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

Sébastien Le Roux

This LiveCoMS document is maintained online on GitHub at https://github.com/Slookeur/Bonds; to provide feedback, suggestions, or help improve it, please visit the GitHub repository and participate via the issue tracker.

This version dated February 27, 2026

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:

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

With:

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

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

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

and:

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

The lattice volume:

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

can then be calculated using:

$$V = \mathrm{A}\times\mathrm{B}\times\mathrm{C}\times\mathbf{Z} \quad (6)$$

With:

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

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:

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

Where the transformation matrix $\mathbf{T_f}$ is defined as:

$$\mathbf{T_f} = \begin{bmatrix} \frac{1}{\mathrm{A}} & -\frac{\cos\gamma}{\mathrm{A}\sin\gamma} & \frac{\cos\alpha\cos\gamma-\cos\beta}{\mathrm{A}\mathbf{Z}\sin\gamma} \\ 0.0 & \frac{1}{\mathrm{B}\,\sin\gamma} & \frac{\cos\alpha\cos\gamma-\cos\beta}{\mathrm{B}\mathbf{Z}} \\ 0.0 & 0.0 & \frac{\sin\gamma}{\mathrm{C}\mathbf{Z}} \end{bmatrix} \quad (9)$$

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:

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

Where the transformation matrix $\mathbf{T_c}$ is defined as:

$$\mathbf{T_c} = \mathbf{T_f}^{-1} = \begin{bmatrix} \mathrm{A} & \mathrm{B}\cos\gamma & \mathrm{C}\cos\beta \\ 0.0 & \mathrm{B}\sin\gamma & \mathrm{C}\,\frac{\cos\alpha-\cos\beta\cos\gamma}{\sin\gamma} \\ 0.0 & 0.0 & \frac{\mathrm{C}\mathbf{Z}}{\sin\gamma} \end{bmatrix} \quad (11)$$

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 $\mathrm{R}_{cut}(\alpha,\beta)$, 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 $\mathrm{R}_{cut}(\alpha,\beta)$ 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},

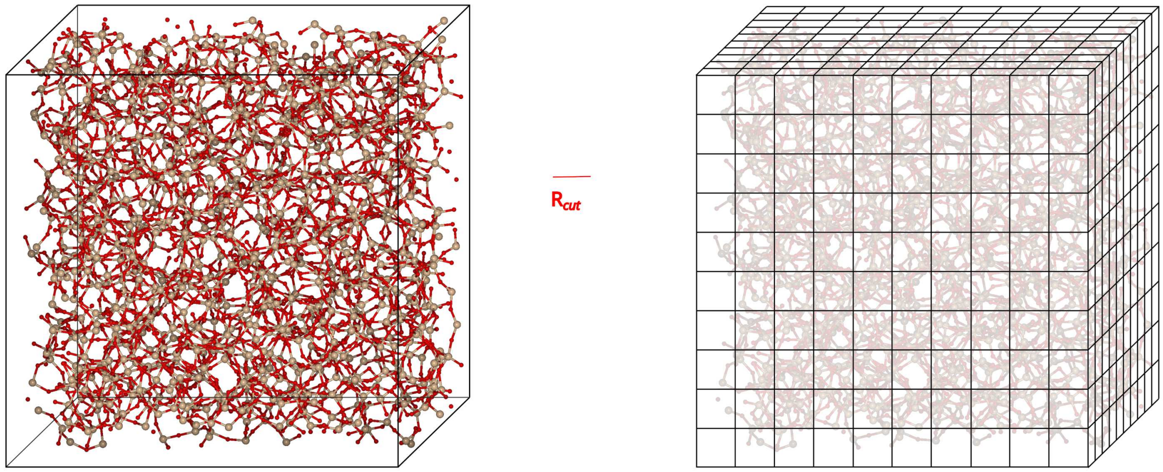


Figure 1. Pixelation of the model box

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:

$$n_p(axis) = \left\lceil \frac{D_{max}(axis)}{\mathrm{R}_{cut}} \right\rceil \tag{12}$$

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:

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

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:

$$pixels = n_p(x) \times n_p(y) \times n_p(z) \tag{14}$$

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:

$$p_{axis} = \left\lfloor \frac{r_{axis} - min_{axis}}{\mathrm{R}_{cut}} \right\rfloor \tag{15}$$

With:

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

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:

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

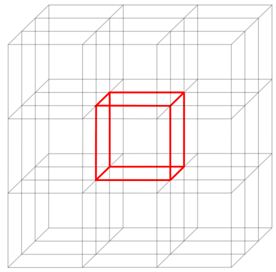
With:

$$P_{id} \in [0, pixels - 1] \tag{18}$$

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:

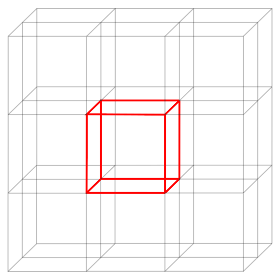
- Pixel inside the pixel grid:



Self + 26 neighbors

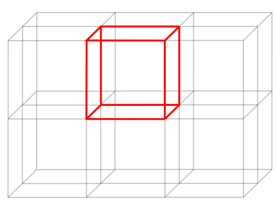
- Pixel on the boundary of the pixel grid : $p_{axis} = 0$ or $p_{axis} = n_p(axis) - 1$

Face of the pixel grid



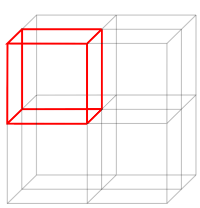
Self + 17 neighbors

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:

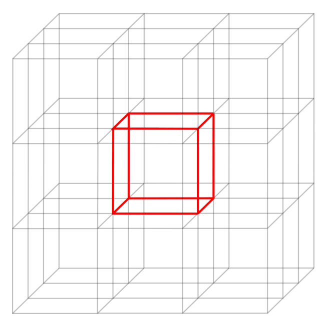
1. Evaluate the number of pixel(s) on each axis, $n_p(x)$, $n_p(y)$ and $n_p(z)$, using equations 12 and 13.
2. Convert atomic Cartesian coordinates to fractional coordinates using $\mathbf{T_f}$.
   Using the transformation to fractional coordinates is the easiest way to compute the distance between 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:

   $$f_{c,axis} = f_{axis} - \lfloor f_{axis} \rfloor \quad \text{with} \quad 0 \leq f_{c,axis} < 1 \tag{19}$$

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

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

4. Determine each pixel neighbors:
   - Pixel inside the pixel grid:



Self + 26 neighbors, no PBC transformation required.

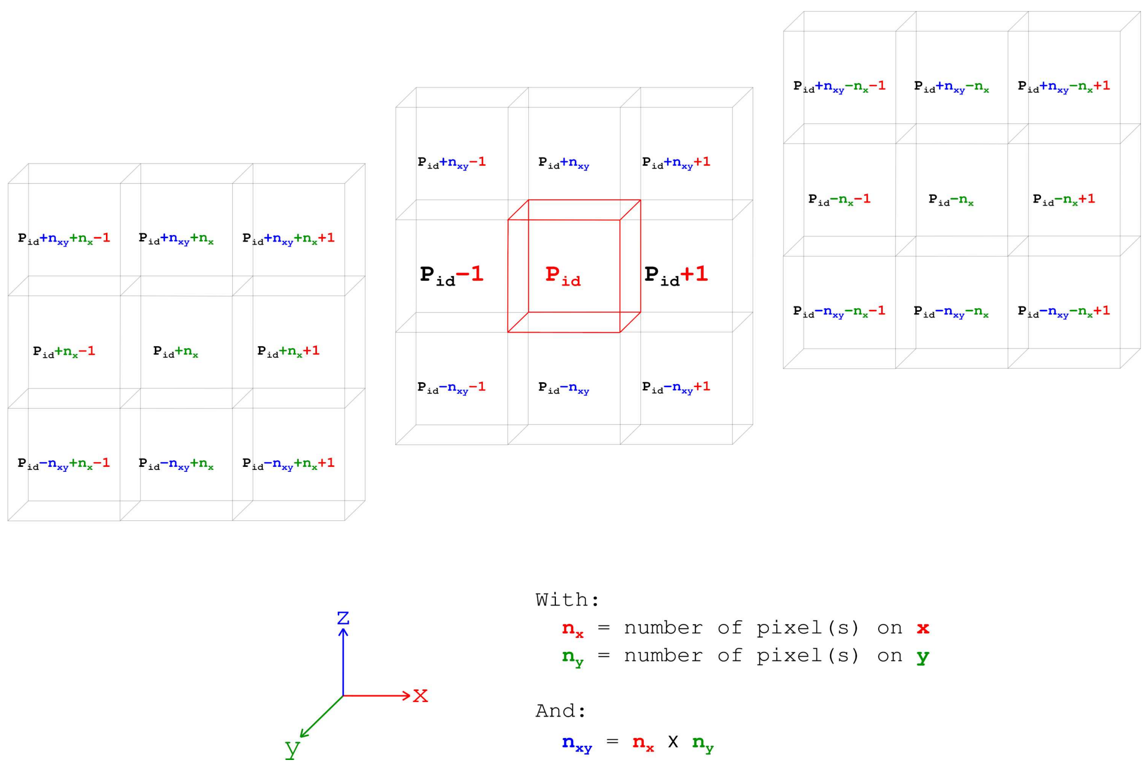
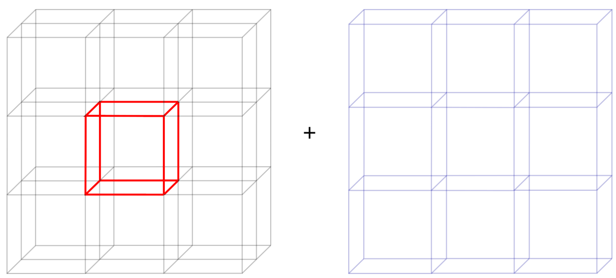


Figure 2. Finding pixel neighbors for pixel P_{id}: operation(s) on each axis are illustrated in the appropriate color(s)

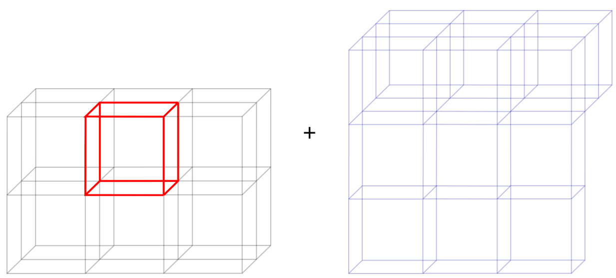
- Pixel on the boundary of the pixel grid :
  $p_{axis} = 0$ or $p_{axis} = n_p(axis) - 1$

Face of the pixel grid:



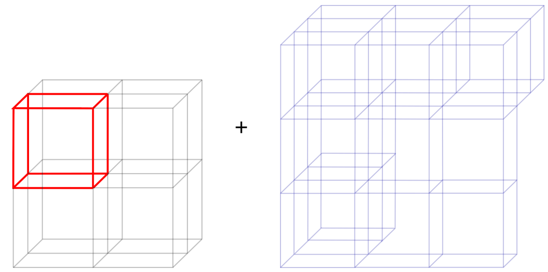
Self + 17 + 9 neighbors using PBC

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:

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

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

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

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

$$\vec{r}(ab) = \mathrm{T}_{\mathbf{c}} \times \vec{f}(ab) \tag{25}$$

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

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

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

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.6.
- FORTRAN90 codes: B.1 through B.6.
- Python codes: C.1 through C.6.
  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 $n_p(axis) = \left\lceil \frac{D_{max}(axis)}{\mathrm{R}_{cut}} \right\rceil$
- ☐ 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 $axis$: $p_{axis} = \left\lfloor \frac{r_{axis} - min_{axis}}{\mathrm{R}_{cut}} \right\rfloor$
  - ☐ 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: $D_{max}(x) = \mathrm{A}$, $D_{max}(y) = \mathrm{B}$ and $D_{max}(z) = \mathrm{C}$
- ☐ Determine the number of pixels, $n_p(axis)$, on x, y and z using $n_p(axis) = \left\lceil \frac{D_{max}(axis)}{\mathrm{R}_{cut}} \right\rceil$
- ☐ 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_co[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
};

// 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 (grid pixel_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; // at first there is no 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] = pixel_grid->n_pix[0];
      }
    }
  }
  else if ( pixel_coord[0] == pixel_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] = - pixel_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] += pixel_grid->n_xy;
      }
    }
  }
  else if ( pixel_coord[1] == pixel_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] -= pixel_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] += pixel_grid->pixels;
      }
    }
  }
  else if ( pixel_coord[2] == pixel_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] -= pixel_grid->pixels;
      }
    }
  }
}
```

A.3 C code: finding pixel neighbors

```c
// Finding neighbor pixels for pixel in the grid
// - bool use_pbc : flag to set if PBC are used or not
// - grid * the_gird : pointer to the pixel grid
// - pixel * the_pix : pointer to the pixel with neighbors to be found
void find_pixel_neighbors (bool use_pbc, grid * the_grid, pixel * the_pix)
{
  int axis;                       // axis loop iterator
  int xpos, ypos, zpos;           // neighbor position on x, y and z
  int l_start[3] = { 0, 0, 0};    // loop iterators starting value
  int l_end[3] = { 3, 3, 3};      // loop iterators ending value
  int pmod[3] = {-1, 0, 1};       // position modifiers
  int nnp;                        // number of neighbors for pixel
  int nid;                        // neighbor id for pixel
  int pbc_shift[3][3][3];         // shift for pixel neighbor number due to PBC
  bool boundary = FALSE;          // is pixel on the boundary of the grid
  bool keep_neighbor = TRUE;      // keep or not neighbor during analysis

if ( use_pbc )
  {
    set_pbc_shift (the_grid, the_pix->p_co, pbc_shift);
  }
  else
  {
    for ( axis = 0 ; axis < 3 ; axis ++ )
    {
      if ( the_pix->p_co[axis] == 0 || the_pix->p_co[axis] == the_grid->n_pix[axis] - 1 ) boundary = TRUE;
    }
  }
  for ( axis = 0 ; axis < 3 ; axis ++ )
  {
    if ( the_grid->n_pix[axis] == 1 )
    {
      l_start[axis] = 1;
      l_end[axis] = 2;
    }
  }
  nnp = 0;
  for ( xpos = l_start[0] ; xpos < l_end[0] ; xpos ++ )
  {
    for ( ypos = l_start[1] ; ypos < l_end[1] ; ypos ++ )
    {
      for ( zpos = l_start[2] ; zpos < l_end[2] ; zpos ++ )
      {
        keep_neighbor = TRUE;
        if ( ! use_pbc && boundary )
        {
          if (( the_pix->p_co[0] == 0 && xpos == 0 ) || ( the_pix->p_co[0] == the_grid->n_pix[0] && xpos == 2 ))
          {
            keep_neighbor = FALSE;
          }
          else if (( the_pix->p_co[1] == 0 && ypos == 0 ) || ( the_pix->p_co[1] == the_grid->n_pix[1] && ypos == 2 ))
          {
            keep_neighbor = FALSE;
          }
          else if (( the_pix->p_co[2] == 0 && zpos == 0 ) || ( the_pix->p_co[2] == the_grid->n_pix[2] && zpos == 2 ))
          {
            keep_neighbor = FALSE;
          }
        }
        if ( keep_neighbor )
        {
          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 ++ ;
        }
      }
    }
  }
  the_pix->neighbors = nnp;
}
```

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;                 // integer loop axis id (0=x, 1=y, 2=z)
  int aid;                  // integer loop 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

  // User defined function to allocate the memory to store the pixel grid data
  grid = allocate_grid_data ();
  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
  // User defined function to allocate the memory to store the pixel information for the grid
  grid->pixel_list = allocate_pixel_data (grid->pixels);
  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 + 1;
      // User defined function to:
      // - Add atom 'aid' with coordinates 'c_coord[aid]' to pixel 'pixel_number'
      // - Increment the number of atom(s) in pixel 'pixel_number'
      // - If needed (for the first atom) set pixel coordinates in the grid to 'pixel_pos'
      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] * n_pix[axis]);
      }
      pixel_num = pixel_pos[0] + pixel_pos[1] * grid->n_pix[0] + pixel_pos[2] * grid->n_xy + 1;
      add_atom_to_pixel (grid, pixel_num, pixel_pos, aid, f_coord); // User defined function (see above)
    }
  }
  return grid;
}
```

A.5 C code: inter-atomic distance calculation

```c
// 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
distance evaluate_distance (bool use_pbc, pixel_atom * at_i, pixel_atom * at_j)
{
  int axis;           // integer parameter loop iterator
  float u, v;         // float parameters
  distance dist;      // distance data to store calculation results
  for ( axis = 0 ; axis < 3 ; axis ++ )
  {
    dist.Rij[axis] = at_i->coord[axis] - at_j->coord[axis];
  }
  if ( use_pbc )
  {
    // Pixel atom's coordinates are in corrected fractional format
    for ( axis = 0 ; axis < 3 ; axis ++ )
    {
      // Absolute value in float format
      u = fabs (dist.Rij[axis]);
      v = min (u, 1.0 - u);
      // Proper value, with proper sign
      dist.Rij[axis] = (dist.Rij[axis] / u) * v;
    }
    // Transform back to Cartesian coordinates
    // with 'matrix_multiplication' a user defined function to perform the operation
    dist.Rij = matrix_multiplication (frac_to_cart, dist.Rij);
  }
  dist.length = 0.0;
  for ( axis = 0 ; axis < 3 ; axis ++ )
  {
    dist.length += dist.Rij[axis] * dist.Rij[axis];
  }
  // 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
  return dist;
}
```

A.6 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 start, 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 '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
      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_grid->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
          end = (pjx != pix) ? 0 : 1
          // For all atom(s) in 'pix'
          for ( aid = 0 ; aid < pix_i->patoms - end ; aid ++ )
          {
            // Set pointer to the first atom to test
            at_i = & pix_i->pix_atom[aid];
            start = (pjx != pix) ? 0 : aid + 1
            // For all atom(s) in 'pix_j'
            for ( bid = start ; bid < pix_j->patoms ; bid ++ )
            {
              // Set pointer to the second atom to test
              at_j = & pix_j->pix_atom[bid];
              // Evaluate interatomic distance
              Dij = evaluate_distance (use_pbc, at_i, at_j);
              if ( Dij.length < cutoff_squared )
              {
                // This is a 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

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

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

  ! Pixel data structure
  TYPE pixel
    INTEGER                                     :: pid             ! the pixel number
    INTEGER, DIMENSION(3)                       :: p_co            ! the pixel coordinates in the grid
    LOGICAL                                     :: tested          ! was the pixel checked already
    INTEGER                                     :: patoms          ! number of atom(s) in pixel
    TYPE(atom), DIMENSION(:), ALLOCATABLE       :: pix_atom        ! 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 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     :: pixel_list      ! pointer to the pixels, to be allocated
  END TYPE grid

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

  ! 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 atomic bond(s)
  REAL                                          :: cutoff_squared  ! squared value for the cutoff

  ! 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 Commented FORTRAN90 code: set pixel periodic boundary condition shift

```
SUBROUTINE set_pbc_shift (the_grid, pixel_coord, pbc_shift)

  USE parameters
  IMPLICIT NONE

  TYPE (grid), INTENT(IN)                    :: the_grid            ! the pixel grid
  INTEGER, DIMENSION(3), INTENT(IN)          :: pixel_coord         ! the pixel coordinates in the grid
  INTEGER, DIMENSION(3,3,3), INTENT(INOUT)   :: pbc_shift           ! the shift, correction, to be calculated

  pbc_shift(:,:,:) = 0                                              ! at first there is no shift

  if ( pixel_coord(1) .eq. 1 ) then                                 ! pixel position on 'x' is min
    pbc_shift(1,:,:) = the_grid%n_pix(1)
  else if ( pixel_coord(1) .eq. the_grid%n_pix(1) ) then            ! pixel position on 'x' is max
    pbc_shift(3,:,:) = - the_grid%n_pix(1)
  endif

  if ( pixel_coord(2) .eq. 1 ) then                                 ! pixel position on 'y' is min
    pbc_shift(:,1,:) = pbc_shift(:,1,:) + the_grid%n_xy
  else if ( pixel_coord(2) .eq. the_grid%n_pix(2) ) then            ! pixel position on 'y' is max
    pbc_shift(:,3,:) = pbc_shift(:,3,:) - the_grid%n_xy
  endif

  if ( pixel_coord(3) .eq. 1 ) then                                 ! pixel position on 'z' is min
    pbc_shift(:,:,1) = pbc_shift(:,:,1) + the_grid%pixels
  else if ( pixel_coord(3) .eq. the_grid%n_pix(3) ) then            ! pixel position on 'z' is max
    pbc_shift(:,:,3) = pbc_shift(:,:,3) - the_grid%pixels
  endif

END SUBROUTINE set_pbc_shift
```

B.3 Commented FORTRAN90 code: finding pixel neighbors

```fortran
SUBROUTINE find_pixel_neighbors (use_pbc, the_grid, the_pix)

  USE parameters
  IMPLICIT NONE

  LOGICAL, INTENT(IN)          :: use_pbc                     ! flag to set if PBC are used or no
  TYPE (grid), INTENT(INOUT)   :: the_grid                    ! pointer to the pixel grid
  TYPE (pixel), INTENT(INOUT)  :: the_pix                     ! pointer to the pixel
  INTEGER                      :: axis                        ! loop iterator axis id (1=x , 2=y , 3=z)
  INTEGER                      :: xpos, ypos, zpos            ! neighbor position on x, y and z
  INTEGER, DIMENSION(3)        :: l_start = (\1, 1, 1\)       ! loop iterators starting value
  INTEGER, DIMENSION(3)        :: l_end = (\3, 3, 3\)         ! loop iterators ending value
  INTEGER, DIMENSION(3)        :: pmod = (\-1, 0, 1\)         ! position modifiers
  INTEGER                      :: nnp                         ! number of neighbors for pixel
  INTEGER                      :: nid                         ! neighbor id for pixel
  INTEGER, DIMENSION(3,3,3)    :: pbc_shift                   ! shift, correction, due to PBC
  LOGICAL                      :: boundary=.false.            ! is pixel on the boundary of the grid
  LOGICAL                      :: keep_neighbor = .true.      ! keep or not neighbor during analysis

  if ( use_pbc ) then
    call set_pbc_shift (the_grid, the_pix%p_co, pbc_shift)
  else
    do axis = 1 , 3
      if ( the_pix%p_co(axis) .eq. 1 .or. the_pix%p_co(axis) .eq. the_grid%n_pix(axis) ) then
        boundary = .true.
      endif
    enddo
  endif
  do axis = 1 , 3
    if ( the_grid%n_pix(axis) .eq. 1 ) then
      l_start(axis) = 2
      l_end(axis) = 2
    endif
  enddo
  nnp = 0
  do xpos = l_start(1) , l_end(1)
    do ypos = l_start(2) , l_end(2)
      do zpos = l_start(3) , l_end(3)
        keep_neighbor = .true.
        if ( .not. use_pbc .and. boundary ) then
          if ( the_pix%p_co(1) .eq. 1 .and. xpos .eq. 1 ) then
            keep_neighbor = .false.
          else if ( the_pix%p_co(1) .eq. the_grid%n_pix(1) .and. xpos .eq. 3 ) then
            keep_neighbor = .false.
          else if ( the_pix%p_co(2) .eq. 1 .and. ypos .eq. 1 ) then
            keep_neighbor = .false.
          else if ( the_pix%p_co(2) .eq. the_grid%n_pix(2) .and. ypos .eq. 3 ) then
            keep_neighbor = .false.
          else if ( the_pix%p_co(3) .eq. 1 .and. zpos .eq. 1 ) then
            keep_neighbor = .false.
          else if ( the_pix%p_co(3) .eq. the_grid%n_pix(3) .and. zpos .eq. 3 ) then
            keep_neighbor = .false.
          endif
        endif
        if ( keep_neighbor ) then
          ! Evaluating neighbor pixel number in the grid
          nid = the_pix%pid + pmod(xpos) + pmod(ypos) * the_grid%n_pix(1) + pmod(zpos) * the_grid%n_xy
          if ( use_pbc ) then
            ! Correcting the value if PBC are used
            nid = nid + pbc_shift(xpos, ypos, zpos)
          endif
          the_pix%pixel_neighbors(nnp) = nid
          nnp = nnp + 1
        endif
      enddo
    enddo
  enddo
  the_pix%neighbors = nnp

END SUBROUTINE find_pixel_neighbors
```

B.4 Commented FORTRAN90 code: preparation of the pixel grid

```fortran
! Preparation of the pixel grid
SUBROUTINE prepare_pixel_grid (use_pbc, grid)

  USE parameters
  IMPLICIT NONE

  LOGICAL, INTENT(IN)        :: use_pbc                ! flag to set if PBC are used or not
  TYPE (grid), INTENT(INOUT) :: grid                   ! the pixel grid to prepare
  INTEGER                    :: axis                   ! loop iterator axis id (1=x, 2=y, 3=z)
  INTEGER                    :: aid                    ! loop iterator atom number (1, atoms)
  INTEGER                    :: pixel_num              ! pixe number in the grid
  INTEGER, DIMENSION(3)      :: pixel_pos              ! pixel coordinates in the grid
  REAL, DIMENSION(3)         :: cmin, cmax             ! real precision coordinates min, max
  REAL, DIMENSION(3)         :: f_coord                ! real precision fractional coordinates

  if ( .not. use_pbc ) then                            ! Without periodic boundary conditions
    do axis = 1 , 3
      cmin(axis) = c_coord(1,axis)
      cmax(axis) = c_coord(1,axis)
    enddo
    do aid = 2 , atoms                                 ! For all atoms
      do axis = 1 , 3                                  ! For x, y and z
        cmin(axis) = min(cmin(axis), c_coord(aid,axis))
        cmax(axis) = max(cmax(axis), c_coord(aid,axis))
      enddo
    enddo
    do axis = 1 , 3                                    ! For x, y and z
      ! Number of pixel(s) on axis 'axis'
      grid%n_pix(axis) = INT((cmax(axis) - cmin(axis)) / cutoff) + 1
    enddo
  else                                                 ! Using periodic boundary conditions
    do axis = 1 , 3                                    ! For x, y and z
      ! Number of pixel(s) on axis 'axis'
      grid%n_pix(axis) = INT(l_params(axis) / cutoff) + 1
    enddo
  endif
  do axis = 1 , 3                                      ! For x, y and z
    ! Correction if the number of pixel(s) on 'axis' is too small
    if ( grid%n_pix(axis) .lt. 4 ) then
      grid%n_pix(axis) = 1
    endif
  enddo

  grid%n_xy = grid%n_pix(1) * grid%n_pix(2)            ! Number of pixels on the plan 'xy'
  grid%pixels = grid%n_xy * grid%n_pix(3)              ! Total number of pixels in the grid
  ! User defined function to allocate the memory to store the pixel information for the grid
  grid%pixel_list = allocate_pixel_data (grid%pixels);

  if ( .not. use_pbc ) then                            ! Without periodic boundary conditions
    do aid = 1 , atoms                                 ! For all atoms
      do axis = 1 , 3                                  ! For x, y and z
        pixel_pos(axis) = INT( (c_coord(aid,axis) - cmin(axis) )/ cutoff)
      enddo
      pixel_num = pixel_pos(1) + pixel_pos(2) * grid%n_pix(1) + pixel_pos(3) * grid%n_xy + 1
      ! User defined function to:
      !   - Add atom 'aid' with coordinates 'c_coord[aid]' to pixel 'pixel_number'
      !   - Increment the number of atom(s) in pixel 'pixel_number'
      !   - If needed (for the first atom) set pixel coordinates in the grid to 'pixel_pos'
      call add_atom_to_pixel (grid, pixel_num, pixel_pos, aid, c_coord(aid))
    enddo
  else                                                 ! Using periodic boundary conditions
    do aid = 1 , atoms                                 ! For all atoms
     f_coord = MATMUL ( c_coord(aid), cart_to_frac )
     do axis = 1 , 3                                   ! For x, y and z
       f_coord(axis) = f_coord(axis) - floor(f_coord(axis))
       pixel_pos(axis) = INT(f_coord(axis) * n_pix(axis))
     enddo
     pixel_num = pixel_pos(1) + pixel_pos(1) * grid%n_pix(2) + pixel_pos(3) * grid%n_xy + 1
     call add_atom_to_pixel (grid, pixel_num, pixel_pos, aid, f_coord) ! See above
    enddo
  endif

END SUBROUTINE prepare_pixel_grid
```

B.5 Commented FORTRAN90 code: inter-atomic distance calculation

```
! 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 (atom), INTENT(IN)         :: at_i            ! first pixel atom
  TYPE (atom), 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
    ! 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.6 Commented FORTRAN90 code: pixel search for first neighbor atoms

```fortran
SUBROUTINE pixel_search_for_neighbors (use_pbc)

  USE parameters
  IMPLICIT NONE

  LOGICAL, INTENT(IN)   :: use_pbc                       ! flag to set if PBC are used or not
  TYPE (grid)           :: all_pixels                    ! the pixel grid to analyze
  INTEGER               :: pix, pjx                      ! integer pixel ID numbers
  INTEGER               :: aid, bid                      ! integer loop atom numbers
  INTEGER               :: lstart, lend                  ! integer loop modifier
  INTEGER               :: pid
  TYPE (pixel), POINTER :: pix_i, pix_j                  ! pointers of pixel data structure
  TYPE (atom), POINTER  :: at_i, at_j                    ! pointers on pixel_atom data structure
  TYPE (distance)       :: Dij                           ! distance data structure

  call prepare_pixel_grid (use_pbc, all_pixels)
  ! Note that 'all_pixels' must be prepared before the following
  ! For all pixels in the grid
  do pix = 1 , 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 .gt. 0 ) then
      ! Testing all 'pix_i' neighbor pixels
      do pid = 1 , 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 .gt. 0 .and. .not. pix_j%tested ) then
          ! If 'pix_i' and 'pix_j' are the same, only test pair of different atoms
          if ( pjx .eq. pix ) then
            lend = 1
          else
            lend = 0
          endif
          ! For all atom(s) in 'pix_i'
          do aid = 1 , pix_i%patoms - lend
            ! Set pointers to the first atom to test
            at_i => pix_i%pix_atom(aid)
            if ( pjx .eq. pix ) then
              lstart = aid + 1
            else
              lstart = 1
            endif
            ! For all atom(s) in 'pix_j'
            do bid = lstart , pix_j%patoms
              ! Set pointers to the second atom to test
              at_j => pix_j%pix_atom(bid)
              ! Evaluate interatomic distance
              call evaluate_distance (use_pbc, at_i, at_j, Dij)
              if ( Dij%length .lt. cutoff_squared ) then
                ! This is a bond !
              endif
            enddo
          enddo
        endif
      enddo
      ! Store that pixel 'pix_i' was tested
      pix_i%tested = .true.
    endif
  enddo

END SUBROUTINE pixel_search_for_neighbors
```

Appendix C Commented Python code

C.1 Commented Python code: data structures and global variables

```python
# Global data structures and variables used in the next code sections
import numpy as np

# Atom in pixel data structure
class PixelAtom:
  def __init__(self, atom_id=0, coord=None):
    self.atom_id = atom_id                                           # the atom ID
    self.coord = np.zeros(3) if coord is None else np.array(coord)   # the atom coordinates on x, y and z

# Pixel data structure
class Pixel:
  def __init__(self, pid=0, p_co=None, tested=False, patoms=0, pix_atoms=None, neighbors=0):
    self.pid = pid                                             # the pixel number
    self.p_co = np.zeros(3) if p_co is None else np.array(p_co)  # the pixel coordinates in the grid
    self.tested = tested                                       # was the pixel checked already
    self.patoms = patoms                                       # number of atom(s) in pixel
    self.pix_atoms = [] if pix_atoms is None else pix_atoms      # list of atom(s) in the pixel
    self.neighbors = neighbors                                 # number of neighbors for pixel
    self.pixel_neighbors = np.zeros(27, dtype=int)             # the list of neighbor pixels, maximum 27

# Pixel grid data structure
class PixelGrid:
  def __init__(self, pixels=0, n_pix=None, n_xy=0, pixel_list=None):
    self.pixels = pixels                                       # total number of pixels in the grid
    self.n_pix = np.zeros(3, dtype=int) if n_pix is None else np.array(n_pix)   # pixel(s) on each axis
    self.n_xy = n_xy                                           # number of pixels in the plan xy
    self.pixel_list = [] if pixel_list is None else pixel_list   # pointer to the pixels, to be allocated

# Bond distance data structure
class Distance:
  def __init__(self, length=0.0, Rij=None):
    self.length = length                                       # the distance in Å squared
    self.Rij = np.zeros(3) if Rij is None else np.array(Rij)     # vector components of x, y and z

# Model description
atoms = 0                         # the total number of atom(s)
c_coord = None                    # list of Cartesian coordinates: c_coord[atoms][3]
cutoff = 0.0                      # the cutoff to define atomic bond(s)
cutoff_squared = 0.0              # squared value for the cutoff

# Model box description
l_params = np.zeros(3)            # lattice a, b and c
cart_to_frac = np.zeros((3, 3))   # Cartesian to fractional coordinates matrix
frac_to_cart = np.zeros((3, 3))   # fractional to Cartesian coordinates matrix
```

C.2 Commented 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        # at first there is no 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 Commented 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_co, pbc_shift)
  else:
    for axis in range(3):
      if the_pix.p_co[axis] == 0 or the_pix.p_co[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_co[0] == 0 and xpos == 0:
            keep_neighbor = False
          elif the_pix.p_co[0] == the_grid.n_pix[0] and xpos == 2:
            keep_neighbor = False
          elif the_pix.p_co[1] == 0 and ypos == 0:
            keep_neighbor = False
          elif the_pix.p_co[1] == the_grid.n_pix[1] and ypos == 2:
            keep_neighbor = False
          elif the_pix.p_co[2] == 0 and zpos == 0:
            keep_neighbor = False
          elif the_pix.p_co[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.neighbor_list[nnp] = nid
          nnp += 1

  the_pix.neighbors = nnp
```

C.4 Commented Python code: preparation of the pixel grid

```python
# Preparation of the pixel grid
# - bool use_pbc : flag to set if PBC are used or not
def prepare_pixel_grid(use_pbc : bool):
  grid = PixelGrid()                               # Create a new pixel grid
  cmin = [float('inf')] * 3                        # Initialize to infinity
  cmax = [-float('inf')] * 3                       # Initialize to negative infinity
  pixel_pos = np.zeros(3, dtype=int)
# User defined function to allocate the memory to store the pixel grid data
  grid = allocate_grid_data()

  if not use_pbc:                                  # Without periodic boundary conditions
    for axis in range(3):
      cmin[axis] = cmax[axis] = c_coord[0][axis]
    for aid in range(1, atoms):                    # For all atoms
      for axis in range(3):                        # 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 in range(3):                          # For x, y and z
      grid.n_pix[axis] = int((cmax[axis] - cmin[axis]) / cutoff) + 1  # Number of pixels on axis 'axis'
  else:                                            # Using periodic boundary conditions
    for axis in range(3):                          # For x, y and z
      grid.n_pix[axis] = int(l_params[axis] / cutoff) + 1  # Number of pixels on axis 'axis'

  for axis in range(3):                            # For x, y and z
    # Correction if the number of pixels on 'axis' is too small
    grid.n_pix[axis] = 1 if grid.n_pix][axis] < 4 else grid.n_pix[axis]

  grid.n_xy = grid.p_pix[0] * grid.n_pix[1] # Number of pixels on the plan 'xy'
  grid.pixels = grid.n_xy * grid.p_pix[2]  # Total number of pixels in the grid

  # User defined function to allocate the memory to store the pixel information for the grid
  grid.pixel_list = allocate_pixel_data(grid.pixels)

  if not use_pbc:                                  # Without periodic boundary conditions
    for aid in range(atoms):                       # For all atoms
      for axis in range(3):                        # 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 + 1
      # User defined function to:
      # - Add atom 'aid' with coordinates 'c_coord[aid]' to pixel 'pixel_number'
      # - Increment the number of atom(s) in pixel 'pixel_number'
      # - If needed (for the first atom) set pixel coordinates in the grid to 'pixel_pos'
      add_atom_to_pixel(grid, pixel_num, pixel_pos, aid, c_coord[aid])
  else:  # Using periodic boundary conditions
    for aid in range(atoms):                       # 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 in range(3):                        # For x, y and z
        f_coord[axis] = f_coord[axis] - np.floor(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 + 1
      add_atom_to_pixel(grid, pixel_num, pixel_pos, aid, f_coord)  # User defined function (see above)

  return grid
```

C.5 Commented Python code: inter-atomic distance calculation

```
# 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:
    # Pixel atom's coordinates are in corrected fractional format
    for axis in range(3):
      # Absolute value in float format
      u = abs(Rij[axis])
      v = min(u, 1.0 - u)
      # Proper value, with proper sign
      Rij[axis] = (Rij[axis] / u) * v

    # 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.6 Commented 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
```