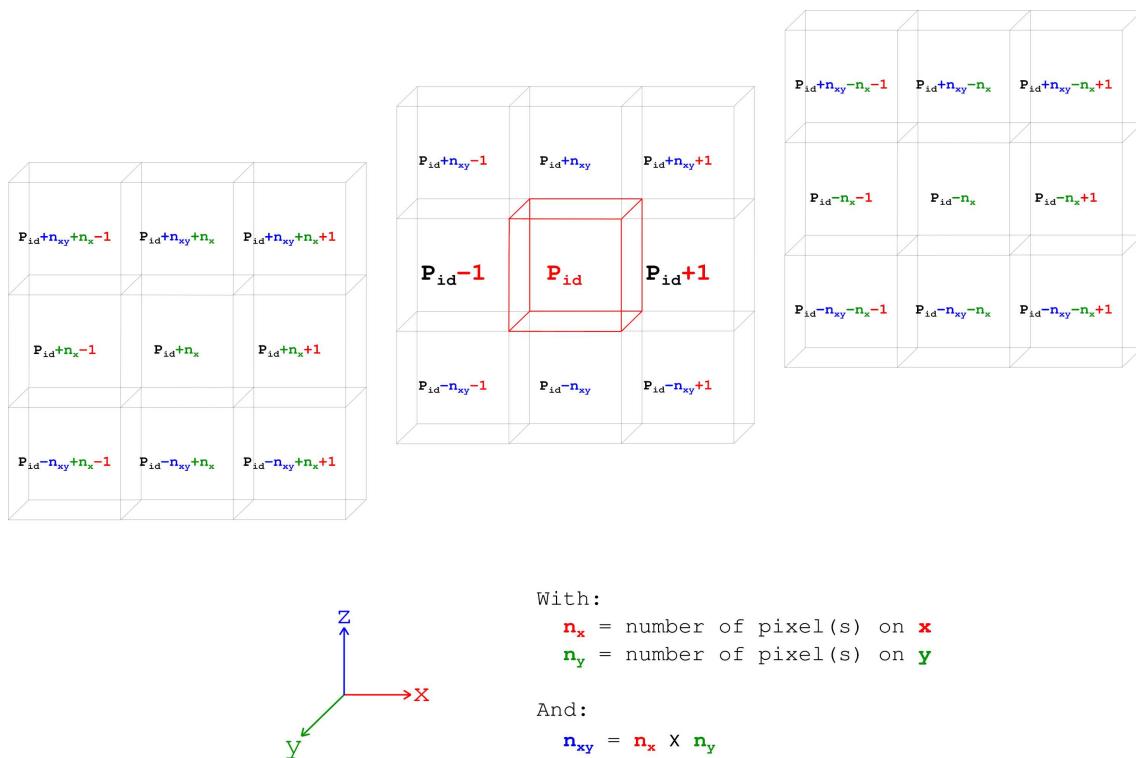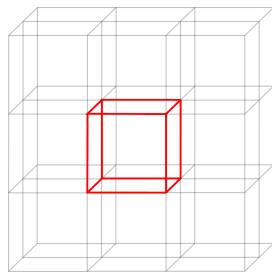
Figure 2. Finding pixel neighbors for pixel P_{id}: operation(s) on each axis are illustrated in the appropriate color(s)
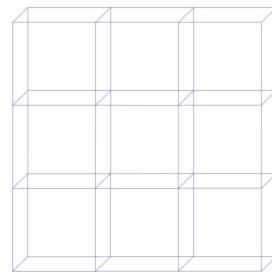
– Pixel on the boundary of the pixel grid :

$p_{axis} = 0$ or $p_{axis} = n_p(axis) - 1$

Face of the pixel grid:

+
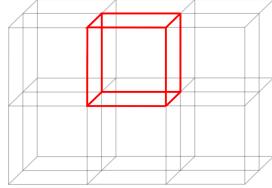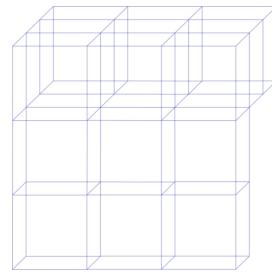
Self + 17 + 9 neighbors using PBC

Edge of the pixel grid:

+
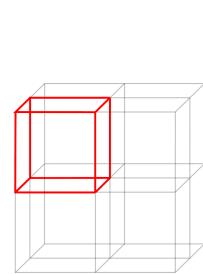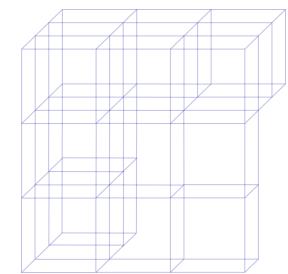
Self + 11 + 15 neighbors using PBC

Corner of the pixel grid:

+

Self + 7 + 19 neighbors using PBC

For a pixel with number p_{id} with pixel coordinates (p_x, p_y, p_z) inside the pixel grid, pixel neighbors are determined as illustrated in figure 2. For pixel on the boundary of the grid, then adjustments are required to find the neighbors via PBC.

5. Compute the interatomic distance D_{ab} between 2 atoms a and b using corrected fractional coordinates:

$$f_{c,axis}(ab) = f_{c,axis}(a) - f_{c,axis}(b) \tag{21}$$

$$F_{axis}(ab) = \|f_{c,axis}(ab)\| \tag{22}$$

$$Fmin_{axis}(ab) = \min\left[F_{axis}(ab), i\, 1.0 - F_{axis}(ab)\right] \tag{23}$$

$$f_{axis}(ab) = \frac{f_{c,axis}(ab)}{F_{axis}(ab)} \times Fmin_{axis}(ab) \tag{24}$$

$$\vec{r}(ab) = \mathbf{T_c} \times \vec{f}(ab) \tag{25}$$

$$\text{with} \quad \vec{r}(ab) = \begin{pmatrix} r_x(ab) \\ r_y(ab) \\ r_z(ab) \end{pmatrix} \tag{26}$$

$$\text{and} \quad \vec{f}(ab) = \begin{pmatrix} f_x(ab) \\ f_y(ab) \\ f_z(ab) \end{pmatrix} \tag{27}$$

$$D(ab) = |\vec{r}(ab)| \tag{28}$$

As mentioned in the previous section this approach only makes sense when the number of pixels in the grid is high enough so that all pixels are not neighbors. In the case where PBC are applied this means that the number of pixels on one dimension, x, y, or z should be higher than 3.

Commented codes that illustrate the entire work procedure are provided in:

- C codes: A.1 through A.7.
- FORTRAN90 codes: B.1 through B.7.
- Python codes: C.1 through C.7.
  Note that this Python code is provided to illustrate the entire implementation in Python, but for that particular programming language several Python libraries already exist and can be used as fronted to simplify the complete coding (ex: ASE, Pysic, MDAnalysis), however in that case the pixelation approach is rarely coded in pure Python language (Pysic using Fortran, MDAnalysis using C).

4 Further optimizations

The analysis time can be reduced using MPI and/or OpenMP parallel programming. Several scenario, or approaches, can be envisioned depending on the size of the system in number of atoms and/or the number of configuration (MD steps):

- Single (MPI or OpenMP): atomic coordinates or pixels can be distributed over the CPU and/or CPU cores.
- Multiples (MPI or OpenMP): configurations can be distributed over the CPU and/or the CPU cores.

- Multiples (MPI and OpenMP): with hybrid parallelization configurations can be distributed over the CPU, and atomic coordinates or pixels can be distributed over the CPU cores.

Ideally the code would provide the option to switch to one or the other approach based on the number of configurations and or atoms in the system.

Note that this is the case of the **atomes** software [1] that implements an adaptive OpenMP programming distributing either the atomic coordinates or the MD steps on the CPU cores.

5 Conclusion

The general methodology behind efficient first neighbor(s) analysis in any kind of atomic scale models was described.

It was illustrated that the understanding of lattice mathematics can simplify the evaluation of interatomic bond distances independently of the periodicity of the system, and that the particular idea of the pixelation, or partitioning, of the model box, is a prerequisite to any modern implementation of this analysis.

The methodoly described in this article is the one implemented in the **atomes** software [1].

Checklist

For a more detailed description of author contributions, see the GitHub issue tracking and changelog at https://github.com/Slookeur/Bonds.

Potentially Conflicting Interests

No conflicting interests.

Author Information

ORCID:
Sébastien Le Roux: 0000-0002-1912-6960

References

[1] **Le Roux S**. Comp. Mat. Sci.. 2025; 253:113805. https://doi.org/10.1016/j.commatsci.2025.113805.

SEARCHING FOR FIRST NEIGHBOR ATOMS

Non-periodic systems

☐ Choose a cutoff radius R_{cut} to determine first neighor atoms
☐ Find the maximum distances, $D_{max}(x)$, $D_{max}(y)$ and $D_{max}(z)$ that separate 2 atoms on x, y and z
☐ Determine the number of pixels, $n_p(axis)$, on x, y and z using $n_p(axis) = \left\lceil \frac{D_{max}(axis)}{R_{cut}} \right\rceil$
☐ Create a pixel grid for a virtual box with dimensions $n_p(x)$, $n_p(y)$ and $n_p(z)$ on x, y and z axis respectively.
☐ Using its Cartesian coordinates, r_{axis}, associate each atom at with a pixel position in the grid

  ☐ Pixel position on each $axis$: $p_{axis} = \left\lfloor \frac{r_{axis} - min_{axis}}{R_{cut}} \right\rfloor$
  ☐ Pixel ID number in the grid: $P_{id}(at) = p_x + n_p(x) \times p_y + \left[n_p(x) \times n_p(y) \right] \times p_z$

☐ Associate each pixel with it surrouding neighbors in the grid
☐ Search for bond candidates between the atoms of a pixel as well as the atoms of its surrouding pixel neighbors
☐ Evaluate distances using the atom's Cartesian coordinates

Periodic systems

☐ Choose a cutoff radius R_{cut} to determine first neighor atoms
☐ Use the lattice parameters as maximum interatomic distances on x, y and z: $D_{max}(x) = A$, $D_{max}(y) = B$ and $D_{max}(z) = C$
☐ Determine the number of pixels, $n_p(axis)$, on x, y and z using $n_p(axis) = \left\lceil \frac{D_{max}(axis)}{R_{cut}} \right\rceil$
☐ Create a pixel grid with dimensions $n_p(x)$, $n_p(y)$ and $n_p(z)$ on x, y and z axis respectively.
☐ Convert all atom Cartesian coordinates to fractional coordinates f
☐ Compute the corrected fractional coordinates f_c to ensure that all are in the unit cell
☐ Using its correct fractional coordinates, $f_{c,axis}$, associate each atom at with a pixel position in the grid

  ☐ Pixel position on each $axis$: $p_{axis} = \left\lfloor f_{c,axis} \times n_p(axis) \right\rfloor$
  ☐ Pixel ID number in the grid: $P_{id}(at) = p_x + n_p(x) \times p_y + \left[n_p(x) \times n_p(y) \right] \times p_z$

☐ Associate each pixel with it surrouding neighbors in the grid, if needed apply PBC transformation(s)
☐ Search for bond candidates between the atoms of a pixel as well as the atoms of its surrouding pixel neighbors
☐ Evaluate distances using the atom's corrected fractional coordinates followed by a matrix transformation to Cartesian coordinates

Appendix A   Commented C code

A.1   C code: data structures and global variables

```c
// global data structures and variables used in the next code sections

#define TRUE 1
#define FALSE 0

// atom in pixel data structure
typedef struct pixel_atom pixel_atom;
struct pixel_atom
{
  int atom_id;              // the atom ID
  float coord[3];           // the atom coordinates on x, y and z
};

// pixel data structure
typedef struct pixel pixel;
struct pixel
{
  int pid;                  // the pixel number
  int p_xyz[3];             // the pixel coordinates in the grid
  bool tested;              // was the pixel checked already
  int patoms;               // number of atom(s) in pixel
  pixel_atom * pix_atoms;   // list of atom(s) in the pixel, to be allocated
  int neighbors;            // number of neighbors for pixel
  int pixel_neighbors[27];  // the list of neighbor pixels, maximum 27
};

// pixel grid data structure
typedef struct pixel_grid pixel_grid;
struct pixel_grid
{
  int pixels;               // total number of pixels in the grid
  int n_pix[3];             // number of pixel(s) on each axis
  int n_xy;                 // number of pixels in the plan xy
  pixel * pixel_list;       // pointer to the pixels, to be allocated
};

// bond distance data structure
typedef struct distance distance;
struct distance
{
  float length;             // the distance in \AA\ squared
  float Rij[3];             // vector components of x, y and z
};

/*
  the following are considered to be provided by the user
*/

// model description
int atoms;                  // the total number of atom(s)
float ** c_coord;           // list of Cartesian coordinates: c_coord[atoms][3]
float cutoff;               // the cutoff to define atomic bond(s)
float cutoff_squared;       // squared value for the cutoff

// model box description
float l_params[3];          // lattice a, b and c
float cart_to_frac[3][3];   // Cartesian to fractional coordinates matrix
float frac_to_cart[3][3];   // fractional to Cartesian coordinates matrix
```

A.2   C code: set periodic boundary condition pixel shift

```c
void set_pbc_shift (pixel_grid * grid, int pixel_coord[3], int pbc_shift[3][3][3])
{
  int x_pos, y_pos, z_pos; // loop iterators
  for ( x_pos = 0 ; x_pos < 3 ; x_pos ++ )
  {
    for ( y_pos = 0 ; y_pos < 3 ; y_pos ++ )
    {
      for ( z_pos = 0 ; z_pos < 3 ; z_pos ++ )
      {
        pbc_shift[x_pos][y_pos][z_pos] = 0;            // initialization without any shift
      }
    }
  }
  if ( pixel_coord[0] == 0 )                           // pixel position on 'x' is min
  {
    for ( y_pos = 0 ; y_pos < 3 ; y_pos ++ )
    {
      for ( z_pos = 0 ; z_pos < 3 ; z_pos ++ )
      {
        pbc_shift[0][y_pos][z_pos] = grid->n_pix[0];
      }
    }
  }
  else if ( pixel_coord[0] == grid->n_pix[0] - 1 )     // pixel position on 'x' is max
  {
    for ( y_pos = 0 ; y_pos < 3 ; y_pos ++ )
    {
      for ( z_pos = 0 ; z_pos < 3 ; z_pos ++ )
      {
        pbc_shift[2][y_pos][z_pos] = - grid->n_pix[0];
      }
    }
  }
  if ( pixel_coord[1] == 0 )                           // pixel position on 'y' is min
  {
    for ( x_pos = 0 ; x_pos < 3 ; x_pos ++ )
    {
      for ( z_pos = 0 ; z_pos < 3 ; z_pos ++ )
      {
        pbc_shift[x_pos][0][z_pos] += grid->n_xy;
      }
    }
  }
  else if ( pixel_coord[1] == grid->n_pix[1] - 1 )     // pixel position on 'y' is max
  {
    for ( x_pos = 0 ; x_pos < 3 ; x_pos ++ )
    {
      for ( z_pos = 0 ; z_pos < 3 ; z_pos ++ )
      {
        pbc_shift[x_pos][2][z_pos] -= grid->n_xy;
      }
    }
  }
  if ( pixel_coord[2] == 0 )                           // pixel position on 'z' is min
  {
    for ( x_pos = 0 ; x_pos < 3 ; x_pos ++ )
    {
      for ( y_pos = 0 ; y_pos < 3 ; y_pos ++ )
      {
        pbc_shift[x_pos][y_pos][0] += grid->pixels;
      }
    }
  }
  else if ( pixel_coord[2] == grid->n_pix[2] - 1 )     // pixel position on 'z' is max
  {
    for ( x_pos = 0 ; x_pos < 3 ; x_pos ++ )
    {
      for ( y_pos = 0 ; y_pos < 3 ; y_pos ++ )
      {
        pbc_shift[x_pos][y_pos][2] -= grid->pixels;
      }
    }
  }
}
```

A.3   C code: add atom to pixel

```
1  void add_atom_to_pixel (pixel * the_pixel, int pixel_coord[3], int atom_id, float atom_coord[3])
2  {
3    int axis;
4    if (! the_pixel->patoms)
5    {
6      // if the pixel do not contains any atom yet, then save its coordinates in the grid
7      for ( axis = 0 ; axis < 3 ; axis ++ )
8      {
9        the_pixel->p_xyz[axis] = pixel_coord[axis];
10     }
11     // allocate the memory to store the first pixel_atom information
12     the_pixel->pix_atoms = malloc(sizeof*the_pixel->pix_atoms);
13   }
14   else
15   {
16     // otherwise reallocate memory to store the new pixel_atom information
17     the_pixel->pix_atoms = realloc(the_pixel->pix_atoms, (the_pixel->patoms+1)*sizeof*the_pixel->pix_atoms);
18   }
19   the_pixel->pix_atoms[the_pixel->patoms].atom_id = atom_id;
20   for ( axis = 0 ; axis < 3 ; axis ++ )
21   {
22     the_pixel->pix_atoms[the_pixel->patoms].coord[axis] = atom_coord[axis];
23   }
24   // increment the number of atom(s) in the pixel
25   the_pixel->patoms ++;
26 }
```

A.4 C code: preparation of the pixel grid

```c
pixel_grid * prepare_pixel_grid (bool use_pbc)
{
  pixel_grid * grid;          // pointer to the pixel grid to create
  int axis;                   // loop iterator axis id (0=x, 1=y, 2=z)
  int aid;                    // loop iterator atom number (0, atoms--1)
  int pixel_num;              // pixel number in the grid
  int pixel_pos[3];           // pixel coordinates in the grid
  float cmin[3], cmax[3];     // float coordinates min, max values
  float f_coord[3];           // float fractional coordinates

  grid = malloc(sizeof*grid);
  if ( ! use_pbc )                            // without periodic boundary conditions
  {
    for ( axis = 0 ; axis < 3 ; axis ++ ) cmin[axis] = cmax[axis] = c_coord[0][axis];
    for ( aid = 1 ; aid < atoms ; aid ++ )     // For all atoms
    {
      for ( axis = 0 ; axis < 3 ; axis ++ )    // For x, y and z
      {
        cmin[axis] = min(cmin[axis], c_coord[aid][axis]);
        cmax[axis] = max(cmax[axis], c_coord[aid][axis]);
      }
    }
    for ( axis = 0 ; axis < 3 ; axis ++ )      // For x, y and z
    {
      grid->n_pix[axis] = (int)((cmax[axis] - cmin[axis]) / cutoff) + 1; // number of pixel(s) on axis 'axis'
    }
  }
  else  // using periodic boundary conditions
  {
    for ( axis = 0 ; axis < 3 ; axis ++ )      // For x, y and z
    {
      grid->n_pix[axis] = (int)(l_params[axis] / cutoff) + 1; // number of pixel(s) on axis 'axis'
    }
  }
  for ( axis = 0 ; axis < 3 ; axis ++ )        // For x, y and z
  {
    // correction if the number of pixel(s) on 'axis' is too small
    grid->n_pix[axis] = (grid->n_pix[axis] < 4) ? 1 : grid->n_pix[axis];
  }
  grid->n_xy = grid->n_pix[0] * grid->n_pix[1]; // number of pixels on the plan 'xy'
  grid->pixels = grid->n_xy * grid->n_pix[2];   // total number of pixels in the grid
  grid->pixel_list = malloc (grid->pixels*sizeof*grid->pixel_list);
  for ( pixel_num = 0 ; pixel_num < grid->pixels ; pixel_num ++ )
  {
    grid->pixel_list[pixel_num].pid = pixel_num;
    grid->pixel_list[pixel_num].patoms = 0;
    grid->pixel_list[pixel_num].tested = FALSE;
  }
  if ( ! use_pbc )                            // without periodic boundary conditions
  {
    for ( aid = 0 ; aid < atoms ; aid ++ )     // for all atoms
    {
      for ( axis = 0 ; axis < 3 ; axis ++ )    // for x, y and z
      {
        pixel_pos[axis] = (int)((c_coord[aid][axis] - cmin[axis])/cutoff);
      }
      pixel_num = pixel_pos[0] + pixel_pos[1] * grid->n_pix[0] + pixel_pos[2] * grid->n_xy;
      add_atom_to_pixel (grid, pixel_num, pixel_pos, aid, c_coord[aid]);
    }
  }
  else                                        // using periodic boundary conditions
  {
    for ( aid = 0 ; aid < atoms ; aid ++ )     // for all atoms
    {
      // with 'matrix_multiplication' a user defined function to perform the operation
      f_coord = matrix_multiplication (cart_to_frac, c_coord[aid]);
      for ( axis = 0 ; axis < 3 ; axis ++ )    // for x, y and z
      {
        f_coord[axis] = f_coord[axis] - floorf(f_coord[axis]);
        pixel_pos[axis] = (int)((f_coord[axis] * grid->n_pix[axis]);
      }
      pixel_num = pixel_pos[0] + pixel_pos[1] * grid->n_pix[0] + pixel_pos[2] * grid->n_xy;
      add_atom_to_pixel (& grid->pixel_list[pixel_num], aid, f_coord);
    }
  }
  return grid;
}
```

A.5 C code: finding pixel neighbors

```
1   // finding neighbor pixels for pixel in the grid
2   // - bool use_pbc          : flag to set if PBC are used or not
3   // - pixel_grid * the_grid : pointer to the pixel grid
4   // - pixel * the_pix       : pointer to the pixel with neighbors to be found
5   void find_pixel_neighbors (bool use_pbc, pixel_grid * the_grid, pixel * the_pix)
6   {
7     int axis;                      // loop iterator axis id (1=x , 2=y , 3= z)
8     int xpos, ypos, zpos;          // neighbor position on x, y and z
9     int l_start[3] = { 0, 0, 0};   // loop iterators starting value
10    int l_end[3] = { 3, 3, 3};     // loop iterators ending value
11    int pmod[3] = {-1, 0, 1};      // position modifiers
12    int nnp;                       // number of neighbors for pixel
13    int nid;                       // neighbor id for pixel
14    int pbc_shift[3][3][3];        // shift for pixel neighbor number due to PBC
15    bool boundary = FALSE;         // is pixel on the boundary of the grid
16    bool keep_neighbor = TRUE;     // keep or not neighbor during analysis
17
18    if ( use_pbc )
19    {
20      set_pbc_shift (the_grid, the_pix->p_xyz, pbc_shift);
21    }
22    else
23    {
24      for ( axis = 0 ; axis < 3 ; axis ++ )
25      {
26        if ( the_pix->p_xyz[axis] == 0 || the_pix->p_xyz[axis] == the_grid->n_pix[axis] - 1 ) boundary = TRUE;
27      }
28    }
29    for ( axis = 0 ; axis < 3 ; axis ++ )
30    {
31      if ( the_grid->n_pix[axis] == 1 )
32      {
33        l_start[axis] = 1;
34        l_end[axis] = 2;
35      }
36    }
37    nnp = 0;
38    for ( xpos = l_start[0] ; xpos < l_end[0] ; xpos ++ )
39    {
40      for ( ypos = l_start[1] ; ypos < l_end[1] ; ypos ++ )
41      {
42        for ( zpos = l_start[2] ; zpos < l_end[2] ; zpos ++ )
43        {
44          keep_neighbor = TRUE;
45          if ( ! use_pbc && boundary )
46          {
47            if (( the_pix->p_xyz[0] == 0 && xpos == 0 ) || ( the_pix->p_xyz[0] == the_grid->n_pix[0] && xpos == 2 ))
48            {
49              keep_neighbor = FALSE;
50            }
51            else if (( the_pix->p_xyz[1] == 0 && ypos == 0 ) || ( the_pix->p_xyz[1] == the_grid->n_pix[1] && ypos == 2 ))
52            {
53              keep_neighbor = FALSE;
54            }
55            else if (( the_pix->p_xyz[2] == 0 && zpos == 0 ) || ( the_pix->p_xyz[2] == the_grid->n_pix[2] && zpos == 2 ))
56            {
57              keep_neighbor = FALSE;
58            }
59          }
60          if ( keep_neighbor )
61          {
62            nid = the_pix->pid + pmod[xpos] + pmod[ypos] * the_grid->n_pix[0] + pmod[zpos] * the_grid->n_xy;
63            if ( use_pbc ) nid += pbc_shift[xpos][ypos][zpos];
64            the_pix->pixel_neighbors[nnp] = nid;
65            nnp ++ ;
66          }
67        }
68      }
69    }
70    the_pix->neighbors = nnp;
71  }
```

A.6 C code: inter-atomic distance calculation

```
1  // evaluating the interatomic distance between 2 pixel atoms
2  // - bool use_pbc      : flag to set if PBC are used or not
3  // - pixel_atom * at_i : pointer to first pixel atom
4  // - pixel_atom * at_j : pointer to second pixel atom
5  distance evaluate_distance (bool use_pbc, pixel_atom * at_i, pixel_atom * at_j)
6  {
7    int axis;            // axis loop iterator
8    float u, v;          // float parameters
9    distance dist;       // distance data to store calculation results
10   for ( axis = 0 ; axis < 3 ; axis ++ )
11   {
12     dist.Rij[axis] = at_i->coord[axis] - at_j->coord[axis];
13   }
14   if ( use_pbc )
15   {
16     // then the pixel_atom's coordinates are in corrected fractional format
17     for ( axis = 0 ; axis < 3 ; axis ++ )
18     {
19       dist.Rij[axis] = dist.Rij[axis] - roundf(dist.Rij[axis]);
20     }
21     // transform back to Cartesian coordinates
22     // with 'matrix_multiplication' a user defined function to perform the operation
23     dist.Rij = matrix_multiplication (frac_to_cart, dist.Rij);
24   }
25   dist.length = 0.0;
26   for ( axis = 0 ; axis < 3 ; axis ++ )
27   {
28     dist.length += dist.Rij[axis] * dist.Rij[axis];
29   }
30   // returning the 'distance' data structure that contains:
31   // - the squared value for Dij: no time consuming square root calculation !
32   // - the components of the distance vector on x, y and z
33   return dist;
34 }
```

A.7 C code: pixel search for first neighbor atoms

```c
// searching for first neighbor atoms using the grid pixelation/partitioning method
// - bool use_pbc : flag to set if PBC are used or not
void pixel_search_for_neighbors (bool use_pbc)
{
  pixel_grid * all_pixels;    // pointer to the pixel grid for to analyze
  int pix, pjx;               // integer pixel ID numbers
  int aid, bid;               // integer loop atom numbers
  int pid;
  int l_start, l_end;         // integer loop modifier
  pixel * pix_i, * pix_j;     // pointers on pixel data structure
  pixel_atom * at_i, * at_j;  // pointers on pixel_atom data structure
  distance Dij;               // distance data structure

  all_pixels = prepare_pixel_grid (use_pbc);
  // note that the pixel grid 'all_pixels' must be prepared before the following
  // for all pixels in the grid
  for ( pix = 0 ; pix < all_pixels->pixels ; pix ++ )
  {
    // setting 'pix_i' as pointer to pixel number 'pix'
    pix_i = & all_pixels->pixel_list[pix];
    // if pixel 'pix_i' contains atom(s)
    if ( pix_i->patoms )
    {
      // search for neighbbor pixels of 'pix_i'
      find_pixel_neighbors ( use_pbc, all_pixels, pix_i );
      // testing all 'pix_i' neighbor pixels
      for ( pid = 0 ; pid < pix_i->neighbors ; pid ++ )
      {
        pjx = pix_i->pixel_neighbors[pid];
        // setting 'pix_j' as pointer to pixel number 'pjx'
        pix_j = & all_pixels->pixel_list[pjx];
        // checking pixel 'pix_j' if it:
        // - contains atom(s)
        // - was not tested, otherwise the analysis would have been performed already
        if ( pix_j->patoms && ! pix_j->tested )
        {
          // if 'pix_i' and 'pix_j' are the same, only test pair of different atoms
          l_end = (pjx != pix) ? 0 : 1;
          // for all atom(s) in 'pix'
          for ( aid = 0 ; aid < pix_i->patoms - l_end ; aid ++ )
          {
            // set pointer to the first atom to test
            at_i = & pix_i->pix_atoms[aid];
            lstart = (pjx != pix) ? 0 : aid + 1;
            // for all atom(s) in 'pix_j'
            for ( bid = l_start ; bid < pix_j->patoms ; bid ++ )
            {
              // set pointer to the second atom to test
              at_j = & pix_j->pix_atoms[bid];
              // evaluate interatomic distance
              Dij = evaluate_distance (use_pbc, at_i, at_j);
              if ( Dij.length < cutoff_squared )
              {
                // this is a first neighbor bond !
              }
            }
          }
        }
      }
      // store that pixel 'pix' was tested
      pix_i->tested = TRUE;
    }
  }
}
```

Appendix B   Commented FORTRAN90 code

B.1   FORTRAN90 code: data structures and global variables

```fortran
! global data structures and variables used in the next code sections
MODULE parameters

  ! atom in pixel data structure
  TYPE pixel_atom
    INTEGER                                     :: atom_id          ! the atom ID
    REAL, DIMENSION(3)                          :: coord            ! the atom coordinates on x, y and z
  END TYPE pixel_atom

  ! pixel data structure
  TYPE pixel
    INTEGER                                     :: pid              ! the pixel number
    INTEGER, DIMENSION(3)                       :: p_xyz            ! the pixel coordinates in the grid
    LOGICAL                                     :: tested           ! was the pixel checked already
    INTEGER                                     :: patoms           ! number of atom(s) in pixel
    TYPE(pixel_atom), DIMENSION(:), ALLOCATABLE :: pix_atoms        ! list of atom(s) in pixel, to be allocated
    INTEGER                                     :: neighbors        ! number of neighbors for pixel
    INTEGER, DIMENSION(27)                      :: pixel_neighbors  ! the list of neighbor pixels, maximum 27
  END TYPE pixel

  ! pixel grid data structure
  TYPE pixel_grid
    INTEGER                                         :: pixels        ! total number of pixels in the grid
    INTEGER, DIMENSION(3)                           :: n_pix         ! number of pixel(s) on each axis
    INTEGER                                         :: n_xy          ! number of pixels in the plan xy
    TYPE (pixel), DIMENSION(:), ALLOCATABLE, TARGET :: pixel_list    ! pointer to the pixels, to be allocated
  END TYPE pixel_grid

  ! distance data structure
  TYPE distance
    REAL                                        :: length           ! the distance in \AA\ squared
    REAL, DIMENSION(3)                          :: Rij              ! vector components of x, y and z
  END TYPE distance

  !
  ! the following are considered to be provided by the user
  !

  ! model description
  INTEGER                                       :: atoms            ! the total number of atom(s)
  REAL, DIMENSION(atoms,3)                      :: c_coord          ! list of Cartesian coordinates
  REAL                                          :: cutoff           ! the cutoff to define first neighbors
  REAL                                          :: cutoff_squared   ! squared value of the cutoff to define first neighbors

  ! model box description
  REAL, DIMENSION(3)                            :: l_params         ! lattice a, b and c
  REAL, DIMENSION(3,3)                          :: cart_to_frac     ! Cartesian to fractional coordinates matrix
  REAL, DIMENSION(3,3)                          :: frac_to_cart     ! fractional to Cartesian coordinates matrix

END MODULE parameters
```

B.2 FORTRAN90 code: set pixel periodic boundary condition shift

```fortran
1  SUBROUTINE set_pbc_shift (grid, pixel_coord, pbc_shift)
2
3    USE parameters
4    IMPLICIT NONE
5
6    TYPE (pixel_grid), INTENT(IN)          :: grid         ! the pixel grid
7    INTEGER, DIMENSION(3), INTENT(IN)      :: pixel_coord  ! the pixel coordinates in the grid
8    INTEGER, DIMENSION(3,3,3), INTENT(INOUT) :: pbc_shift  ! the shift, correction, to be calculated
9
10   pbc_shift(:,:,:) = 0                                   ! initialization without any shift
11
12   if ( pixel_coord(1) .eq. 1 ) then                     ! pixel position on 'x' is min
13     pbc_shift(1,:,:) = grid%n_pix(1)
14   else if ( pixel_coord(1) .eq. grid%n_pix(1) ) then    ! pixel position on 'x' is max
15     pbc_shift(3,:,:) = - grid%n_pix(1)
16   endif
17
18   if ( pixel_coord(2) .eq. 1 ) then                     ! pixel position on 'y' is min
19     pbc_shift(:,1,:) = pbc_shift(:,1,:) + grid%n_xy
20   else if ( pixel_coord(2) .eq. grid%n_pix(2) ) then    ! pixel position on 'y' is max
21     pbc_shift(:,3,:) = pbc_shift(:,3,:) - grid%n_xy
22   endif
23
24   if ( pixel_coord(3) .eq. 1 ) then                     ! pixel position on 'z' is min
25     pbc_shift(:,:,1) = pbc_shift(:,:,1) + grid%pixels
26   else if ( pixel_coord(3) .eq. grid%n_pix(3) ) then    ! pixel position on 'z' is max
27     pbc_shift(:,:,3) = pbc_shift(:,:,3) - grid%pixels
28   endif
29
30 END SUBROUTINE set_pbc_shift
```

B.3 FORTRAN90 code: add atom to pixel

```fortran
1   SUBROUTINE add_atom_to_pixel (the_pixel, pixel_coord, atom_id, atom_coord)
2
3     USE parameters
4     IMPLICIT NONE
5
6     TYPE (pixel), INTENT(INOUT) :: the_pixel       ! pointer to the pixel
7     INTEGER, DIMENSION(3), INTENT(IN) :: pixel_coord  ! the pixel coordinates in the grid
8     INTEGER, INTENT(IN)               :: atom_id      ! the atom ID number
9     INTEGER                           :: axis         ! loop iterator axis id (1=x, 2=y, 3=z)
10    REAL, DIMENSION(3), INTENT(IN)    :: atom_coord   ! the atomic coordinates
11
12    if (the_pixel%patoms .eq. 0) then
13      ! if the pixel do not contains any atom yet, then save its coordinates in the grid
14      do axis = 1 , 3
15        the_pixel%p_xyz(axis) = pixel_coord(axis)
16      enddo
17       ! allocate the memory to store the first pixel_atom information
18      allocate(the_pixel%pix_atoms(1))
19    else
20      ! otherwise reallocate memory to store the new pixel_atom informatio
21      the_pixel%pix_atoms = realloc(the_pixel%pix_atoms, the_pixel%patoms+1)
22    endif
23    ! increment the number of atom(s) in the pixel
24    the_pixel%patoms = the_pixel%patoms + 1
25    the_pixel%pix_atoms(the_pixel%patoms)%atom_id = atom_id
26    do axis = 1 , 3
27      the_pixel%pix_atoms(the_pixel%patoms)%coord(axis) = atom_coord(axis)
28    enddo
29
30  END SUBROUTINE
```

B.4 FORTRAN90 code: preparation of the pixel grid

```fortran
1  ! preparation of the pixel grid
2  SUBROUTINE prepare_pixel_grid (use_pbc, grid)
3
4    USE parameters
5    IMPLICIT NONE
6
7    LOGICAL, INTENT(IN)            :: use_pbc        ! flag to set if PBC are used or not
8    TYPE (pixel_grid), INTENT(INOUT) :: grid         ! the pixel grid to prepare
9    INTEGER                        :: axis           ! loop iterator axis id (1=x, 2=y, 3=z)
10   INTEGER                        :: aid            ! loop iterator atom number (1, atoms)
11   INTEGER                        :: pixel_num      ! pixel number in the grid
12   INTEGER, DIMENSION(3)          :: pixel_pos      ! pixel coordinates in the grid
13   REAL, DIMENSION(3)             :: cmin, cmax     ! real precision coordinates min, max
14   REAL, DIMENSION(3)             :: f_coord        ! real precision fractional coordinates
15
16   if ( .not. use_pbc ) then                        ! without periodic boundary conditions
17     do axis = 1 , 3
18       cmin(axis) = c_coord(1,axis)
19       cmax(axis) = c_coord(1,axis)
20     enddo
21     do aid = 2 , atoms                             ! for all atoms
22       do axis = 1 , 3                              ! for x, y and z
23         cmin(axis) = min(cmin(axis), c_coord(aid,axis))
24         cmax(axis) = max(cmax(axis), c_coord(aid,axis))
25       enddo
26     enddo
27     do axis = 1 , 3                                ! for x, y and z
28       ! number of pixel(s) on axis 'axis'
29       grid%n_pix(axis) = INT((cmax(axis) - cmin(axis)) / cutoff) + 1
30     enddo
31   else                                             ! using periodic boundary conditions
32     do axis = 1 , 3                                ! for x, y and z
33       ! number of pixel(s) on axis 'axis'
34       grid%n_pix(axis) = INT(l_params(axis) / cutoff) + 1
35     enddo
36   endif
37   do axis = 1 , 3                                  ! for x, y and z
38     ! correction if the number of pixel(s) on 'axis' is too small
39     if ( grid%n_pix(axis) .lt. 4 ) then
40       grid%n_pix(axis) = 1
41     endif
42   enddo
43
44   grid%n_xy = grid%n_pix(1) * grid%n_pix(2)        ! number of pixels on the plan 'xy'
45   grid%pixels = grid%n_xy * grid%n_pix(3)          ! total number of pixels in the grid
46
47   allocate (grid%pixel_list(grid%pixels))
48   do pixel_num = 1 , grid%pixels
49     grid%pixel_list(pixel_num)%pid = pixel_num
50     grid%pixel_list(pixel_num)%patoms = 0
51     grid%pixel_list(pixel_num)%tested = .false.
52   enddo
53   if ( .not. use_pbc ) then                        ! without periodic boundary conditions
54     do aid = 1 , atoms                             ! for all atoms
55       do axis = 1 , 3                              ! for x, y and z
56         pixel_pos(axis) = INT( (c_coord(aid,axis) - cmin(axis) )/ cutoff)
57       enddo
58       pixel_num = pixel_pos(1) + pixel_pos(2) * grid%n_pix(1) + pixel_pos(3) * grid%n_xy + 1
59       call add_atom_to_pixel (grid%pixel_list(pixel_num), pixel_pos, aid, c_coord(aid,:))
60     enddo
61   else                                             ! using periodic boundary conditions
62     do aid = 1 , atoms                             ! for all atoms
63       f_coord = MATMUL ( c_coord(aid), cart_to_frac )
64       do axis = 1 , 3                              ! for x, y and z
65         f_coord(axis) = f_coord(axis) - floor(f_coord(axis))
66         pixel_pos(axis) = INT(f_coord(axis) * grid%n_pix(axis))
67       enddo
68       pixel_num = pixel_pos(1) + pixel_pos(2) * grid%n_pix(1) + pixel_pos(3) * grid%n_xy + 1
69       call add_atom_to_pixel (grid%pixel_list(pixel_num), pixel_pos, aid, f_coord)
70     enddo
71   endif
72
73 END SUBROUTINE prepare_pixel_grid
```

B.5   FORTRAN90 code: finding pixel neighbors

```fortran
1  SUBROUTINE find_pixel_neighbors (use_pbc, the_grid, the_pix)
2
3    USE parameters
4    IMPLICIT NONE
5
6    LOGICAL, INTENT(IN)               :: use_pbc              ! flag to set if PBC are used or not
7    TYPE (pixel_grid), INTENT(INOUT)    :: the_grid              ! pointer to the pixel grid
8    TYPE (pixel), POINTER, INTENT(INOUT) :: the_pix             ! pointer to the pixel
9    INTEGER                           :: axis                 ! loop iterator axis id (1=x, 2=y, 3=z)
10   INTEGER                           :: xpos, ypos, zpos     ! neighbor position on x, y and z
11   INTEGER, DIMENSION(3)             :: l_start = (\1, 1, 1\)  ! loop iterators starting value
12   INTEGER, DIMENSION(3)             :: l_end = (\3, 3, 3\)    ! loop iterators ending value
13   INTEGER, DIMENSION(3)             :: pmod = (\-1, 0, 1\)    ! position modifiers
14   INTEGER                           :: nnp                  ! number of neighbors for pixel
15   INTEGER                           :: nid                  ! neighbor id for pixel
16   INTEGER, DIMENSION(3,3,3)         :: pbc_shift            ! shift, correction, due to PBC
17   LOGICAL                           :: boundary=.false.     ! is pixel on the boundary of the grid
18   LOGICAL                           :: keep_neighbor = .true. ! keep or not neighbor during analysis
19
20   if ( use_pbc ) then
21     call set_pbc_shift (the_grid, the_pix%p_xyz, pbc_shift)
22   else
23     do axis = 1 , 3
24       if ( the_pix%p_xyz(axis) .eq. 1 .or. the_pix%p_xyz(axis) .eq. the_grid%n_pix(axis) ) then
25         boundary = .true.
26       endif
27     enddo
28   endif
29   do axis = 1 , 3
30     if ( the_grid%n_pix(axis) .eq. 1 ) then
31       l_start(axis) = 2
32       l_end(axis) = 2
33     endif
34   enddo
35   nnp = 0
36   do xpos = l_start(1) , l_end(1)
37     do ypos = l_start(2) , l_end(2)
38       do zpos = l_start(3) , l_end(3)
39         keep_neighbor = .true.
40         if ( .not. use_pbc .and. boundary ) then
41           if ( the_pix%p_xyz(1) .eq. 1 .and. xpos .eq. 1 ) then
42             keep_neighbor = .false.
43           else if ( the_pix%p_xyz(1) .eq. the_grid%n_pix(1) .and. xpos .eq. 3 ) then
44             keep_neighbor = .false.
45           else if ( the_pix%p_xyz(2) .eq. 1 .and. ypos .eq. 1 ) then
46             keep_neighbor = .false.
47           else if ( the_pix%p_xyz(2) .eq. the_grid%n_pix(2) .and. ypos .eq. 3 ) then
48             keep_neighbor = .false.
49           else if ( the_pix%p_xyz(3) .eq. 1 .and. zpos .eq. 1 ) then
50             keep_neighbor = .false.
51           else if ( the_pix%p_xyz(3) .eq. the_grid%n_pix(3) .and. zpos .eq. 3 ) then
52             keep_neighbor = .false.
53           endif
54         endif
55         if ( keep_neighbor ) then
56           ! evaluating neighbor pixel number in the grid
57           nid = the_pix%pid + pmod(xpos) + pmod(ypos) * the_grid%n_pix(1) + pmod(zpos) * the_grid%n_xy
58           if ( use_pbc ) then
59             ! correcting the value if PBC are used
60             nid = nid + pbc_shift(xpos, ypos, zpos)
61           endif
62           the_pix%pixel_neighbors(nnp) = nid
63           nnp = nnp + 1
64         endif
65       enddo
66     enddo
67   enddo
68   the_pix%neighbors = nnp
69
70 END SUBROUTINE find_pixel_neighbors
```

B.6 FORTRAN90 code: inter-atomic distance calculation

```fortran
! evaluating the interatomic distance between 2 pixel atoms
SUBROUTINE evaluate_distance (use_pbc, at_i, at_j, dist)

  USE parameters
  IMPLICIT NONE

  LOGICAL, INTENT(IN)                  :: use_pbc   ! flag to set if PBC are used or not
  TYPE (pixel_atom), POINTER, INTENT(IN) :: at_i      ! first pixel atom
  TYPE (pixel_atom), POINTER, INTENT(IN) :: at_j      ! second pixel atom
  TYPE (distance), INTENT(INOUT)       :: dist      ! calculation results
  INTEGER                              :: axis      ! loop iterator axis id (1=x, 2=y, 3=z)

  do axis = 1 , 3
    dist%Rij(axis) = at_i%coord(axis) - at_j%coord(axis)
  enddo
  if ( use_pbc ) then
    ! then the pixel_atom's coordinates are in corrected fractional format
    do axis = 1 , 3
      dist%Rij(axis) = dist%Rij(axis) - AnINT(dist%Rij(axis))
    enddo
    ! transform back to Cartesian coordinates
    dist%Rij = MATMUL( dist%Rij, frac_to_cart )
  endif

  dist%length = 0.0
  do axis = 1 , 3
    dist%length = dist%length + dist%Rij(axis) * dist%Rij(axis)
  enddo
  ! returning the 'distance' data structure that contains:
  ! - the squared value for Dij: no time consuming square root calculation !
  ! - the components of the distance vector on x, y and z
END SUBROUTINE evaluate_distance
```

B.7 FORTRAN90 code: pixel search for first neighbor atoms

```
1  SUBROUTINE pixel_search_for_neighbors (use_pbc)
2
3    USE parameters
4    IMPLICIT NONE
5
6    LOGICAL, INTENT(IN)   :: use_pbc                   ! flag to set if PBC are used or not
7    TYPE (pixel_grid)     :: all_pixels                ! the pixel grid to analyze
8    INTEGER               :: pix, pjx                  ! integer pixel ID numbers
9    INTEGER               :: aid, bid                  ! integer loop atom numbers
10   INTEGER               :: l_start, l_end            ! integer loop modifier
11   INTEGER               :: pid
12   TYPE (pixel_atom), POINTER :: pix_i, pix_j         ! pointers on pixel data structure
13   TYPE (atom), POINTER  :: at_i, at_j                ! pointers on pixel_atom data structure
14   TYPE (distance)       :: Dij                       ! distance data structure
15
16   call prepare_pixel_grid (use_pbc, all_pixels)
17   ! note that the pixel grid 'all_pixels' must be prepared before the following
18   ! for all pixels in the grid
19   do pix = 1 , all_pixels%pixels
20     ! setting 'pix_i' as pointer to pixel number 'pix'
21     pix_i => all_pixels%pixel_list(pix)
22     ! if pixel 'pix_i' contains atom(s)
23     if ( pix_i%patoms .gt. 0 ) then
24
25       ! search for pixel neighbbors of 'pix_'
26       call find_pixel_neighbors (use_pbc, all_pixels, pix_i)
27
28       ! testing all 'pix_i' neighbor pixels
29       do pid = 1 , pix_i%neighbors
30         pjx = pix_i%pixel_neighbors(pid)
31         ! setting 'pix_j' as pointer to pixel number 'pjx'
32         pix_j => all_pixels%pixel_list(pjx)
33         ! checking pixel 'pix_j' if it:
34         ! - contains atom(s)
35         ! - was not tested, otherwise the analysis would have been performed already
36         if ( pix_j%patoms .gt. 0 .and. .not. pix_j%tested ) then
37           ! If 'pix_i' and 'pix_j' are the same, only test pair of different atoms
38           if ( pjx .eq. pix ) then
39             l_end = 1
40           else
41             l_end = 0
42           endif
43           ! for all atom(s) in 'pix_i'
44           do aid = 1 , pix_i%patoms - l_end
45             ! set pointers to the first atom to test
46             at_i => pix_i%pix_atoms(aid)
47             if ( pjx .eq. pix ) then
48               l_start = aid + 1
49             else
50               l_start = 1
51             endif
52             ! for all atom(s) in 'pix_j'
53             do bid = l_start , pix_j%patoms
54               ! set pointers to the second atom to test
55               at_j => pix_j%pix_atoms(bid)
56               ! evaluate interatomic distance
57               call evaluate_distance (use_pbc, at_i, at_j, Dij)
58               if ( Dij%length .lt. cutoff_squared ) then
59                 ! this is a first neighbor bond !
60               endif
61             enddo
62           enddo
63         endif
64       enddo
65       ! store that pixel 'pix_i' was tested
66       pix_i%tested = .true.
67     endif
68   enddo
69
70 END SUBROUTINE pixel_search_for_neighbors
```

Appendix C   Commented Python code

C.1   Python code: data structures and global variables

```python
1  # Global data structures and variables used in the next code sections
2  import numpy as np
3
4  # atom in pixel data structure
5  class PixelAtom:
6    def __init__(self, atom_id=0, coord=None):
7      self.atom_id = atom_id                                 # the atom ID
8      self.coord = np.zeros(3) if coord is None else np.array(coord)    # the atom coordinates on x, y and z
9
10 # pixel data structure
11 class Pixel:
12   def __init__(self, pid=0, p_xyz=None, tested=False, patoms=0, pix_atoms=None, neighbors=0):
13     self.pid = pid                                         # the pixel number
14     self.p_xyz = np.zeros(3) if p_xyz is None else np.array(p_xyz)    # the pixel coordinates in the grid
15     self.tested = tested                                   # was the pixel checked already
16     self.patoms = patoms                                   # number of atom(s) in pixel
17     self.pix_atoms = [] if pix_atoms is None else pix_atoms # list of atom(s) in the pixel
18     self.neighbors = neighbors                             # number of neighbors for pixel
19     self.pixel_neighbors = np.zeros(27, dtype=int)         # the list of neighbor pixels, maximum 27
20
21 # pixel grid data structure
22 class PixelGrid:
23   def __init__(self, pixels=0, n_pix=None, n_xy=0, pixel_list=None):
24     self.pixels = pixels                                   # total number of pixels in the grid
25     self.n_pix = np.zeros(3, dtype=int) if n_pix is None else np.array(n_pix) # pixel(s) on each axis
26     self.n_xy = n_xy                                       # number of pixels in the plan xy
27     self.pixel_list = [] if pixel_list is None else pixel_list # pointer to the pixels, to be allocated
28
29 # bond distance data structure
30 class Distance:
31   def __init__(self, length=0.0, Rij=None):
32     self.length = length                                   # the distance in Å squared
33     self.Rij = np.zeros(3) if Rij is None else np.array(Rij) # vector components of x, y and z
34
35 #
36 # the following are considered to be provided by the user
37 #
38
39 # model description
40 atoms = 0                                                  # the total number of atom(s)
41 c_coord = None                                             # list of Cartesian coordinates: c_coord[atoms][3]
42 cutoff = 0.0                                               # the cutoff to define atomic bond(s)
43 cutoff_squared = 0.0                                       # squared value for the cutoff
44
45 # model box description
46 l_params = np.zeros(3)                                     # lattice a, b and c
47 cart_to_frac = np.zeros((3, 3))                            # Cartesian to fractional coordinates matrix
48 frac_to_cart = np.zeros((3, 3))                            # fractional to Cartesian coordinates matrix
```

C.2 Python code: set pixel periodic boundary condition shift

```python
# adjust, if needed, shift to search for pixel neighbor(s) using PBC
# - grid pixel_grid        : the pixel grid
# - int pixel_coord[3]      : the pixel coordinates in the grid
# - int pbc_shift[3][3][3]   : the shift, correction, to be calculated
def set_pbc_shift(pixel_grid : PixelGrid, pixel_coord : np.ndarray, pbc_shift : np.ndarray):
  # Initialize pbc_shift to zero
  for x_pos in range(3):
    for y_pos in range(3):
      for z_pos in range(3):
        pbc_shift[x_pos][y_pos][z_pos] = 0        # initialization without any shift

  if pixel_coord[0] == 0:                          # pixel position on 'x' is min
    for y_pos in range(3):
      for z_pos in range(3):
        pbc_shift[0][y_pos][z_pos] = pixel_grid.n_pix[0]

  elif pixel_coord[0] == pixel_grid.n_pix[0] - 1: # pixel position on 'x' is max
    for y_pos in range(3):
      for z_pos in range(3):
        pbc_shift[2][y_pos][z_pos] = -pixel_grid.n_pix[0]

  if pixel_coord[1] == 0:                          # pixel position on 'y' is min
    for x_pos in range(3):
      for z_pos in range(3):
        pbc_shift[x_pos][0][z_pos] += pixel_grid.n_xy

  elif pixel_coord[1] == pixel_grid.n_pix[1] - 1: # pixel position on 'y' is max
    for x_pos in range(3):
      for z_pos in range(3):
        pbc_shift[x_pos][2][z_pos] -= pixel_grid.n_xy

  if pixel_coord[2] == 0:                          # pixel position on 'z' is min
    for x_pos in range(3):
      for y_pos in range(3):
        pbc_shift[x_pos][y_pos][0] += pixel_grid.pixels

  elif pixel_coord[2] == pixel_grid.n_pix[2] - 1: # pixel position on 'z' is max
    for x_pos in range(3):
      for y_pos in range(3):
        pbc_shift[x_pos][y_pos][2] -= pixel_grid.pixels
```

C.3 Python code: add atom to pixel

```python
def add_atom_to_pixel(the_pixel: Pixel, pixel_coord: np.ndarray, atom_id: int, atom_coord: np.ndarray):
  if not the_pixel.patoms:
    # if the pixel do not contains any atom yet, then save its coordinates in the grid
    for axis in range(3):
      the_pixel.p_xyz[axis] = pixel_coord[axis]

  new_atom = PixelAtom(atom_id, atom_coord)
  the_pixel.pix_atoms.append(new_atom)

  # increment the number of atom(s) in the pixel
  the_pixel.patoms += 1
```

C.4 Python code: preparation of the pixel grid

```python
1  # preparation of the pixel grid
2  # - bool use_pbc : flag to set if PBC are used or not
3  def prepare_pixel_grid(use_pbc : bool):
4    grid = PixelGrid()                        # create a new pixel grid
5    cmin = [float('inf')] * 3                  # initialize to infinity
6    cmax = [-float('inf')] * 3                 # initialize to negative infinity
7    pixel_pos = np.zeros(3, dtype=int)
8
9    if not use_pbc:                            # without periodic boundary conditions
10     for axis in range(3):
11       cmin[axis] = cmax[axis] = c_coord[0][axis]
12     for aid in range(1, atoms):              # for all atoms
13       for axis in range(3):                  # for x, y and z
14         cmin[axis] = min(cmin[axis], c_coord[aid][axis])
15         cmax[axis] = max(cmax[axis], c_coord[aid][axis])
16     for axis in range(3):                    # for x, y and z
17       # number of pixels on axis 'axis'
18       grid.n_pix[axis] = int((cmax[axis] - cmin[axis]) / cutoff) + 1
19   else:                                      # Using periodic boundary conditions
20     for axis in range(3):                    # for x, y and z
21       # number of pixels on axis 'axis'
22       grid.n_pix[axis] = int(l_params[axis] / cutoff) + 1
23
24   for axis in range(3):                      # for x, y and z
25     # correction if the number of pixels on 'axis' is too small
26     grid.n_pix[axis] = 1 if grid.n_pix[axis] < 4 else grid.n_pix[axis]
27
28   grid.n_xy = grid.n_pix[0] * grid.n_pix[1]      # number of pixels on the plan 'xy'
29   grid.pixels = grid.n_xy * grid.n_pix[2]        # total number of pixels in the grid
30
31   grid.pixel_list = [Pixel(pid=i) for i in range(grid.pixels)]
32
33   if not use_pbc:                            # without periodic boundary conditions
34     for aid in range(atoms):                 # for all atoms
35       for axis in range(3):                  # for x, y and z
36         pixel_pos[axis] = int((c_coord[aid][axis] - cmin[axis]) / cutoff)
37       pixel_num = pixel_pos[0] + pixel_pos[1] * grid.n_pix[0] + pixel_pos[2] * grid.n_xy
38       add_atom_to_pixel(grid.pixel_list[pixel_num], pixel_pos, aid, c_coord[aid])
39   else:                                      # using periodic boundary conditions
40     for aid in range(atoms):                 # for all atoms
41       # with 'matrix_multiplication' a user defined function to perform the operation
42       f_coord = matrix_multiplication(cart_to_frac, c_coord[aid])
43       for axis in range(3):                  # for x, y and z
44         f_coord[axis] = f_coord[axis] - np.floor(f_coord[axis])
45         pixel_pos[axis] = int(f_coord[axis] * grid.n_pix[axis])
46       pixel_num = pixel_pos[0] + pixel_pos[1] * grid.n_pix[0] + pixel_pos[2] * grid.n_xy
47       add_atom_to_pixel(grid.pixel_list[pixel_num], pixel_pos, aid, f_coord)
48
49   return grid
```

C.5   Python code: finding pixel neighbors

```python
# Finding neighbor pixels for pixel in the grid
# - bool use_pbc     : flag to set if PBC are used or not
# - grid * the_grid : pointer to the pixel grid
# - pixel * the_pix : pointer to the pixel with neighbors to be found
def find_pixel_neighbors(use_pbc : bool, the_grid : PixelGrid, the_pix : Pixel):
  boundary = False                           # is pixel on the boundary of the grid
  keep_neighbor = True                       # keep or not neighbor during analysis
  l_start = [0, 0, 0]                        # loop iterators starting value
  l_end = [3, 3, 3]                          # loop iterators ending value
  pmod = [-1, 0, 1]                          # position modifiers
  pbc_shift = np.zeros((3, 3, 3), dtype=int)  # shift for pixel neighbor number due to PBC

  # check if PBC are used
  if use_pbc:
    set_pbc_shift(the_grid, the_pix.p_xyz, pbc_shift)
  else:
    for axis in range(3):
      if the_pix.p_xyz[axis] == 0 or the_pix.p_xyz[axis] == the_grid.n_pix[axis] - 1:
        boundary = True

  # adjust the loop start and end based on the grid dimensions
  for axis in range(3):
    if the_grid.n_pix[axis] == 1:
      l_start[axis] = 1
      l_end[axis] = 2

  nnp = 0  # number of neighbors
  for xpos in range(l_start[0], l_end[0]):
    for ypos in range(l_start[1], l_end[1]):
      for zpos in range(l_start[2], l_end[2]):
        keep_neighbor = True

        if not use_pbc and boundary:
          if the_pix.p_xyz[0] == 0 and xpos == 0:
            keep_neighbor = False
          elif the_pix.p_xyz[0] == the_grid.n_pix[0] and xpos == 2:
            keep_neighbor = False
          elif the_pix.p_xyz[1] == 0 and ypos == 0:
            keep_neighbor = False
          elif the_pix.p_xyz[1] == the_grid.n_pix[1] and ypos == 2:
            keep_neighbor = False
          elif the_pix.p_xyz[2] == 0 and zpos == 0:
            keep_neighbor = False
          elif the_pix.p_xyz[2] == the_grid.n_pix[2] and zpos == 2:
            keep_neighbor = False

        if keep_neighbor:
          # calculate the neighbor id
          nid = the_pix.pid + pmod[xpos] + pmod[ypos] * the_grid.n_pix[0] + pmod[zpos] * the_grid.n_xy
          if use_pbc:
            nid += pbc_shift[xpos][ypos][zpos]
          the_pix.pixel_neighbors[nnp] = nid
          nnp += 1

  the_pix.neighbors = nnp
```

C.6   Python code: inter-atomic distance calculation

```python
# evaluating the interatomic distance between 2 pixel atoms
# - bool use_pbc      : flag to set if PBC are used or not
# - pixel_atom * at_i : pointer to first pixel atom
# - pixel_atom * at_j : pointer to second pixel atom
def evaluate_distance(use_pbc : bool, at_i : PixelAtom, at_j : PixelAtom):
    dist = Distance()   # placeholder for the distance data structure
    Rij = np.zeros(3)   # initialize the distance vector
    # calculating the distance components between atoms
    for axis in range(3):
        Rij[axis] = at_i.coord[axis] - at_j.coord[axis]

    if use_pbc:
        # then the pixel_atom's coordinates are in corrected fractional format
        for axis in range(3):
            Rij[axis] = Rij[axis] - round(Rij[axis])

        # transform back to Cartesian coordinates
        # matrix_multiplication is assumed to be defined elsewhere
        Rij = matrix_multiplication(frac_to_cart, Rij)

    # calculating the squared distance (no square root for efficiency)
    dist.length = np.sum(Rij ** 2)
    dist.Rij = Rij  # store the distance vector components

    # returning the 'distance' data structure that contains:
    # - the squared value for Dij
    # - the components of the distance vector on x, y, and z
    return dist
```

C.7 Python code: pixel search for first neighbor atoms

```python
def pixel_search_for_neighbors(use_pbc : bool):
  # pointer to the pixel grid for analysis
  all_pixels = prepare_pixel_grid(use_pbc)

  # for all pixels in the grid
  for pix in range(all_pixels.pixels):
    # setting pix_i as pointer to pixel number pix
    pix_i = all_pixels.pixel_list[pix]

    # if pixel pix_i contains atom(s)
    if pix_i.patoms:
      # search for neighbor pixels
      find_pixel_neighbors(use_pbc, all_pixels, pix_i)

      # testing all pix_i neighbor pixels
      for pid in range(pix_i.neighbors):
        pjx = pix_i.pixel_neighbors[pid]
        # Setting pix_j as pointer to pixel number pjx
        pix_j = all_pixels.pixel_list[pjx]

        # checking pixel pix_j if it:
        # - contains atom(s)
        # - was not tested, otherwise the analysis would have been performed already
        if pix_j.patoms and not pix_j.tested:
          # If pix_i and pix_j are the same, only test pair of different atoms
          end = 0 if pjx != pix else 1

          # for all atom(s) in pix_i
          for aid in range(pix_i.patoms - end):
            # Set pointer to the first atom to test
            at_i = pix_i.pix_atoms[aid]
            start = 0 if pjx != pix else aid + 1

            # for all atom(s) in pix_j
            for bid in range(start, pix_j.patoms):
              # Set pointer to the second atom to test
              at_j = pix_j.pix_atoms[bid]

              # evaluate interatomic distance
              Dij = evaluate_distance(use_pbc, at_i, at_j)

              if Dij.length < cutoff_squared:
                # This is a bond!
                pass

      # store that pixel pix_i was tested
      pix_i.tested = True
```