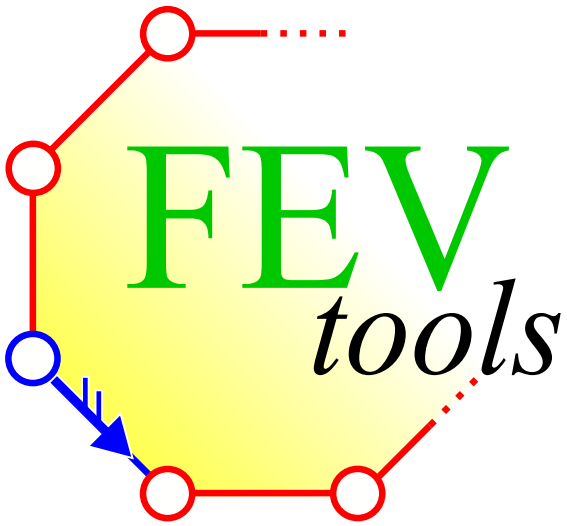
FEVtools

User Guide

Version 1.0

January 18, 2026



Contents

1 The Emedding Tensor and the Flag-Graph

The *Embedding Tensor* consists of a cuboid data structure Φ with elements $\Phi_{f,e,v}$, where the $f/e/v$ indexes label the faces/edges/vertices of the system. These elements are given by:

$$\Phi_{f,e,v} = \begin{cases} 1, & \text{if vertex } v \text{ is contained in edge } e \\ & \text{and edge } e \text{ is contained in face } f, \text{ and} \\ 0, & \text{otherwise.} \end{cases} \tag{1.1}$$

The labelling of the face/edge/vertex objects is illustrated for the structure in Fig. 1.1a, where we label the 6 vertices by black color numbers, the 9 edges by blue color integers, and the 3 faces by red color values. Note that we must obey periodic boundary conditions, so that the four octagons centered on the corners of the unit cell are actually the same face (face 3), as well as the two squares cut by the horizontal limits of the rectangular cell (face 2). This also applies to the edges, as edge 9 cuts both vertical limits of the unit cell, similarly to edges 7 and 8, which cut both horizontal boundaries. The embedding tensor for this structure is illustrated in Fig. 1.1b. Note, for instance, that $\Phi_{3,7,1} = 1$ (see green circle in Fig. 1.1b), as vertex 1 and edge 7 both make part of face 3, as well as edge 7 contains vertex 1.

Each nonzero element of the embedding tensor defines an entity called a *flag*. In other words, a flag is defined as a (f, e, v) integer trio for which $\Phi_{f,e,v} = 1$. When the involved edge is a bridge, each nonzero tensor element defines a pair of twin flags, which are basically two flags related by $(f', e', v') = (f, e, v)$. A flag can be represented graphically over the structure by an arrow that emerges from the corresponding vertex, along its corresponding edge, and has a mark (two small parallel segments) towards the corresponding cell. It also defines either a clockwise or a counterclockwise orientation around the face. In Fig. 1.1 we illustrate the $(1, 2, 3)flag$, where the blue color of the flag symbol and the blue counterclockwise-oriented circle indicate its orientation. In Fig. 1.1d we illustrate the counterclockwise-oriented $(1, 4, 4)$ in red color.

Each flag has neighboring flags, with which they share two elements in common. Each flag has a face-neighbor (same e and v), an edge-neighbor (same f and v), and a vertex-neighbor (same f and e). When we connect each flag to its neighbors, we have what we define as the flag-graph.

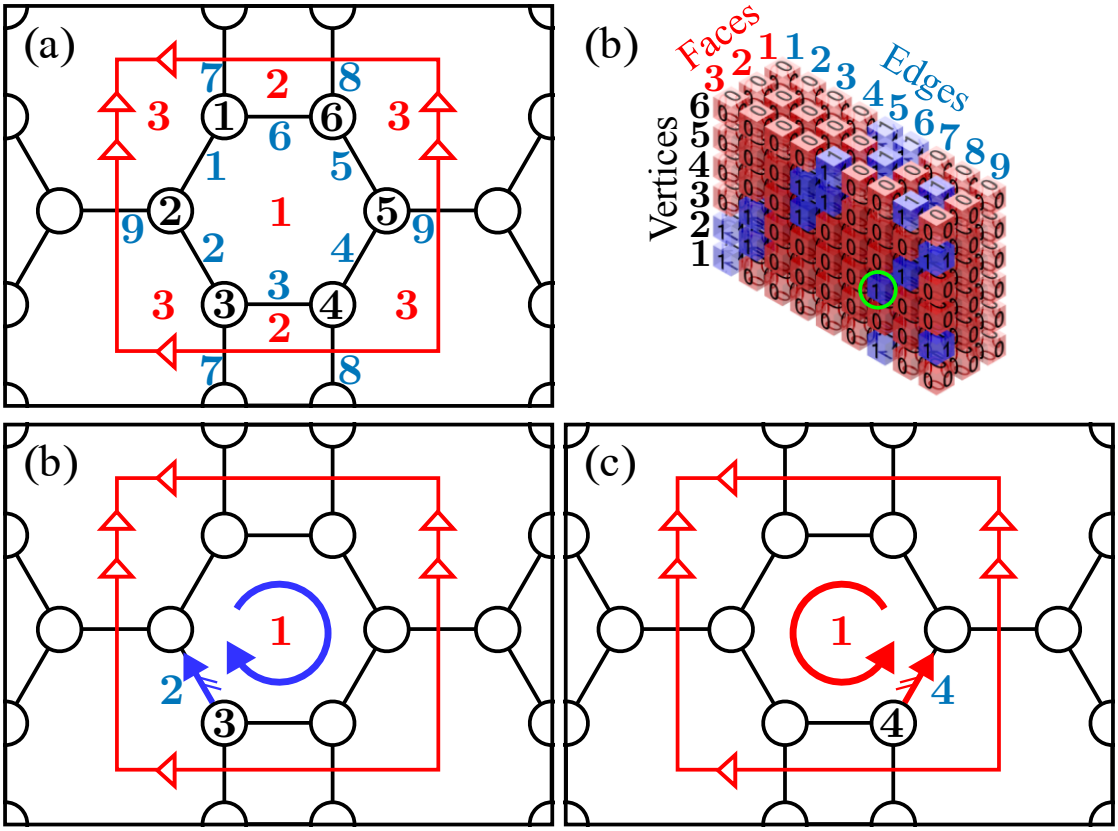


Figure 1.1: (a) Illustration of how faces (red numbers), edges (blue numbers), and vertices (black numbers) are labelled over a biphenylene periodic system. (b) Tensor cuboid data structure for biphenylene, where $\Phi_{f,e,v} = 1/\Phi_{f,e,v} = 0$ entries are represented by blue/red cubes. The green circle highlights the $\Phi_{3,7,1} = 1$ element. (c) Graphical representation of the $(1, 2, 3)$ flag, and its clockwise orientation. (d) Graphical representation of the $(1, 4, 4)$ flag, and its counterclockwise orientation.

2 File formats

The FEVTOOLS package deals with three fundamenta file formats (`.fev`, `.flg`, and `.fcs`), which are schematically illustrated in Fig. 2.1. Each carries different information related to a given structure. In the following, we detail each of them.

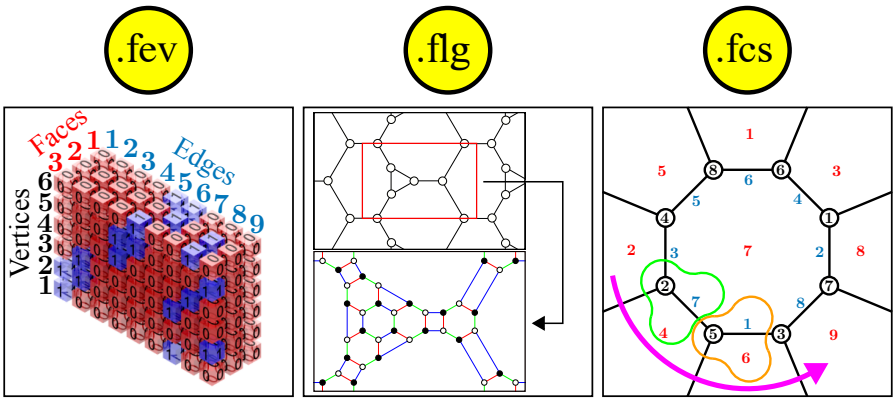


Figure 2.1: Schematicillustration of the `.fev`, `.flg`, and `.fcs` file formats.

2.1 The .fev format

In FEVTOOLS, the embedding tensor is exchanged with the user through a `.fev` file. This format is described below:

- A first line containing the three integers `nf`, `ne`, and `nv`, corresponding to the number of faces, edges, and vertices of the tensor, respectively;
- One line for each tensor entry (either 0 or 1), where the elements are ordered in a nested triple-loop, where the innermost (outermost) loop runs over the vertices (faces).

A simple FORTRAN code reads the tensor from a `.fev` file with `UNIT=1` as illustrated below:

```
READ(1,*) nf,ne,nv          ! .fev file format:
DO i=1,nf                   ! - 1st line: nf ne nv
  DO j=1,ne                 ! - Then one line for each tensor entry (0 or 1)
    DO l=1,nv               ! -> Outer loop --> Faces
```

```
      READ(1,*) fev(i,j,l)  ! -> Middle loop -> Edges
    END DO                  ! -> Inner loop --> Vertices
  END DO                    !
END DO
```

The `.fev` is an easy way for the user to input the studied structure. Other formats are more adequate to obtain further information, but they can be extracted from the `.fev` file from specific tools.

2.2 The .flg format

The flag-graph that can be extracted from a given embedding tensor, and consists of: 1) the list of flags, 2) the list of face-, edge-, and vertex-neighbors of each flag, and 3) the flags' orientations. In FEVTOOLS, the flag-graph of a system can be exchanged with the user through a `.flg` file. This format is described below:

- A first line containing the four integers `nflags`, `nf`, `ne`, and `nv`, corresponding to the number of flags, faces, edges, and vertices of the structure, respectively;
- A second line containing a vector with the number of edges from each face of the structure.
- In the following, we have one line per flag, containing 7 integers: 1-3) the f, e, and v elements of the flag, 4-6) its face-, edge-, and vertex neighbor flags, and 7) the flag orientation (either +1 or -1).

In FORTRAN it can be read as:

```
READ(1,*) nflags,nf,ne,nv
READ(1,*) nface
DO i=1,nflags
  READ(1,*) flag(i,:),neigh_flag(i,:),flag_color(i)
END DO
```

where `flag` and `neigh_flag` are `nflag`×3 matrices. An alternative `.b.flg` format is written/read in binary form in FORTRAN as

```
READ(1) nflags,nf,ne,nv
READ(1) nface
READ(1) flag
READ(1) neigh_flag
READ(1) flag_color
```

The `.flg` format is convenient to find symmetry maps that enable us to determine the symmetry group of a given structure or to determine if two structures are equivalent or not.

2.3 The .fcs format

The faces-info format contains structural information relative to each face of the system. In FEVTOOLS, it can be exchanged with the user through a `.fcs` file. This format is described below:

- A first line containing the four integers `nf`, `ne`, `nv`, and `nmax`, corresponding to the number of faces, edges, vertices of the structure, and the maximal number of edges in a face, respectively;
- Then we have blocks of 6 lines, one for each face of the structure. Each block contains:
  1. The number of edges in the face and a logical (`T` or `F`) value indicating if the face is a representative face (`T`) or if it is symmetry related to a representative one (`F`);
  2. The faces touching the reference face;
  3. The edge forming the reference face;
  4. The vertices forming the reference face;
  5. Logical entries indicating if the edges forming the face are bridges or not;
  6. A set of logical values (one for each edge of the face) indicating if the edge is a representative face (`T`) or if it is symmetry related to a representative one (`F`) within the face.

The vertices forming the reference face `i` are stored in a `v_in_f(i,j)` vector, in which `j` runs from 1 to the face size `n`. The same applies to the edges forming face `i`, which are stored in the `e_in_f(i,j)` vector, and to the faces touching face `i`, stored in the `f_in_f(i,j)` vector, always with `j=1,...,n`. All the elements in `f_in_f`, `e_in_f`, and `v_in_f` vectors have to be ordered around the face. This is done in such a way that the `j`-th edge from face `i` starts in vertex `v_in_f(i,j)` and finishes in vertex `v_in_f(i,j+1)`, while the `j=n`-th edge (`n` the size of face `i`) starts in vertex `v_in_f(i,n)` and finishes in vertex `v_in_f(i,1)`. The `j`-th edge (`e_in_f(i,j)`) is also the frontier between faces `i` and `f_in_f(i,j)`.

For instance, look at the example from Fig. 2.2a, where the reference face is `i=7`. Its starting elements are `f_in_f(7,1)=4`, `e_in_f(7,1)=7`, and `v_in_f(7,1)=2` (elements within the green curve). In the following, we have `f_in_f(7,2)=6`, `e_in_f(7,2)=1`, and `v_in_f(7,2)=5` (elements within the orange path). Explicitly, we can write

```
v_in_f(7,1:8) = [2, 5, 3, 7, 1, 6, 8, 4]
e_in_f(7,1:8) = [7, 1, 8, 2, 4, 6, 5, 3]
f_in_f(7,1:8) = [4, 6, 9, 8, 3, 1, 5, 2]
```

In this example, no edges are bridges, so that

```
b_in_f(7,1:8) = [F, F, F, F, F, F, F, F]
```

Suppose only edges 4 and 7 are equivalent by symmetry, then `u_in_f(7,1)=T` (relative to edge `e=7`) and `u_in_f(7,5)=F` (relative to edge `e=4`), since edge 7 is the first, and edge 4 is the fifth to appear in the sequence around the face (`u_in_f(7,j)=T` for all the other `j` values different from 1 and 5, since there are no more equivalent edges). Explicitly: In this example, no edges are bridges, so that

```
u_in_f(7,1:8) = [T, T, T, T, F, T, T, T]
```

Let us now consider the example from Fig. 2.2b, which features a bridge. In this case, edge `e=2` appears twice around `f=7`, and we have

```
v_in_f(7,1:8) = [2, 5, 3, 2, 4, 6, 1, 4]
e_in_f(7,1:8) = [7, 1, 3, 2, 4, 6, 5, 2]
f_in_f(7,1:8) = [4, 6, 2, 7, 3, 1, 5, 7]
```

Since edge 2 is a bridge, `b_in_f(7,4)=T` and `b_in_f(7,8)=T`, while `b_in_f(7,i)=F` for the other values. As they are the same edge, `e_in_f(7,4)=e_in_f(7,8)=2` are equivalent so that `u_in_f(7,4)=T` and `u_in_f(7,8)=F` since it appears firstly on the 4-th position. Explicitly

```
b_in_f(7,1:8) = [F, F, F, T, F, F, F, T]
u_in_f(7,1:8) = [T, T, T, T, T, T, T, F]
```

In FORTRAN it can be read as:

```
READ(1,*) nf,ne,nv,nmax
DO i=1,nf
  READ(1,*) nface(i),uneq_face(i)
  READ(1,*) f_in_f(i,1:nface(i))
  READ(1,*) e_in_f(i,1:nface(i))
  READ(1,*) v_in_f(i,1:nface(i))
  READ(1,*) b_in_f(i,1:nface(i))
  READ(1,*) u_in_f(i,1:nface(i))
END DO
```

where `f_in_f` is a `nf`×`nmax` matrix, just as `e_in_f` and the others. An alternative `.b.fcs` format is written/read in binary form in FORTRAN as

```
READ(1) nf,ne,nv,nmax
DO i=1,nf
  READ(1) nface(i),uneq_face(i)
  READ(1) f_in_f(i,1:nface(i))
  READ(1) e_in_f(i,1:nface(i))
  READ(1) v_in_f(i,1:nface(i))
  READ(1) b_in_f(i,1:nface(i))
  READ(1) u_in_f(i,1:nface(i))
END DO
```

The `.fcs` format is convenient to conduct structure generation through an add dimer approach.

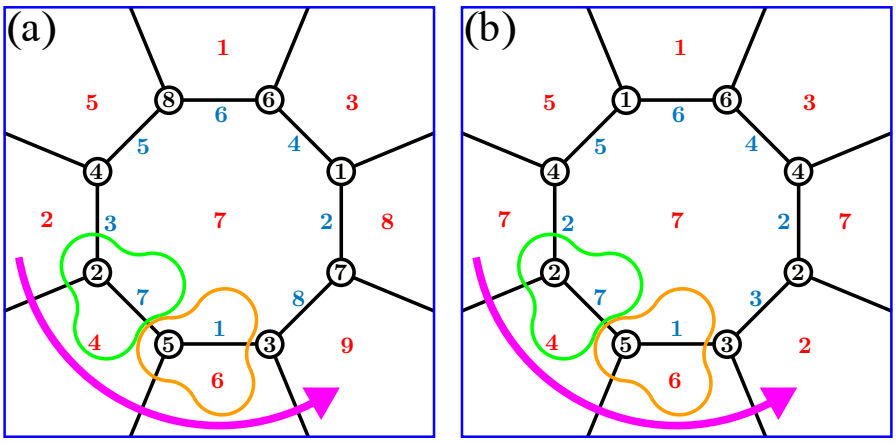


Figure 2.2: (a) Illustration of the sequence of neighboring faces, and composing edges and vertices of a given face (face 7). For the face `i=7` with size `n=8`, `v_in_f(7,1:8)` contains 2, 5, 3, 7, 1, 6, 8, and 4 in this order, while `e_in_f(7,1:8)` contains 7, 1, 8, 2, 4, 6, 5, and 3, as well as `f_in_f(7,1:8)` contains 4, 6, 9, 8, 3, 1, 5, and 2. The green curve highilights the `f_in_f(7,1)=4`, `e_in_f(7,1)=7`, and `v_in_f(7,1)=2` elements, while the orange curve highlights the `f_in_f(7,2)=6`, `e_in_f(7,2)=1`, and `v_in_f(7,2)=5` elements. (b) Same as (a), but an example with a bridge, where `v_in_f(7,1:8)` contains 2, 5, 3, 2, 4, 6, 1, and 4, `e_in_f(7,1:8)` contains 7, 1, 3, 2, 4, 6, 5, and 2, and `f_in_f(7,1:8)` contains 4, 6, 2, 7, 3, 1, 5, and 7.

3 Using FEVtools

FEVtools is a FORTRAN package of applications that includes a total of 15 tools:

- `fev2fcs`: tensor to faces-info conversion;
- `fev2flg`: tensor to flag-graph conversion;
- `fev2pov`: Tensor illustration generation in POVRAY;
- `fev2sum`: computes sum rules for a tensor;
- `fcs2fev`: faces-info to tensor conversion;
- `fcs2flg`: faces-info to flag-graph conversion;
- `fcs2kid`: structure generation;
- `fcs2smb`: polygonal symbol generation;
- `fcs2xyz`: coordinates generation;
- `flg2fcs`: flag-graph to faces-info conversion;
- `flg2fev`: flag-graph to tensor conversion;
- `flg2iso`: isomorphism checking;
- `flg2pri`: reduction to primitive cell;
- `flg2sym`: computes the symmetry group from a flag-graph;
- `for2bin`: Formatted to unformatted format conversion (and vice-versa).

A fluxogram detailing the tools and their purposes is illustrated in Fig. 3.1. We detail the use of each of them in the next sections.

To compile it, just edit the `makefile` indicating your FORTRAN compiler and type `make` to build all the 15 tools.

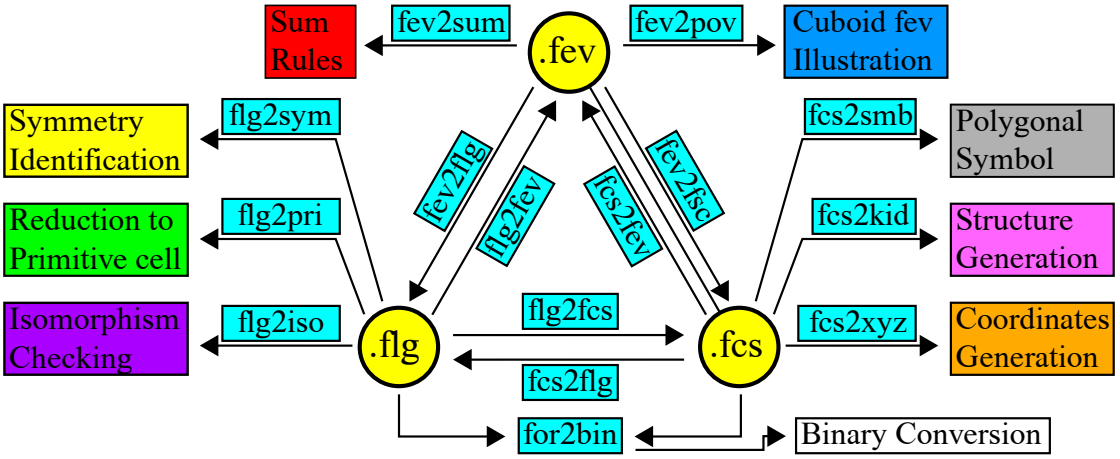


Figure 3.1: Summary of the utilities of the FEVTOOLS package.

3.1 Extracting the flag-graph from the embedding tensor

Use the `fev2flg` executable to create `.flg` and `.b.flg` files from the `.fev` file from a given structure. In addition to the `systemlabel.fev` file, you have to have the `fev2flg` executable and a `inp` file (containing only the `systemlabel` string in the first line) in the same folder. Just execute it with

```
./fev2flg
```

to make the conversion. If there are two or more non-equivalent flag-graphs associated with the tensor, it will generate `.flg` and `.b.flg` files for them all. In that case, the files for the first graph (structure) will be named as `systemlabel.flg` and `systemlabel.b.flg`, while the files for the second one will be named as `systemlabel_2.flg` and `systemlabel_2.b.flg`, and so on.

3.2 Extracting the embedding tensor from the flag-graph

Use the `flg2fev` executable to create `.fev` file from the `.flg` file from a given structure. In addition to the `systemlabel.flg` file, you have to have the `flg2fev` executable and a `inp` file (containing only the `systemlabel` string in the first line) in the same folder.

IMPORTANT NOTE More than being used for `.flg` to `.fev` conversion, it can be used as a tool to build the `.fev` file for a structure that you draw on a sheet of paper, for example. In your drawing, enumerate all the elements and create a "fake" `.flg` with the list of flags. For the `nface`, `neigh_flag`, and `flag_color` entries, write anything; they do not matter to build the embedding tensor. However, after generating the `.fev` from the drawing of your face `.flg`, you have to run `fev2flg` to create the corret `.flg` file.

3.3 Isomorphism checking

Use the `flg2iso` executable to verify if there is an isomorphism relation between two structures. You will need the `.flg` files for the two systems and the `inp` file has to have the `systemlabel` strings for both systems (one per line). Upon execution, the program will print a message saying

```
Isomorphic structures.
```

or

```
Non-isomorphic structures.
```

on the screen.

3.4 Extracting the faces-info data from the flag-graph

Use the `flg2fcs` executable to create `.fcs` and `.b.fcs` files from the `.flg` file from a given structure. In addition to the `systemlabel.flg` file, you have to have the `flg2fcs` executable and a `inp` file (containing only the `systemlabel` string in the first line) in the same folder.

3.5 Extracting the faces-info from the embedding tensor

The `fev2fcs` was created just for completeness. In fact, it is a `fev` to `flg` to `fcs` tool that has the same effect as executing `fev2flg` and then `flg2fcs` for each flag graph extracted for the embedding tensor. You use it just as you use `fev2flg`, but as a result, you will obtain the `.flg`, `.b.flg`, `.fcs`, and `.b.fcs` files for all the possible structures that can be extracted from the `.fev` file.

3.6 Extracting the flag-graph data from the faces-info

Use the `fcs2flg` executable to create `.flg` and `.b.flg` files from the `.fcs` file from a given structure. It is the opposite of `flg2fcs` In addition to the `systemlabel.fcs` file, you have to have the `fcs2flg` executable and a `inp` file (containing only the `systemlabel` string in the first line) in the same folder.

3.7 Formatted to unformatted file conversion

The `for2bin` executable checks if the `.flg` and `.b.flg` files for the structure indicated in the `inp` file exist. If only one of them exists, it creates the other. The same is done for the `.fcs` and `.b.fcs` files for the structure indicated in the `inp`.

3.8 Extracting the embedding tensor from the faces-info

Use the `fcs2fev` executable to create `.fev` file from the `.fcs` file from a given structure. In addition to the `systemlabel.fcs` file, you have to have the `fcs2fev` executable and a `inp` file (containing only the `systemlabel` string in the first line) in the same folder.

3.9 Creating an illustrative figure of the tensor

Use the `fev2pov` executable to create a `.pov` file for an illustration of the tensor in a perspective view, with 0/1 entries represented by red/blue cubes labelled with their values. It further illustrates the faces/edges/vertices axes in green, and a black background. In addition to the `systemlabel.fev` file, you have to have the `fev2pov` executable and a `inp` file (containing only the `systemlabel` string in the first line) in the same folder. Just execute it to create the `systemlabel.pov` file.

3.10 Computing sum rules for an embedding tensor

Use the `fev2sum` executable to compute the sum rules for an embedding tensor from a `systemlabel.fev` file and print the results on the screen. In addition to the `systemlabel.fev` file, you have to have the `fev2sum` executable and a `inp` file (containing only the `systemlabel` string in the first line) in the same folder. You simply type

```
./fev2sum
```

and it does the job.

3.11 Identifying the symmetry group from the flag-graph

Use the `flg2sym` executable to identify the symmetry group (printed on the screen). You have to have the `systemlabel.flg` file, the `flg2sym` executable, and the `inp` file (containing only the `systemlabel` string in the first line) in the same folder. The `flg2sym` tool also produces:

- A `.map` file with the flag-symmetry maps;
- A `.rot` file with structural elements over which we have a symmetry axis, and its order;
- A `.mir` file with the structural elements unchanged under mirror reflections, and their corresponding maps (a single element can be part of more than one mirror map);
- A `.sym` file with a line containing the symmetry group (also printed on the screen).

3.12 Polygonal symbol generation

Use the `fcs2smb` executable to create the polygonal symbol for the studied structure. You have to have the `systemlabel.fcs` and `systemlabel.sym` files, the `fcs2smb` executable and the `inp` file (containing only the `systemlabel` string in the first line) in the same folder. The `fcs2smb` tool produces:

- A `.smb` file with 3 versions of the symbol in latex commands;
- A pdf file for each of the 3 versions of the symbol.

The first symbol version is the simplest, most synthetic one. The second includes subscripts indicating the neighboring faces of each face, but grouped when two faces of the same size have the same set of neighbors. The third is a fully extended one, with subscripts indicating the neighboring faces of each face (not grouping faces with the same set of neighbors).

3.13 Reduction to primitive cell

Suppose you have a `.fev`, `.flg`, or `.fcs` file for a structure that does not represent a primitive cell. You can construct the `.fev` file for the corresponding primitive cell with `flg2pri`. You have to have the `.flg` file of your structure to make the reduction. In addition to the `systemlabel.flg` file, you have to have the `flg2pri` executable and a `inp` file (containing only the `systemlabel` string in the first line) in the same folder. Just execute `flg2pri`.

3.14 Structure generation

The `fcs2kid` tool creates a generation of structures with $N + 1$ faces from a parent generation of structures with N faces by means of the add dimer algorithm. To run it, you have to have

- A `gensizeN` file (substitute `N` in the filename by the number of faces) with the number of structures in the database of structures with N faces;
- A `gensystemsN` file (substitute `N` in the filename by the number of faces) with the systemlabel strings (one per line) for the `systemlabel.fcs` files from the database of structures of structures with N faces. In each line, after the system label, we also have three integers, indicating 1) the number of its ancestor in the previous database, 2) the face index in which the dimer was added to generate the present structure, and 3) the number of translation maps (should be 0 for primitive cell files). Note: add 1 1 0 to the graphene system in the `N=1` generation file;
- An `igen` file containing the N value of the parent generation;
- The `.fcs` files for all the structures from the database.

Once you have the correct files and the fev2kid executable, you just run it with

```
./fcs2kid
```

to generate the child generation of structures. It will generate the `.b.flg` and `.b.fcs` files for the new generation.

If you input 0 in `igen`, it will automatically generate the graphene files (labelled as `6`), together with the `gensize1` and `gensystems1` files.

3.15 Coordinates generation

During structure generation with the `fcs2kid` tool, a `.anc` file is also written for each generated system. It contains 1) the system label of the ancestor system, 2) the ancestor's face number in which the dimer is added, and 3) the indices for the two edges within the face in which we add the dimer atoms.

Suppose we want to generate a rough initial set of coordinates for a given target structure. You can do it with the `fcs2xyz` tool. To use it, you will need the following files:

- An `inp` file with the system label for the target system;
- The `.b.fcs` and `.anc` files for the target system;
- The `.b.fcs`, `.xyz`, and `.nxy` files for the target's ancestor.

Then you just need to execute

```
./fcs2xyz
```

to do it.

The generated `.xyz` file (and the used ancestor's one) contains x and y coordinates between 0 and 1 (fractional coordinates), while $z = 0$ for all the atoms. It implicitly considers a pair of lattice vectors

$$\mathbf{a}_1 = (1, 0) \quad \text{and} \quad \mathbf{a}_2 = (0, 1). \tag{3.1}$$

The `.nxy` contains blocks of 3 lines, one for each face of the structure. Each block contains:

1. The number of edges in the face (`i` labels the face);
2. The `x_in_f(i,j)` values, with `j` ranging from 1 to the number of edges in the face;
3. The `y_in_f(i,j)` values, with `j` ranging from 1 to the number of edges in the face.

To understand the meaning of `x_in_f(i,j)` and `y_in_f(i,j)`, suppose the `j`-th vertex of face `i` (`v_in_f(i,j)`) is located by the $\mathbf{r}_a$ vector, while the following vertex within the same face (according to the `v_in_f` data) is located by the $\mathbf{r}_b$ vector (both vectors

with coordinates within 0 to 1). However, if we eventually cross the $\mathbf{a}_1/\mathbf{a}_2$ cell when moving from the `v_in_f(i,j)` vertex to the next one within the same (`i`) face, then we move from $\mathbf{r}_a$ to $\mathbf{r}_b + x\mathbf{a}_1 + y\mathbf{a}_2$, with x being given by `x_in_f(i,j)`, and y given by `y_in_f(i,j)`.

The `fcs2xyz` tool takes the ancestor's coordinates, adds an atom in the center of each ot the edges defining the added dimer, and performs a very simple steepest descent geometry optimization along a set of steps. From the beginning, we know which are the neighbors of each atom, and this information does not change (non-reactive optimization). For each geometry step, we calculate the "force" F over each atom, and promote a small displacement given by the force multiplied by a "timestep" dt. The displacement is limited by a tenth of the average distance d_0 between neighbors in the system (updated in each geometry step). Such an optimization considers three kinds of forces.

1 - *Elastic bond stretching forces:* a Hooke's law like force given by

$$\mathbf{F}_i = -k\frac{(d-d_0)}{d_0}\mathbf{u}_{ij}, \tag{3.2}$$

where

$$\mathbf{u}_{ij} = \frac{\mathbf{r}_i - \mathbf{r}_j}{d} \tag{3.3}$$

is the unit vector pointing from $\mathbf{r}_j$ to $\mathbf{r}_i$, $\mathbf{r}_i$ is the position of the atom feeling the force, $\mathbf{r}_j$ is its neighbor position, $d = |\mathbf{r}_i - \mathbf{r}_j|$ is the distance between them, d_0 is the average neighbor distance in the system, and k is the force constant.

2 - *Elastic bond angle forces:* a Hooke's law-like force for bond angle variation. Consider the atom i located by $\mathbf{r}_i$ and its neighbors j and l, located by $\mathbf{r}_j$ and $\mathbf{r}_l$, respectively. Suppose $\mathbf{v}_j$ and $\mathbf{v}_l$ are the unit vectors orthogonal to $\mathbf{r}_j - \mathbf{r}_i$ and $\mathbf{r}_l - \mathbf{r}_i$, respectively, both pointing outside the $\widehat{jil} = \theta$ angle. These vectors are represented in Fig. 3.2.

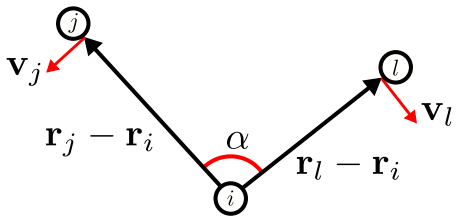


Figure 3.2: Illustration of the position $\mathbf{r}_i$ of the atom i the positions $\mathbf{r}_j$ and $\mathbf{r}_l$ of its neighbors j and l, as well as the $\widehat{jil} = \theta$ angle and the unit vectors $\mathbf{v}_j$ and $\mathbf{v}_l$ (orthogonal to $\mathbf{r}_j - \mathbf{r}_i$ and $\mathbf{r}_l - \mathbf{r}_i$, respectively) pointing outside the $\widehat{jil} = \theta$ angle.

The force on atom j will be given by

$$\mathbf{F}_j = -k_a\frac{(\theta-\theta_0)}{\theta_0}\mathbf{v}_j, \tag{3.4}$$

where $\theta_0 = 120^\circ$ is the reference sp^2 bond angle for graphene and k_a the force constant. The force on atom l will be given by

$$\mathbf{F}_l = -k_a \frac{(\theta - \theta_0)}{\theta_0} \mathbf{v}_l, \tag{3.5}$$

and the force over atom i will be given by

$$\mathbf{F}_i = -\mathbf{F}_j - \mathbf{F}_l. \tag{3.6}$$

3 - *Coulomb repulsive forces:* a Coulomb like force given by

$$\mathbf{F}_i = \frac{q}{d/d_0} \mathbf{u}_{ij}, \tag{3.7}$$

where q is the "charge" force constant, and $\mathbf{F}_i$ the force over atom i. The atom j will feel the force $\mathbf{F}_j$ given by

$$\mathbf{F}_j = \frac{q}{d/d_0} \mathbf{u}_{ji} = \mathbf{F}_i, \tag{3.8}$$

as $\mathbf{u}_{ji} = -\mathbf{u}_{ij}$.

The important quantities for the structure optimization are k, k_a, q, and dt. These can be determined by the user through the `relax.par` file, which contains these quantities (one per line) in this sequence. The `relax.par` file still has a fifth line containing an integer that tells if the program will draw a `.svg` and a `.pdf` file with an illustration of the atomic structure (0 for no, any other integer for yes). Below we show an example for this set of parameters in the `relax.par` format.

```
1.0D0   # k
8.0D0   # ka
1.0D0   # q
0.001   # dt
10000   # nstepmax
1       # makesvgpdf
```

Depending on your system, you may have to tune these parameters to find reasonable results.

If you want to generate the coordinates for a given system, you have to have its `.anc` file and the `.xyz`, `.nxy`, and `.anc` files for its ancestor. If you do not have the coordinates for the ancestor, the program looks for the ancestor's ancestor, and so on, down to graphene (which is the only system with two atoms). This works if you have run `fcs2kid` and have the files for the previous generations of structures in the working directory. Then, when it finds an ancestor of the tree with already generated `.xyz` and `.nxy` files, it generates the coordinates through the following generations until the target system. Even if you do not have the `.xyz` and `.nxy` files for graphene, the program will create them in the last scenario. Even though creating these files for graphene is not a hard task, with a little thinking and maybe a drawing on a piece of paper. Below we show a possible form for graphene's `.xyz` and `.nxy` files.
`graphene.xyz`:

```
2

C  0.37  0.37  0.00
C  0.67  0.67  0.00
```

`graphene.nxy`:

```
6
0  0 -1  0  0  1
0  1  0  0 -1  0
```

It is worth remembering that the `.fev` and `.nxy` have very simple forms, as represented below.
`graphene.fev`:

```
1    3    2
1
1
1
1
1
1
```

`graphene.fcs`:

```
1    3    2    6
6  T
1      1      1      1      1      1
1      2      3      1      2      3
1      2      1      2      1      2
T      T      T      T      T      T
T      F      F      F      F      F
```

If the target system listed on the `inp` file is labelled as `graphene` or simply `6`, the `fcs2xyz` tool will produce sample `.fev`, `.flg`, `.fcs`, `.b.fcs`, `.nxy`, and `.xyz` files for the graphene structure, as well as a `.svg` and a `.pdf` with the corresponding structure drawing.

Below, we reproduce the pdf images generated for structures with up to 3 faces.

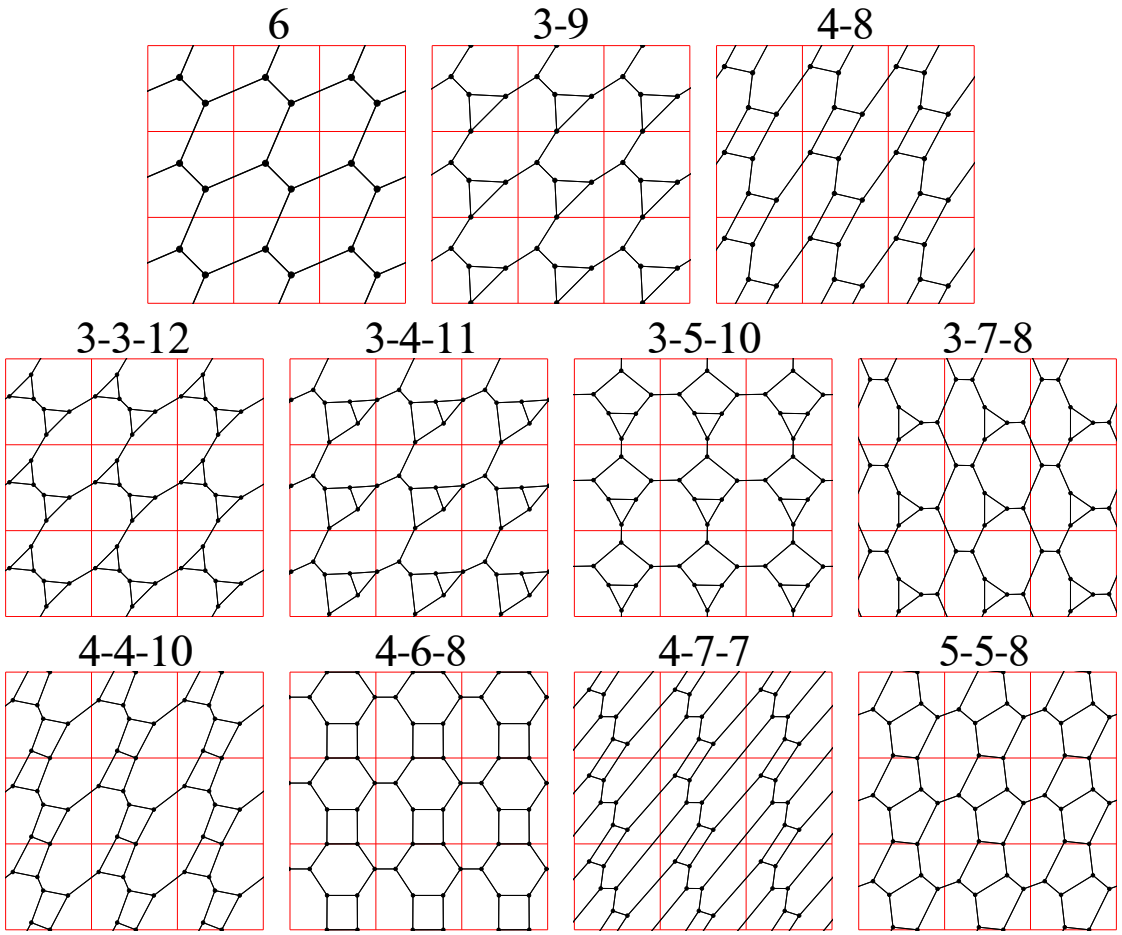


Figure 3.3: Output pdf images for the coordinates generated for the systems with up to 3 faces.