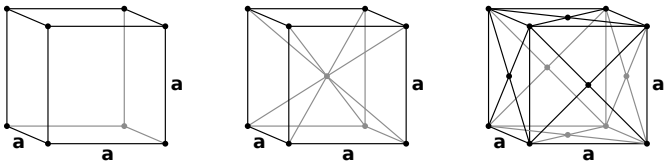
Creating lattice coordination files for simulations and for visualisation purposes

December 24, 2012

J.L.

This short briefing gives an idea how to simulate and visualise common atom configurations (lattices) by means of programming. As a (very) shallow introduction, Figure 1 illustrates the three most common structures of atoms occurring in metals. However. these are not the only possible lattice structures. In general, materials have their individual atoms grouped in several of these kinds of configurations.



(a) SC (b) BCC (c) FCC

Figure 1: The unit cells of common lattice structures

Since the atom configurations are symmetrically distributed, it is possible to generate the structures by means of computer programming. The code for generating lattice blocks obviously has loops for all three dimensions and for generality the size of the lattice is allowed to vary in each coordinate direction. The given example C-code demonstrates a straightforward lattice generator function:

```
/*
* void latticebuilder(int X, int Y, int Z, int *xx, int *yy, int *zz, int type)
*
* A handy function that generates the lattice site coordinates in the
* simulation box in integer values, for SC, BCC or FCC lattice structures.
* The lattice structure is chosen by setting the defined constant
* value for TYPE as 6, 8 or 12 according to the bulk coordination number of
* each lattice type.
*
* INPUT:
*
* 1. The number of full SC/BCC/FCC unit lattices in X-direction.
*    Value fed through main();
* 2. The number of full SC/BCC/FCC unit lattices in Y-direction.
*    Value fed through main();
* 3. The number of full SC/BCC/FCC unit lattices in Z-direction.
*    Value fed through main();
* 4. The pointer to the x-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[0]
* 5. The pointer to the y-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[1]
* 6. The pointer to the z-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[2]
* 7. The bulk coordination number of the used lattice structure
```

```
*    defined as the constant TYPE;
*
* OUTPUT: no return values (void)
*
*/
void latticebuilder(int X, int Y, int Z, int *xx, int *yy, int *zz, int type)
{
  int v = 2, x, y, z, xmax, ymax, zmax, arg;
  float a, b, c;

  xmax = 2*X;
  ymax = 2*Y;
  zmax = 2*Z;

  for (z = 0; z <= zmax; z++) {
    for (y = 0; y <= ymax; y++) {
      for (x = 0; x <= xmax; x++) {
        a = x%v;
        b = y%v;
        c = z%v;

        /* arguments to produce BCC (8), SC (6) and FCC (12) lattices */
        if (type == 8)
          arg = (!a && !b && !c)||(a && b && c);
        else if (type == 6)
          arg = (!a && !b && !c);
        else
          arg = (!a && !b && !c)
              ||(a && b && !c)
              ||(!a && b && c)
              ||(a && !b && c);

        if (arg) {
          *xx++ = x-X;
          *yy++ = y-Y;
          *zz++ = z-Z;
        }
      }
    }
  }
}
```

In this function, the constants X, Y and Z are used for determining the size of the generated lattice. The actual (cartesian) coordinates of the lattice sites are stored in the arrays represented by the pointers xx, yy and zz. The generated lattice configuration is centred around the coordinate origin. Problems might arise if one wants to dynamically allocate the memory for the arrays. Then the number of lattice points should be known before calling this function. One solution is to use the lattice generator function to calculate the number of lattice points. Second solution is to derive a mathematical expression to evaluate the required storage space for the coordination arrays. The code-snippet given below is based on the `latticebuilder()` function and it

calculates the number of atoms in the lattice.

```
/*
* int countlatticeatoms(int type, int X, int Y, int Z)
*
* This function calculates the total number of lattice sites in the
* simulation box for BCC or FCC lattice structures. The lattice structure
* is chosen by setting the value for the constant TYPE as 8 or
* 12 according to the bulk coordination number of each lattice type.
* The expressions are based on the latticebuilder function.
* The extension to SC lattice is not included.
*
* INPUT:
* 1. The bulk coordination number of the used lattice structure
*    defined as a constant TYPE;
* 2. The number of full BCC/FCC unit lattices in X-direction.
*    Value fed through main();
* 3. The number of full BCC/FCC unit lattices in Y-direction.
*    Value fed through main();
* 4. The number of full BCC/FCC unit lattices in Z-direction.
*    Value fed through main();
*
* OUTPUT:
* The total number of lattice sites L in the simulation box.
*
*/
int countlatticeatoms(int type, int X, int Y, int Z)
{
  if (type == 8)
    return (Z + 1)*(Y + 1)*(X + 1) + Z*Y*X;         /* BCC */
  else
    return ( (2*Z + 1)*(2*Y + 1)*(2*X + 1) + 1 )/2; /* FCC */
}
```

The ideology here is to first generate a bulk lattice with unpopulated lattice sites. This means that at this point there are no atoms attached to the lattice structure. With additional functions the bulk lattice can be populated by atom configurations that build up different geometrical shapes. As an example, Figure 2 shows a FCC-lattice based cube. The cube has been generated with the function `markatomsinsiderect()` shown below. This function also takes into account the actual dimensions of the atom structure by assigning the material-dependent lattice constant a to the lattice coordinates.

```
/*
* int markatomsinsiderect(int L, double a, double x, double y, double z,
                          int *xx, int *yy, int *zz, int *c_ptr)
*
* This function fills the list of occupied atom sites using the
* rectangular starting configuration. Function marks the indices of
* the atoms as "1" and indices of the vacancies as "0".  The list
* is an allocated memory area in the heap to which the int *c_ptr
* pointer is associated.
```

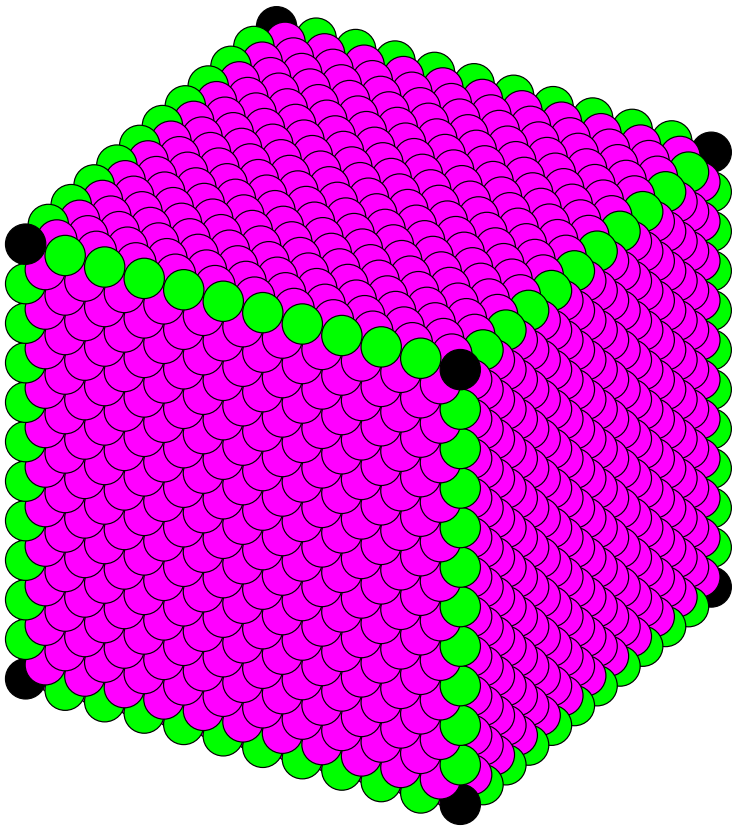


Figure 2: A cube of atoms based on the FCC lattice

```
*
* INPUT:
*
* 1. The total number of lattice sites L defined by
*    countlatticeatoms(); The value is fed through main();
* 2. The lattice constant a
*    defined as the constant value LCONST;
* 3. The size of the rectangular particle in x-direction
*    defined as the constant value XXX;
* 4. The size of the rectangular particle in y-direction
*    defined as the constant value YYY;
* 5. The size of the rectangular particle in z-direction
*    defined as the constant value ZZZ;
* 6. The pointer to the x-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[0]
* 7. The pointer to the y-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[1]
* 8. The pointer to the z-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[2]
* 9. The pointer to the list of occupied lattice sites allocated in
*    main(); associated pointer xyz[3]
*
* OUTPUT:
* The total number of populated atoms N in the lattice.
*
```

```
*/
int markatomsinsiderect(int L, double a, double x, double y, double z,
                        int *xx, int *yy, int *zz, int *c_ptr)
{
  int s, i = 0;
  double b, c, d;

  for (s = 0; s < L; s++) {
    b = a*(*(xx + s))/2;
    c = a*(*(yy + s))/2;
    d = a*(*(zz + s))/2;

    if( ((b/(x/2)) <= 1) && ((b/(x/2)) >= -1)
     && ((c/(y/2)) <= 1) && ((c/(y/2)) >= -1)
     && ((d/(z/2)) <= 1) && ((d/(z/2)) >= -1)) {
      *c_ptr++ = 1;
      i++;
    }
    else
      *c_ptr++ = 0;
  }
  return i;
}
```

The second easily generated geometrical atom structure is the sphere. A visual example of an atomic sphere is given in Figure 3. This atom configuration is also based on the FCC lattice. The function `markatomsinsideradius()`

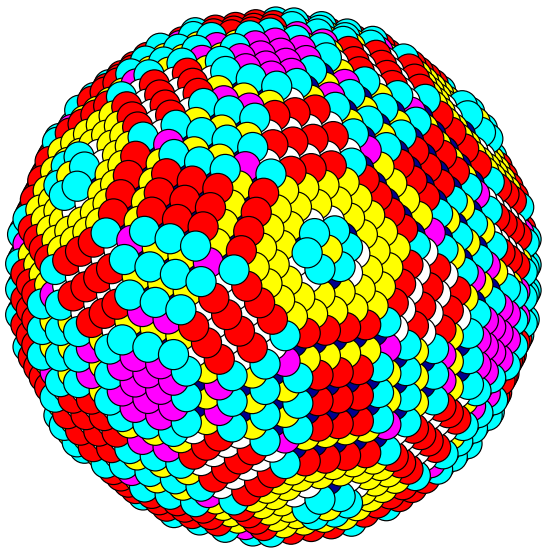


Figure 3: A sphere of atoms based on the FCC lattice

indicating the atoms inside a spherical shell of radius R is given in the code-snippet below. The function scales the atom structure into realistic dimensions according to the lattice constant a.

```
/*
```

```
* int markatomsinsideradius(int L, double a, double R,
                            int *xx, int *yy, int *zz, int *c_ptr)
*
* This function fills the list of occupied atom sites using the
* spherical starting configuration. Function marks the indices of
* the atoms as "1" and indices of the vacancies as "0".  The list
* is an allocated memory area in the heap to which the int *c_ptr
* pointer is associated. The memory allocation is made in main();
* for xyz[3] pointer.
*
* INPUT:
*
* 1. The total number of lattice sites L defined by
*    countlatticeatoms(); The value is fed through main();
* 2. The lattice constant a
*    defined as the constant value LCONST;
* 3. The radius of the starting spherical configuration R
*    defined as the constant value RADIUS;
* 4. The pointer to the x-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[0]
* 5. The pointer to the y-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[1]
* 6. The pointer to the z-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[2]
* 7. The pointer to the list of occupied lattice sites allocated in
*    main(); associated pointer xyz[3]
*
* OUTPUT:
* The total number of populated atoms N in the lattice.
*
*/
int markatomsinsideradius(int L, double a, double R, int *xx,
                          int *yy, int *zz, int *c_ptr)
{
  int s, i = 0;
  double x, y, z;

  for (s = 0; s < L; s++) {
    x = a*(*(xx + s))/2;
    y = a*(*(yy + s))/2;
    z = a*(*(zz + s))/2;

    if ( ( (x/R)*(x/R) + (y/R)*(y/R) + (z/R)*(z/R) ) <= 1) {
      *c_ptr++ = 1;
      i++;
    }
    else
      *c_ptr++ = 0;
  }
  return i;
}
```

The final theoretical geometrical atom structure presented in this example is the truncated octahedron. The truncated octahedron is actually the form

where the total energy of the atom configuration is minimised. The ideal truncated octahedron is shown in Figure 4 and when the truncation is extended, it creates forms as shown in Figure 5. The structures in Figures

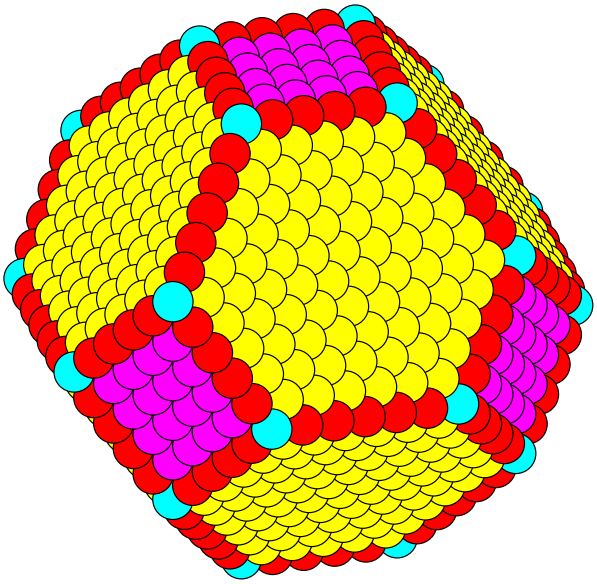


Figure 4: A truncated octahedron based on the FCC lattice

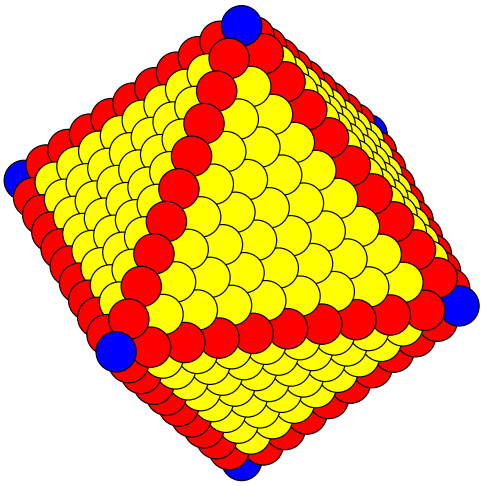


Figure 5: A 'diamond' shaped atom configuration

4 and 5 have been generated with the function `truncatedoctahedron()` shown below. This function takes into account the actual dimensions of the atom structure by assigning the material-dependent lattice constant a to the lattice coordinates.

```
/*
* int truncatedoctahedron(int L, double a, double x, double y, double z,
```

```
                          int *xx, int *yy, int *zz, int *c_ptr)
*
* Using the cubic starting configuration as the basis,
* this function cuts slices off from the cube to form the
* coordinates of a truncated octahedron.
* Please note that the use of this function should replace
* the function markatomsinsinderect();
*
* INPUT:
*
* 1. The total number of lattice sites L defined by
*    countlatticeatoms(); The value is fed through main();
* 2. The lattice constant a
*    defined as the constant value LCONST;
* 3. The size of the rectangular particle in x-direction
*    defined as the constant value XXX;
* 4. The size of the rectangular particle in y-direction
*    defined as the constant value YYY;
* 5. The size of the rectangular particle in z-direction
*    defined as the constant value ZZZ;
* 6. The pointer to the x-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[0]
* 7. The pointer to the y-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[1]
* 8. The pointer to the z-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[2]
* 9. The pointer to the list of occupied lattice sites allocated in
*    main(); associated pointer xyz[3]
*
* OUTPUT:
* The total number of populated atoms N in the lattice.
*
*/
int truncatedoctahedron(int L, double a, double x, double y, double z,
                        int *xx, int *yy, int *zz, int *c_ptr)
{
  int s, i = 0;
  double b, c, d;

  for (s = 0; s < L; s++) {
    b = a*(*(xx + s))/2;
    c = a*(*(yy + s))/2;
    d = a*(*(zz + s))/2;

    if ( ((b/(x/2)) <= 1) && ((b/(x/2)) >= -1)
      && ((c/(y/2)) <= 1) && ((c/(y/2)) >= -1)
      && ((d/(z/2)) <= 1) && ((d/(z/2)) >= -1)
      && ((((ABS(b) + ABS(c) + ABS(d))/(x/2)) - 0.5) < 1 ) ) {
      *c_ptr++ = 1;
      i++;
    }
    else
      *c_ptr++ = 0;
  }
  return i;
}
```

One might also be interested how the different colour schemes of the atoms have been accomplished. In simulations of several thousand atoms it is beneficial to update the locations of the atoms inside the configuration by changing only the number of the nearest neighbouring atoms. By generating a list that indicates the number of nearest neighbours of each atom, the colours can be chosen to represent the position of the atom in the configuration.

The code below can be used for generating the near-neighbour list. The function also takes into account the whole lattice, which means that there are lattice sites which have no atoms attached. This code was originally designed for simulations, where the atom configuration was allowed to move in the created 'lattice-space'.

```
/*
* void countcoordinationnumbers(int L, int *xx, int *yy, int *zz, int *c_ptr,
                                int *z_ptr, int *indexes_ptr, int type)
*
* This function fills two tables. Firstly the nearest neighbour
* coordination numbers of each lattice site is counted in a memory
* area with pointer int *z_ptr. Also a nearest neighbour list is
* generated in a memory area with pointer int *indexes_ptr. Both
* memory areas are allocated in main();. The coordination number
* list is updated during the simulation, but the neighbour list
* stays as it is through the whole simulation.
*
* INPUT:
*
* 1. The total number of lattice sites L defined by
*    countlatticeatoms(); The value is fed through main();
* 2. The pointer to the x-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[0]
* 3. The pointer to the y-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[1]
* 4. The pointer to the z-direction coordinate-table for the whole
*    simulation box. The memory area for the table is allocated in
*    main(); associated pointer xyz[2]
* 5. The pointer to the list of occupied lattice sites allocated
*    in main(); associated pointer xyz[3]
* 6. The pointer to the list of coordination numbers allocated in
*    main(); associated pointer xyz[4]
* 7. The pointer to the nearest neighbour list allocated in main();
* 8. The bulk coordination number of the used lattice structure
*    defined as a constant value TYPE;
*
* OUTPUT: (void)
*
*/
void countcoordinationnumbers(int L, int *xx, int *yy, int *zz, int *c_ptr,
                              int *z_ptr, int *indexes_ptr, int type)
{
  int j = 0, s, x, y, z, p, arg;
```

```
  int a, b, c;

  for (s = 0; s < L; s++) {
    j=0;
    for (z = -1; z <= 1; z++) {
      for (y = -1; y <= 1; y++) {
        for (x = -1; x <= 1; x++) {

          /* arguments for BCC and FCC lattices */
          if (type == 8)
            arg = (x && y && z);
          else
            arg = (x && y && !z)||(x && !y && z)||(!x && y && z);

          if (arg) {
            /*
             * The coordinates that are outside the simulation box
             * are marked with -1 to the nearest neighbour list.
             * This is relevant for many other functions.
             */
            *indexes_ptr++ = -1;
            a = x + (*(xx + s));
            b = y + (*(yy + s));
            c = z + (*(zz + s));

            for (p = 0; p < L; p++) {
              if ( (*(xx + p) == a) && (*(yy + p) == b) && (*(zz + p) == c) ) {
                *(indexes_ptr - 1) = p;

                if ( *(c_ptr + p) )
                  j++;
              }
            }
          }
        }
      }
    }
    *(z_ptr + s) = j;
  }
}
```

The four figures of the atom configurations have been plotted using the `fig` terminal in `gnuplot`. The coordinate-file with the near-neighbour numbering for the plot has been produced with the function `printformatlab()`. This function is shown below.

```
/*
* void printformatlab(int L, double a, int *xx, int *yy, int *zz, int *c_ptr, int *z_ptr)
*
* When called from main(); or some other function, this
* function prints a coordination file to be processed later
* with Matlab or some other plotting program such as gnuplot.
* The file name is enumerated and the number-part increases every
* time the function is called:
* 0000 --> 0001 --> 0002 --> 0003 --> i tak dalej....
```

```
*
* INPUT:
*
* 1. The total number of lattice sites.
* 2. The lattice constant a
*    defined as the constant value LCONST;
* 3. The pointer to the integer form x-direction coordinate-table for
*    the whole simulation box.
* 4. The pointer to the integer form y-direction coordinate-table for
*    the whole simulation box.
* 5. The pointer to the integer form z-direction coordinate-table for
*    the whole simulation box.
* 6. The pointer to the list of occupied lattice sites.
* 7. The pointer to the list of coordination numbers.
*
* OUTPUT: A text-file of the form "coord0000.txt".
*
*/
void printformatlab(int L, double a, int *xx, int *yy, int *zz,
    int *c_ptr, int *z_ptr)
{
  static int q = 0;
  int s, i = 0, j = 1;
  double x, y;
  char filename[] = "coord0000.txt";
  FILE *data;

  q++;
  for (s = 0; s < 4; s++) {
      j = j*10;
      i = (1.0*q)/j;
      x = (1.0*q)/(1.0*j);
      y = (x - i)*10;
      i = y + 0.000001;
      filename[8-s] += i;
    }

  if ( (data = fopen(filename,"w")) == NULL) {
    printf("\ncannot open file");
    exit(1);
  }

  for (s = 0; s < L; s++) {
    if ( (*(c_ptr+s) == 1) )
      fprintf(data,"%14.5f %14.5f %14.5f %4d\n",
              a*(*(xx+s))/2,a*(*(yy+s))/2,a*(*(zz+s))/2,*(z_ptr+s));
  }
  fclose(data);
}
```