Polynator – A Tool for Local Geometry Analysis

Contents

1 Introduction

Polynator is a computer program written in the Python language. It allows the user to evaluate coordination environments and other shapes found in crystal structures by fitting model polyhedra to them. This documentation is intended to help the user navigate the features Polynator provides. It also gives an overview of the code structure and the mathematical methods fueling Polynator.

2 Getting Started

The program is available as a free download from *https://www.iac.uni-stuttgart.de/forschung/akniewa/downloads/*. This website currently gives the options of downloading the Python script or a .zip folder containing a Windows executable file. The Python version provides full access to the source code. It requires Python 3.9 or a newer version of Python 3 (slightly older versions might be

fine too). All imported modules are part of the Python standard library. The executable version requires Windows 10 or a newer version of Microsoft Windows. To get started with the latter, unpack the .zip folder wherever you would store applications on your computer. Then start Polynator by executing polynator_main.exe. The folder extracted from the .zip file is self-contained, it doesn't create configuration files anywhere else on your computer. Therefore, you can uninstall Polynator by simply deleting the folder. For convenience, you may want to create a shortcut to polynator_main.exe somewhere easy to reach on your computer. Polynator comes with a graphical user interface (GUI). The main window is created when starting the program. With the .exe version in particular, this may take a few seconds. If the window doesn't appear in your tab bar after 10-15 seconds, check if you meet the system requirements and that you haven't made any changes to the folder.

3 Using Polynator - Navigating the GUI

3.1 Fitting Polyhedra – The Main Window

To start working with the program, click on 'load input files' on the bottom left corner of the main window and locate any number of .cif files on your computer. You may then click on 'Run'. In this case, Polynator will construct a coordination environment around every unique atom it finds in the .cif files using default cutoffs and then sequentially fit each built-in model polyhedron to each of these environments. This is most likely not what you want to happen. You have several tools at your disposal to shape Polynators behaviour:

- The criterion panels at the top of the window allow you to filter out central atoms, ligand atoms and model polyhedra, respectively. To do this, first select a logic operater (and / or / not), then type in or select from the dropdown menu any valid entry for that box. Press Enter to confirm your entry. It will appear in the box. The polyhedron criteria panel will also accept fragments of valid entries, which will often be less selective, but the atom criterion panels do not (otherwise 'N' would also find Nb, Ni, Zn etc.). The atom criteria panel includes wildcards like '*M' for all metals (see tab. 3). The atom criterion panels also have some advanced functionality in allowing you to add dummy atoms and setting a maximum for the number of (non-rotational) degrees of freedom per polyhedron. Central dummy atoms

are especially useful, as they allow you to inspect any shape in your crystal structure that has no atoms at its center. To add one, simply type in a string of three coordinate numbers between 0 and 1, separated by commas and/or spaces, then press Enter.

- The coordination environment filters allow you to define minimal and maximal global values for the distances of ligand atoms from central atoms. You may also find it useful to specify a maximal value for the coordination number of central atoms; atoms in excess of this number will be dropped until $\mathrm{CN}_{\mathrm{max}}$ is reached, with the most distant atoms going first. As an alternative to these caps, Polynator can automatically carve out coordination environments for you. This feature is purely for convenience and isn't based on a very scientific method or intelligent algorithm. It basically looks for atoms with a similar distance from the central atom and jams them into a coordination sphere, starting a new one whenever a large jump occurs or the distance range within a coordination sphere grows too large. Use it by entering one or more small integers into the field labelled 'coord. sphere:'. If you do, d_{min}, d_{max} and $\mathrm{CN}_{\mathrm{max}}$, as well as atomic radii will be ignored.

- Atomic radii allow you to be more selective when it comes to manually defining a coordination environment. If atomic radii are defined for the central and / or the ligand atoms and the sum of both is smaller than d_{max}, it replaces d_{max} for this combination of atoms. To use atomic radii, first select one or more entries you made in either central atom or ligand atom criterion box, then enter a number into the respective 'set radius:' field and press enter.

The 'fit settings:' panel gives the user the choice to include the central atom in the list of atoms Polynator will include in the list of atoms when measuring the csm. It is measured against the centroid of the model vertices and if 'best fit' is also selected below, it will factor into the centering of the model polyhedron.

The user has the option of manually entering a main axis (see chapter 4, fig. 2). This bypasses the automatic belt assignment step and forces Polynator to assign belts according to this axis. This may be useful if the user suspects a wrong belt assignment. An axis is entered in the form of fractional coordinates, separated by commas or spaces. If the 'don't adjust' box is ticked, this axis will be static throughout the whole fitting process.

3.2 Output Files

Upon a successful run, output files can be generated via 'Generate Output'. They will be put into a subdirectory of the folder holding the .cif files that were evaluated. There are five types of output files, the generation of which can be toggled in the settings menu.

Detailed output files (.out) are purely text-based. One .out file is generated for each fit. The information in an .out file is split up into blocks.
The block of information at the top should mostly be self-explanatory. The excentricity vector is the distance between the real central atom and the model center.
The next block gives vertex-specific coordinates and directional distortion information. All coordinates given are fractional (the same format you would find in a .cif file). The delta vector gives the difference between the model and real ligand vector. Its length is given in the next line. The entry 'angular delta' refers to the angle a given ligand vector is removed from its model counterpart, from the perspective of the model center. 'Radial delta' means the difference between the distances of that atom vector and its model counterpart from the model center. 'Delta phi' and 'delta theta' split the 'angular delta' into the respective spherical coordinate representations.
The third block holds some statistical information. This includes averages for the delta length, radial delta and angular delta measures from the previous block. The standard deviation given for the delta length measure should not be confused with a quantification of measurement errors; Polynator's statistical errors are negligible (many tests suggest the same is true for systematic errors). In addition, this block gives the *csm* value for a central projection of all real and model vertices onto a unit sphere and for a cylindrical projection onto the main axis.
The model constructor gives a compact, Python-dictionary-like representation of the model polyhedron belts and their parameters. If you have trouble understanding this part, try playing around with the custom polyhedron window or see chapters 4 and 5.
Lastly, there are the parameter values broken down into free and, for some polyhedra, constrained ('bound') parameters.

The minimal .cif files Polynator creates are intended mostly for visualiza-

tion and perhaps verification in an external program. They have space group $P1$ regardless of the original space group and hold only the real and model vertices involved in the respective fit. Both are true to the size and proportions of the original coordination environment, but only the fractional version is also true to the original unit cell. However, that version may also have unconvenient placement and overlaps between translated coordination environments, which are not present in the cartesian version.

The data table is generated as a single .csv file which contains the most important information about all fits in the last run combined. This includes the *csm*, the averaged 'delta length', 'radial delta' and 'angular delta' values discussed above and the free and constrained parameters for each fit.

The .log file tries to record if something went wrong during the calculations. If you observe unexpected behaviour, it might be worthwile to look at this file, otherwise don't worry about it.

3.3 The Settings Menu

The settings menu allows the user to customize the set of model polyhedra Polynator is actively using without having to specify your preferences in the 'polyhedron criteria' box of the main window every time. For example, if you don't care about 'exotic' model polyhedra, you may find it useful to enter and confirm the inclusion criterion *#essential* (see the Polynator's tags in tabs. 1 and 2). The menu also allows you to customize the output behaviour. Lastly, there are three fields holding numbers which allow you to change some of Polynators behaviour when fitting Polyhedra.

3.4 The Custom Polyhedron Construction Menu

The custom polyhedron construction window is accessible from the main window via 'use as blueprint' (select a polyhedron in the box above to use it as a starting point) or from the settings menu. It allows the user to construct their own model polyhedra. The central component of this is worked out in the 'belts:' panel, where the user can add or remove belts (see fig. 2) with a number of vertices

specified in the 'v:' field. There is also a number of optional parameters the user can apply to each belt. These are probably best understood by selecting various model polyhedra in the 'model polyhedra' box at the bottom of the main window and observing them in the 'viewer:' panel. Click at the boxes at the bottom of that panel to toggle the effects of each parameter separately. Alternatively, it is highly recommended to just play around and create your own polyhedra. The comment panel at the bottom will guide you.

The new model polyhedron also needs a name. Additionally, you have the option of assigning a point group, a symmetry operation pertaining to the main axis, a list of 'aristohedra', i. e. preexisting polyhedra with higher symmetry or fewer degrees of freedom, as well as a list of search terms or categories for the new polyhedron. The point group has no bearing on the functionality other than being displayed at various points and allowing the user to search for it. The axis symmetry allows Polynator to determine whether a rotoinversion center or horizontal mirror plane is present in main axis direction. This changes the behaviour when the atom vertices are assigned to belts. The aristohedron category affects the tree-like structure displayed in the main window's results box after fitting. It also precludes the model polyhedron from being evaluated if a polyhedron higher up the chain is already a perfect fit. Custom polyhedra can currently not inherit their belt assignments from their aristohedra. Finally, the 'tags' category mostly provides search terms, but there are three built-in tags that actually change Polynators behaviour in another way (see tab. **??**).

Lastly, you may add constraints ('bundles'). This is not necessarily a simple task. Some guidelines will be given here, but it is recommended to look at existing polyhedra with bundled parameters (They all share the *#bundled* tag). There are two types of bundled polyhedra:

- The first type is based on a rigid base polyhedron, which is modified by the bundled parameters. it has a 'len' parameter that is **not** part of the bundle. The bundled parameters may change its shape, but must not change the average distance of the vertices from the center (otherwise calulations will be inaccurate, potentially in a subtle way). The parameters for this type of polyhedron must thus work together to manipulate the angular components of the spherical vertex coordinates. This type is best suited for polyhedra

with a cubic point group. Examples include the >pyritohedral_icosahedron and >tetrahedral_cuboctahedron.

- The second type is usually exclusively handled by bundled parameters. Whenever a variable is adjusted, each parameter that depends on it must also be adjusted, as calculations would otherwise be inaccurate. This makes these the most computationally expensive polyhedra in Polynator. There is currently only one built-in example; the >ax-truncated_hexagonal_trapezohedron, which requires the bundle to keep its pentagonal faces from folding up.

Table 1: Tags attached to model polyhedra. Note: The tags *#molecule*, *#axis_fixed* and *#chiral* directly affect Polynators behaviour, as explained here, while the others only serve as categories and search terms. Most tags refer to the shape of a polyhedron and should be self-explanatory. Those are listed in list 1.

tag	explanation
#molecule	The polyhedron is centered on its central atom.
#axis_fixed	The main axis determined in the belt assignment step is never changed during the optimization steps. Orientation is only optimized with regards to rotation around this axis.
#chiral	Rigid chiral polyhedra, such as the archimedean snub cube, are tested for either enantiomer, independently from this tag. This tag adds the suffixes '_clockwise' or '_anticlockwise' to the polyhedron name in the results list and the output files, depending on which enantiomer fits better. This tag should not be applied to every polyhedron with chiral space groups such as twisted prisms, whose parameters actually allow the manifestation of either enantiomer without a separate test run.
#rigid	The polyhedron is only scaled with fixed proportions, never deformed.
#free	The polyhedron has all degrees of freedom allowed by its point group (this includes some rigid polyhedra with cubic point groups).
#limited	The deformation of the polyhedron is restricted by non-symmetry based constraints (the opposite of *#free*). This includes all Johnson and Catalan polyhedra, many archimedean polyhedra and some others.
#essential	Subjective category comprising only the polyhedra most commonly encountered in inorganic chemistry.
#bundled	All polyhedra with at least one parameter bundle (see section 4). While this amounts to constrained parameters, it is not to be confused with non-symmetrical shape constraints as captured by the *#constrained* tag.
#equilateral	All vertices have the same distance from neighboring vertices they share an edge with.
#equidistant	All vertices have the same distance from the center

Table 2: Tags referring to the polyhedron shape.

#antifrustum	*#capped_octahedron*	*#gyrobicupola*
#antiprism	*#capped_prism*	*#gyroprism*
#archimedean	*#capped_tetrahedron*	*#heterobipyramid*
#axis-bicapped_antiprism	*#capped_truncated_cube*	*#johnson*
#axis-bicapped_prism	*#catalan*	*#kinked*
#axis-capped_antifrustum	*#chamfered_cube*	*#linear*
#axis-capped_antiprism	*#cupola*	*#orthobicupola*
#axis-capped_frustum	*#deltahedron*	*#planar*
#axis-capped_prism	*#diminished_icosahedron*	*#platonic*
#axis_truncated	*#distorted_cuboctahedron*	*#prism*
#bidisphenoid	*#edshammar*	*#pyramid*
#bipyramid	*#elongated_bicupola*	*#pyritohedron*
#capped_antiprism	*#equator-capped_prism*	*#regular*
#capped_biprism	*#frank-kasper*	*#scalenohedron*
#capped_cube	*#frustum*	*#skewed_prism*
#capped_cuboctahedron	*#fullerene*	*#twisted_prism*
#capped_cupola	*#fully_capped_prism*	

Table 3: Wildcards for groups of elements in the atom criteria panels (not case sensitive).

wildcard	corresponding elements
*	all elements
*Gr**n**	all elements in periodic table group **n**
*M	all metals
*TM	all transition metals (except rare earth metals)
*RE	all rare earth elements including lanthanoids, actinoids, Sc and Y.
*E	all main group elements
*Ln	lanthanoids including La and Lu
*An	actinoids including Ac and Lr
*X	typical anions (N, P, O, S, Se, F, Cl, Br and I)

4 Mathematical Aspects

4.1 Overview

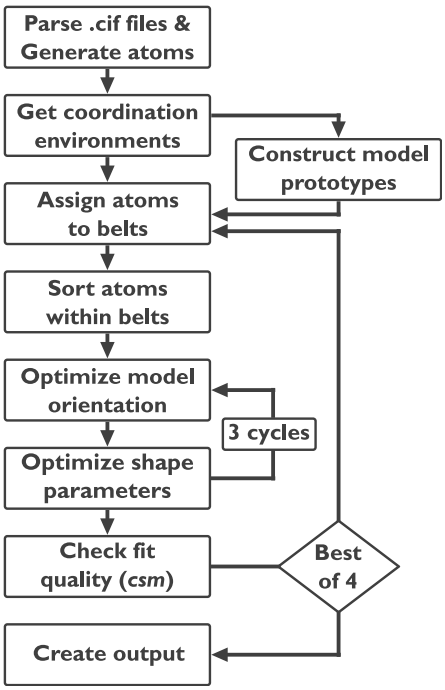

Figure 1: Flow chart of Polynators working process.

Polynator seeks to minimize the *continuous symmetry measure* (*csm*) [1] , which is a least squares based metric for the dissimilarity between a set of pairs of real and model vertices $\vec{a}_i$ and $\vec{v}_i$ defined as

$$csm = 100 \cdot \frac{\sum_i |\vec{a}_i - \vec{v}_i|^2}{\sum_i |\vec{a}_i - \vec{c}|^2} \,. \tag{1}$$

where $\vec{c}$ is the centroid of the real vertices.
To minimize the *csm*, Polynator internally has to solve four main problems:

1. Pairing up model vertices with real vertices in an optimal fashion.
2. Finding the optimal coordinates to center the polyhedron at.
3. Finding the optimal orientation of the model polyhedron. This has an analytical solution in the Kabsch Algorithm [2], which will not be discussed here.

4. Optimizing one or more parameters to obtain the ideal shape of the model polyhedron.

Table 4: Mathematical symbols presented in this chapter generally do not match their counterparts in the code or the GUI. Here is a translation table.

symbol	code name	location in the code
$\vec{a}_{ij}$	vec_real	Polyhedron.belts_real (a list of lists of vectors)
$\vec{v}_{ij}$	vec_model	Polyhedron.belts_model (a list of lists of vectors)
$\mathbf{M}$	var_matrix	covariance_eigen() function
$\hat{m}$	main_axis	Polyhedron.main_axis
S	HALF_SPHERE	assign_to_belts_from_scratch() function
q	tuple_quality	assign_to_belts_from_scratch() function
Q	total_delta	Polyhedron.match_real_vecs_to_model() method
P_{ij}	(various names)	Polyhedron.adjust...() methods
n	"v"	DICT_BLUEPRINTS, Polyhedron.belt_dicts
s	"len"	Polyhedron.belt_dicts, Polyhedron.dict_parameters
h	"z"	Polyhedron.belt_dicts, Polyhedron.dict_parameters
w	"xy"	Polyhedron.belt_dicts, Polyhedron.dict_parameters
φ	"phi"	Polyhedron.belt_dicts, Polyhedron.dict_parameters

4.2 Pairing Up Model and Real Vertices

This problem is in theory easily solved by just checking every permutation of vertex pairings. However, since the number of permutations grows factorially with the number of vertices and each 'checking' step comes with a significant computational cost, this approach has to be discarded. As far as we are aware, there is no substitute that is determined to yield the perfect solution for any distribution of atom vectors while allowing for a computationally cheap implementation. However, some observations about the typical shape of coordination environments can be exploited to formulate a heuristic that comes very close to this goal.

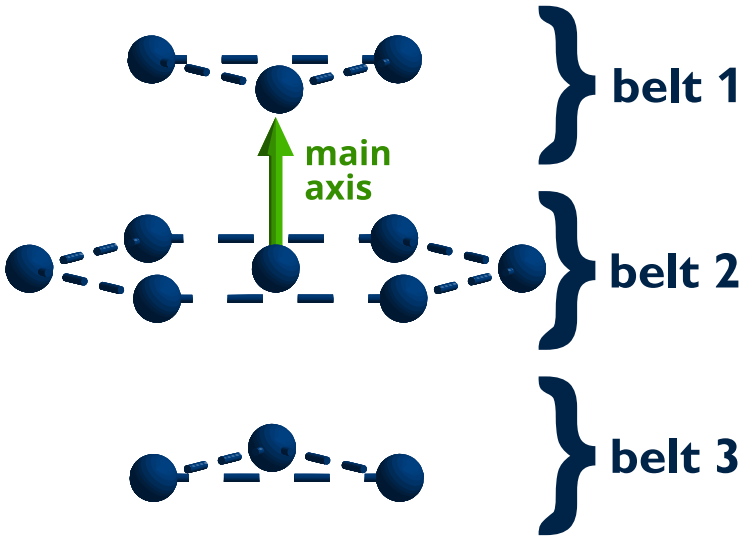

Figure 2: Separation of a cuboctahedron into belts.

The first step towards this is to separate the model polyhedron into subsections, which we will call belts. This is done along a main axis (see fig. 2), which is a high symmetry axis of the model polyhedron. The origin is set to either the central atom or the centroid of all ligand atoms (the difference is unimportant here). The atom vectors are then projected onto a unit sphere surface (central projection). To assign each atom vertex to a belt of the model polyhedron (not yet to a specific model vertex), a tentative main axis vector is chosen from among a predefined set S of 92 vectors, which are distributed almost evenly on a sphere surface (they correspond to the vertices of a fully capped truncated icosahedron). The atom vectors are ranked according to their dot product with this axis vector and the belts are filled up in this order. This process is repeated for each remaining axis vector in S. Model polyhedra with a rotoinversion center or horizontal mirror plane require only the 46 axis vectors corresponding to one hemisphere. Duplicate belt assignments are discarded. The remainder is ranked according to the heuristical quality measure q. To obtain it, the main axis for each provisional assignment is first refined as follows:

1. The centroid $\vec{c}_i$ of the atom vertices in each belt i is calculated and subtracted from each atom vector $\vec{a}_{ij}$ in the respective belts to obtain an auxiliary vector $\vec{b}$:

$$\vec{b}_{ij} = \vec{a}_{ij} - \vec{c}_i \,. \tag{2}$$

2. A covariance matrix $\mathbf{M}$ is constructed from the cartesian coordinates x, y

and z of the $\vec{b}_{ij}$ vectors:

$$\mathbf{M} = \begin{pmatrix} \sum_{ij} x_{ij}^2 & \sum_{ij} x_{ij} \cdot y_{ij} & \sum_{ij} x_{ij} \cdot z_{ij} \\ \sum_{ij} y_{ij} \cdot x_{ij} & \sum_{ij} y_{ij}^2 & \sum_{ij} y_{ij} \cdot z_{ij} \\ \sum_{ij} z_{ij} \cdot x_{ij} & \sum_{ij} z_{ij} \cdot y_{ij} & \sum_{ij} z_{ij}^2 \end{pmatrix} . \tag{3}$$

3. The unit eigenvector of $\mathbf{M}$ with the smallest eigenvalue is chosen as the main axis $\hat{m}$.

4. The quality measure q can now be calculated as

$$q = \sum_i |\vec{c}_i \times \hat{m}|^2 \cdot n_i + \sum_{ij} (\vec{a}_{ij} \cdot \hat{m})^2 \tag{4}$$

   where n_i is the number of vertices in belt i. The first term represents horizontal displacements of the belts, the second vertical displacements of the individual vertices within each belt.

The four belt assignments with the smallest q are evaluated further, the rest is discarded. To pair up ligand atoms and model vertices, the combinations that maintain the order of dihedral angles (up to one per ligand atom) are ranked, again, heuristically. To do this, the model polyhedron is first rotated around $\hat{m}$ so that atom vector $\vec{a}_{00}$ lies in the plane constructed by $\hat{m}$ and an arbitrarily picked model vertex $\vec{v}_{\text{tare}}$ ($\vec{v}_{\text{tare}}$ must not be collinear with $\hat{m}$). For each atom and model vector, the dihedral angle α in relation to $\vec{v}_0$ is measured ($\hat{m}$ serves as the hinge). Atom and model vectors are then (separately) ordered within the individual belts according to their dihedral angles and tentatively paired up in this order. The quality Q is measured as

$$Q = \sum_{ij} |\vec{v}_{ij} \times \hat{m}| \cdot |\Delta\alpha_{ij}| \tag{5}$$

where $\Delta\alpha_{ij}$ is the angular difference between the paired vectors with belt and vector numbers i and j. This is repeated until each atom vector $\vec{a}_{ij}$ has been in the $\alpha = 0$ position once. The pairing scheme with the smallest Q is evaluated further, the rest is discarded.

4.3 Centering the Model Polyhedron

Due to the nature of the least squares fit, the optimal center for the model polyhedron is always the centroid of all atoms that are fitted. Depending on the user input, this may or may not include the central atom. One property of the least squares fit is that the polyhedron centering is independent from orientation and shape of the model vertices, meaning it can in principle be set wherever and only adjusted after the other calculations are finished. However, Polynator centers the polyhedron on the centroid of all ligand atoms while other calculations are ongoing, unless the polyhedron characteristics or the user input require it to be placed on the central atom. For convenience, if the central atom is included in the list of atoms to be fitted, it is ignored at first and the centering is retrofitted at the end to include it.

4.4 Optimization of Shape Parameters

In the simplest case, the model polyhedron is a rigid body and the only parameter to be optimized is its size s. To do this, the length P of each each atom vector $\vec{a}_{ij}$ when projected onto its corresponding model vector $\vec{v}_{ij}$ is aquired as

$$P_{ij} = \vec{a}_{ij} \cdot \hat{v}_{ij} \,. \tag{6}$$

If all model vectors have the same length, the problem can be solved analytically by taking the arithmetic mean of P:

$$s_{\text{opt}} = \frac{\sum_{ij} P_{ij}}{\sum_{ij} 1} \,. \tag{7}$$

Otherwise it is solved by iteratively minimizing the squares of the differences between $|\vec{v}_{ij}|$ and P_{ij}. This is still very straightforward, as the corresponding function is always a parabola, so there are no problems with local minima or discontinuities. Sometimes, for example with the generic fully capped cube, there are two or more size parameters (one for the cube vertices, the other for the caps). These can just be solved separately with the same methods.
Many polyhedra, such as generic prisms, antiprisms, pyramids..., require the separate optimization of height and width parameters h and w (instead of a single s parameter, not in addition). Thanks to the pythagorean theorem, this is very easily done. To optimize h, instead of projecting the atom vectors onto the model

vectors, as was done before, they are projected onto the main axis $\hat{m}$:

$$P_{ij} = \vec{a}_{ij} \cdot \hat{m}\,. \tag{8}$$

Similarly, to optimize w, they are projected onto the normal plane of $\hat{m}$:

$$P_{ij} = \vec{a}_{ij} \cdot \hat{n}_{ij} \quad \text{where} \quad \vec{n}_{ij} = \vec{a}_{ij} - (\vec{a}_{ij} \cdot \hat{m}) \cdot \hat{m}\,. \tag{9}$$

From here, the exact same methods as for optimizing s are applied.
Parameters φ for twisting motions (e. g. for twisted prisms) are also straightforward, but computationally more expensive, as they require vector operations during iterations. During these iterations, the model vectors are incrementally rotated around $\hat{m}$. After each rotation step, the sum of the squared distances between atom and model vectors is measured. If it is satisfactorily minimized, the iteration is terminated. In the built-in polyhedra, φ is always balanced by a counterrotation $\varphi*$ in a different belt, which is optimized along with φ, in order to maintain as much as possible of the model orientation.
There are also modulating versions of h, w and φ which allow the modelling of normal mode vibrations of orders higher than 0. As an example, a square is modelled by a single belt containing four vertices, with a single w parameter. By adding h_{mod}, w_{mod} or φ_{mod}, the disphenoid, rhombus or rectangle, respectively, can be derived from it. Modulated parameters are not applied equally to all vertices in a belt, but rather with a prefactor generated by a sinusoidal function (cos for h_{mod} and w_{mod}, sin for φ_{mod}). The frequency f and offset δ of these functions are determined when a model polyhedron is defined and never changed or optimized. Hence, the value of modulating parameters corresponds solely to the amplitude of the respective sinusoidal function. The prefactor g for each vertex is obtained as a function of its position p in an n-membered belt (belts are sorted according to the dihedral angles of their member vertices around $\hat{m}$, enumeration starts at 0):

$$g(p) = \begin{cases} h_{\mathrm{mod}},\ w_{\mathrm{mod}} & \to \cos\left(\frac{2\,\pi \cdot f \cdot (p+\delta)}{n}\right) \\ \varphi_{\mathrm{mod}} & \to \sin\left(\frac{2\,\pi \cdot f \cdot (p+\delta)}{n}\right) \end{cases} \tag{10}$$

The divergent choice of basic trigonometric functions may seem odd, but in practice makes it easier to think about these, due to their transversal (h_{mod}, w_{mod}) and longitudinal (φ_{mod}) nature. Optimization of the modulating parameters

follows the same priciples as for their non-modulating counterparts.
Lastly, there is the option to bind multiple parameters to freely definable functions of one or more newly defined variables, effectively constraining them. This allows for model polyhedra such as the elpasolite cuboctahedron, the pyritohedron and pyritohedral icosahedron, as well as symmetry-preserving non-equilateral versions of archimedean solids and more. Optimization of these variables always involves an iterative process where each parameter bound to the variable is applied at each step and the progress is gauged as explained for the φ parameter.

After all shape parameters have been optimized, the orientation is adjusted again, taking into account the new shape of the polyhedron. Three repetitions of this cycle are generally sufficient to refine the *csm* to at least six decimal places, which is already more than most crystal structure data provide in the first place.

5 Navigating the Code

Polynator is written in a loosely object-oriented style. Other than the modules imported from the standard library, all of the code is contained in the polynator_main.py file. There are seven large classes, seven smaller classes and some regular functions. At the top, several global constants are defined, including the gargantuan DICT_BLUEPRINTS, which has coordination numbers as keys and lists of tuples as values. The tuples are referred to as blueprints. They store all the information needed to construct the model polyhedra. Each has three entries, which are referred to as name, dict_info and belt_dicts, respectively.

The main process is initiated at the very bottom of the script. It mostly instantiates a MainWindow object, which contstructs the main GUI window and orchestrates the further procedure. Upon calling the MainWindow.preview() or MainWindow.run_full() methods, it instantiates a CifFile object for each .cif file it was fed by the user, which reads the file and stores the information it contains. Depending on the crystal structure and other inputs, each CifFile may host any number of CoordinationEnvironment objects. Each CoordinationEnvironment contains a fixed set of atoms and is home to any number of Polyhedron objects, which are built from blueprints fitting the coordination number of the respective CoordinationEnvironment object. Each Polyhedron has a belts_real and a belts_model attribute, which store the vertices as a list of lists. The

main calculations are carried out within the Polyhedron objects. The actual optimization methods have names starting with adjust_ and are specific to the type of parameter being fitted.

Other central components include the prototype_polyhedron() function, which constructs the vertices of a blank model polyhedron, as well as the assign_belts() function, which wraps assign_belts_from_scratch() and recycle_belt_assignment(), which take care of most of problem 1 as discussed in chapter 4 (the vertices are finally paired by Polyhedron.match_real_vecs_to_model()). The recycle_belt_assignment() function implements 'piggybacking', i. e. it allows a polyhedron to inherit its belt assignment from one of its aristohedra under certain conditions.

The modulating parameters described in section 4 (which are named differently in the code, see tab. 4) go along with modifiers '_frq' and '_off' for the frequency and offset of the sinusoidal wave function, respectively. Parameter types 'phi', 'phi_mod', 'xy_mod' and 'z_mod' also have an '_init' modifier, which allows for the introduction of a proportionally fixed value of this parameter, without introducing a degree of freedom. This is handled by the prototype_polyhedron() function. The '_init' suffix is useful for the construction of many rigid polyhedron models, including the rigid base models for polyhedra with bundled parameters. As discussed in section 4, some model polyhedra require multiple parameters to be modified in a synchronized manner. This is referred to as a 'parameter_bundle'. It is defined in the dict_info section of a blueprint containing one. The parse_math() function has the main purpose of processing the mathematical expressions binding parameters to variables. This could also be done using eval() or probably some function from a module, but the custom function allows for dealing with floating point issues (e. g. arccos(1.000000000000002) nested somewhere in a mathematical expression).

The ConfigWindow and CustomPolyhedronWindow classes are responsible for the other two windows of the GUI. The VisualPolyhedron class is responsible for the graphical polyhedron models displayed in the top right corner of each GUI window. The Vec class is a reinvention of the wheel, but should be self-explanatory. The TreeNode class is used to construct data trees which hold blueprints or Polyhedron objects and encode their symmetry relations. These

are visualized in the 'results:' box of the main window and of utility for the recycle_belt_assignment() function. The remaining classes Buttoon, CritSearch, CheckList, RadioList and EntryList are GUI utility classes.

References

[1] M. Pinsky, D. Avnir, *Inorg. Chem.*, **1998**, 37, 5575.
[2] W. Kabsch, *Acta Crystallogr. A*, **1976**, 30, 513.