User Manual for Polynator 1.6

Contents

1 Introduction

Polynator is a computer program written in Python. Its main purpose is to fit models to atom arrangements. An atom arrangement is a set of atoms found in a crystal structure, typically a coordination environment or a molecule. A model represents a specific geometric shape, e.g. an octahedron or a pentagonal prism. Models are defined by a set of vertices which may be manipulated according to a set of rules specific to each model. In the context of computations, atoms and model vertices are internally represented by vectors. A model may be rigid or dynamic. Rigid models have fixed proportions, meaning they can be rotated and resized, but not deformed in any other way. Dynamic models can be deformed in a variety of ways (e.g. stretching, twisting, puckering...). Fitting a model to an atom arrangement allows for the quantification of the distortion of that atom arrangement with respect to that model. This allows the user to more precisely describe a coordination environment or molecule, compare distortions between similar structures, or find trends within structural families. Polynator can also be utilized to construct Voronoi polyhedra and to calculate angles, volumes and surface areas. In addition, it provides a convenient way to find and extract specific structural units from larger crystal structures. This documentation is intended to assist the user in navigating these features. It also gives an overview of the code structure and the mathematical methods fueling the program.

2 Getting Started

Polynator is available via 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). The graphical user interface is accessed by running the polynator_gui.py script. The main program is contained in polynator_main.py and can be run independently. 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.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.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 the tab bar after 10 seconds, check if the system requirements are met and that no changes have been made to the folder.

3 Mathematical Aspects

3.1 Overview

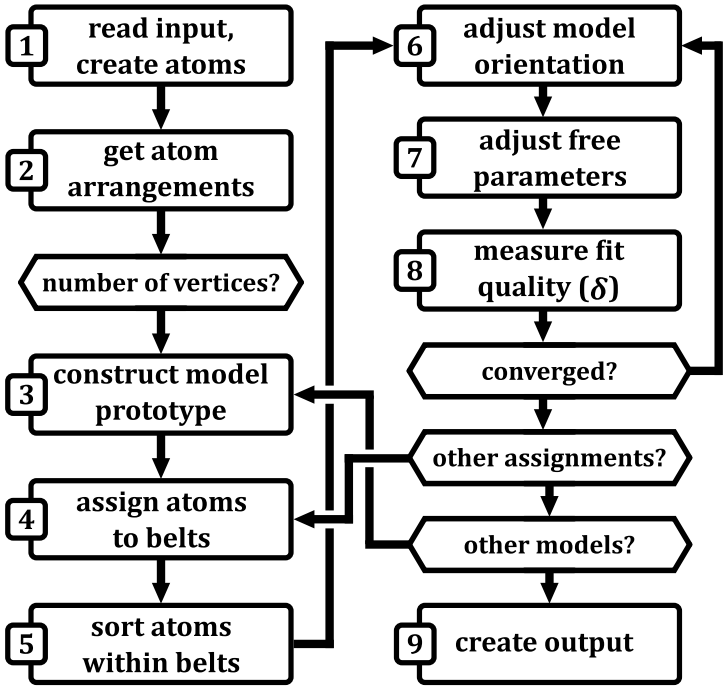

Figure 1: Flow chart of Polynators working process.

Polynator seeks to minimize the deviation δ, which is a least squares based metric for the purely geometrical dissimilarity between a set of pairs of atom and model vectors $\vec{a}_i$ and $\vec{v}_i$ defined as

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

where $\vec{c}$ is the centroid of all atom vectors. A perfect fit between model and atom vectors yields $\delta = 0$. Larger values indicate stronger distortions, up to a theoretical upper bound of $\delta = 100$. In practice, values above $\delta \approx 30$ should typically not be interpreted as validation for describing an atom arrangement in terms of the fitted model. Values in the range $18 < \delta < 30$ indicate strong distortions, while lower values correspond to moderate or small distortions.

The δ metric is closely related to the continuous symmetry measure (CSM) originally defined by Zabrodsky, Peleg and Avnir [3]. Values can be converted to and from the CSM-type S parameter via

$$S = 0.01 \cdot \delta^2 \quad \text{or} \quad \delta = 10 \cdot \sqrt{S} \tag{2}$$

However, there are several reasons for preferring the δ parameter. First of all, even though there are at least four types of values in the CSM family (CSM, CShM, SOM and CCM), dynamic models are not adequately covered by any of them. Secondly, CSM-type values increase approximately quadratically with increasing distortion, while δ increases approximately linearly, making it much more intuitive. Lastly, due to their quadratic nature, it is not uncommon for small distortions to register only in the third or fourth decimal place of CSM-type values, making them somewhat unhandy. With δ values, two decimal places are virtually always sufficient to distinguish between a perfect fit by symmetry and a tiny distortion.

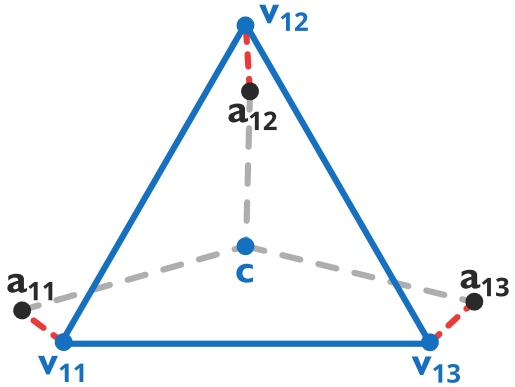

Figure 2: Measuring the value of δ for a regular triangular model (blue) fitted to three atom vectors (black). Red dotted lines contribute to the numerator and grey dotted lines to the denominator of equation 1.

To minimize δ, Polynator internally has to solve four main problems:

1. Pairing up atom and model vertices in an optimal fashion.

2. Finding the optimal coordinates to center the model at.

3. Finding the optimal orientation of the model. This has a reliable solution in the Kabsch algorithm [5], which will not be discussed here.

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

Table 1: Mathematical symbols presented in this chapter do not always match their counterparts in the code or the GUI. This table translates the code names.

symbol	code name	location in the code
$\vec{a}_{ij}$	vec_real	ModelFit.belts_real (a list of lists of vectors)
$\vec{v}_{ij}$	vec_model	ModelFit.belts_model (a list of lists of vectors)
$\mathbf{M}$	var_matrix	get_covariance_matrix_eigenvectors function
$\hat{m}$	model_axis	ModelFit.model_axis
S	HALF_SPHERE	assign_to_belts_from_scratch function
q_1	estimated_cost	AtomArrangement.get_assignment_cost _estimate method
q_2	total_cost	ModelFit.match_real_vecs_to_model method
P_{ij}	(various names)	ModelFit.adjust...() methods
s	'sc'	ModelFit.belt_dicts, ModelFit.dict_parameters
$\grave{h}$, $\grave{w}$	'>h', '>w'	ModelFit.belt_dicts, ModelFit.dict_parameters

3.2 Pairing Up Atom and Model Vertices

3.2.1 The Assignment Problem

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 was 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 and simple molecules can be exploited to develop strategies that come very close to this goal. Using these strategies, deviations from the optimal solution of the assignment problem will sometimes occur if the atom arrangement is unrecognizably dissimilar from the model, but are extremely unlikely if the

atom arrangement can reasonably be described as a distorted version of the model.

3.2.2 Geometrical Solutions

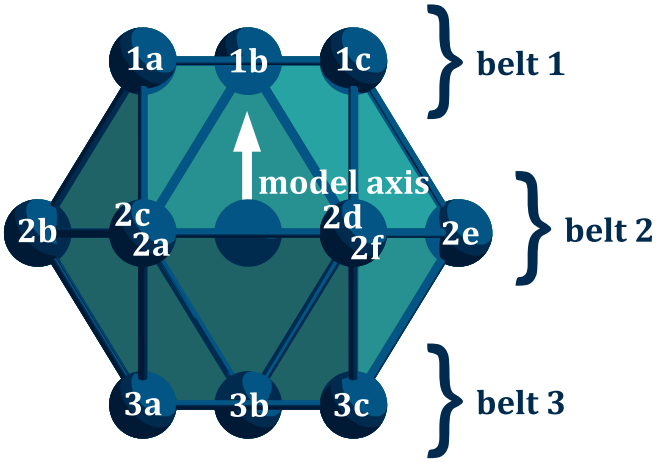

Figure 3: Separation of an anticuboctahedron into belts.

The first step towards this is to separate the model into subsections, which we will call belts. This separation into belts is specific to each model. Belts can be thought of as rings of vertices which lie perpendicular to the model axis (see fig. 3). The model axis is a high symmetry axis of the model (in some cases, the choice may be ambiguous, e.g. either three- or fourfold rotation axes are valid for models with a cubic point group). Depending on the model, vertices are assigned to belts following one out of several strategies. In all cases, the origin is set to the centroid of all atoms which contribute to the fit.

For models representing coordination polyhedra or cages, vertices are assigned to belts based on strategy 1, which assumes that the atom arrangement is roughly spherical. The atom vectors are projected onto the surface of a unit sphere around the centroid. To assign each atom vector to a model belt, (not yet to a specific vertex), a tentative model 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. Models 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 estimated assignment cost q_1. To obtain it, the model 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 $\vec{b}$:

$$\vec{b}_{ij} = \vec{a}_{ij} - \vec{c}_i \,. \tag{3}$$

2. A covariance matrix $\mathbf{M}$ is constructed from the cartesian coordinates x, y and z of the $\vec{b}_{ij}$ vectors:

$$\mathbf{M} = \begin{pmatrix} \sum_{ij} x_{ij}^2 & \sum_{ij} x_{ij} \cdot y_{ij} & \sum_{ij} x_{ij} \cdot z_{ij} \\ \sum_{ij} y_{ij} \cdot x_{ij} & \sum_{ij} y_{ij}^2 & \sum_{ij} y_{ij} \cdot z_{ij} \\ \sum_{ij} z_{ij} \cdot x_{ij} & \sum_{ij} z_{ij} \cdot y_{ij} & \sum_{ij} z_{ij}^2 \end{pmatrix} . \tag{4}$$

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

4. The estimated cost q_1 for this assignment can now be calculated as

$$q_1 = \sum_i |\vec{c}_i \times \hat{m}|^2 \cdot n_i + \sum_{ij} (\vec{a}_{ij} \cdot \hat{m})^2 \tag{5}$$

   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 cost q_1 are evaluated further, the rest is discarded.

Some models, for example those for the biphenyl and porphyrin skeletons, do not work very well with strategy 1. Instead, atom vertices are assigned to their belts based on the much simpler strategies 2 or 3, respectively (this is encoded by a 'shape_type' entry in the dict_info of those model blueprints). Strategy 2 is designed to assign atom vectors to the belts of very prolate models. To do this, the most prolate axis of the atom arrangement has to be identified first. Thankfully, this is easily achieved: it is the eigenvector with the largest eigenvalue

of the covariance matrix constructed from all atom vectors. The atom vertices are then simply ranked according to their dot product with the prolate axis and sequentially filled into belts in this order. Strategy 3, designed for complex oblate models, such as the porphyrin skeleton, works similarly. The most oblate axis of the atom arrangement is obtained as the eigenvector with the smallest eigenvalue of the covariance matrix constructed from all atom vectors. Atom vertices are ranked according to their cross product with this axis and sequentially assigned to belts.

Some models can be thought of as derivative of one or more other models. For example, the tetragonal prism can be derived from the cube by adding a degree of freedom (independent height and width parameters instead of a single scaling parameter). For such a derivative model, the belt assignment step can effectively be skipped if the parent model fits sufficiently well. The belt assignment is then simply inherited from the parent model according to instructions encoded for each derivative model.

To pair up atom and model vectors, the combinations that maintain the order of dihedral angles (up to one per atom) are ranked by their estimated cost q_2. To do this, the model is first rotated around $\hat{m}$ so that the atom vector $\vec{a}_{11}$ lies in the plane containing $\hat{m}$ and an arbitrarily picked model vertex $\vec{v}_{\text{tare}}$. 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 estimated cost q_2 is measured as

$$q_2 = \sum_{ij} |\vec{v}_{ij} \times \hat{m}| \cdot |\Delta\alpha_{ij}| \tag{6}$$

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_2 is evaluated further, the rest is discarded.

3.2.3 Graph Theoretical Solution

A graph is a set of vertices which are connected by a set of edges. The graph of an atom arrangement refers to the set of atoms (the vertices) in that arrangement which are connected by bonds (the edges). Geometric information, such as distances and angles, is ignored in this context once bonds between atoms have been established. An automorphism is any permutation of the vertices of a graph that preserves the connectivity of the graph (see fig 4). Provided that the graph of an atom arrangement is isomorphic (equivalent) to the graph of a given model, the vertices can be paired up in a typically very limited number of permutations. More specifically, the number of required permutations is at most the order of the automorphism group of the graph. The rotation group of the model divides the automorphism group into cosets. Only one element from each of these cosets needs to be evaluated, as the others are congruent. As a result, most models representing polyhedra require only two permutations, one corresponding to the identity and the other to an inversion or reflection. Models of regular polygons, as well as chiral polyhedron models such as the snub cube, require only the identity. However, models representing molecules with tree-like branching tend to have a number of automorphisms that do not correspond to rotational symmetry. For example, the graph of the model for a ψ^1-pentagonal bipyramid is a star with six leaves (see fig 4). Its automorphism group has the order $6! = 720$. This number can be divided by the order of the rotation group of the model to obtain $\frac{6!}{5} = 144$ vertex permutations, no two of which can be superimposed onto each other by rotating the model.

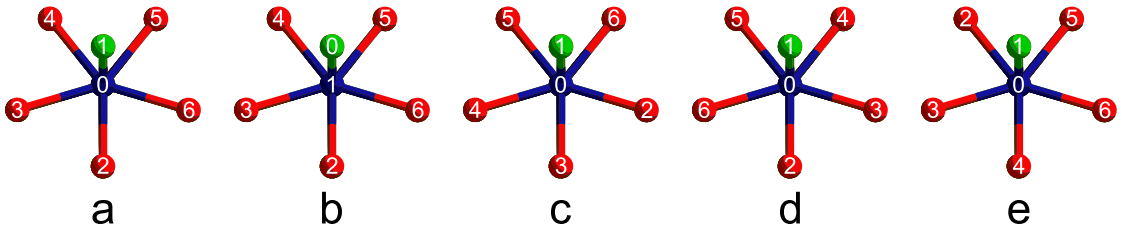

Figure 4: Some vertex permutations of the model of a ψ^1-pentagonal bipyramid, with (a) representing the identity permutation. All of these, except for (b), are automorphisms of the model graph. Permutation (c) is equivalent to (a) via rotation by 72 °. Permutation (d) is equivalent to (a) via reflection, but not congruent to (a). Permutation (e) is one of many automorphisms of the model graph which do not correspond to geometric symmetries of the model.

3.3 Centering the Model

Each models in Polynator are defined such that the centroid of all model vertices always rests on the origin. Due to the nature of the least squares fit, the centroid of all fitted atom vectors must also be located at the origin to achieve an optimal fit. The properties of the least squares fit entail that the optimal orientation and shape of the model vertices can be computed independently from the centering. Thus, the atom vectors could in principle be kept in place, correcting for the misalignment of the two centroids only retroactively. However, Polynator translates the atom vectors such that the two centroids coincide before starting the fitting procedure.

3.4 Optimization of Shape Parameters

In the simplest case, the model 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

$$P_{ij} = \vec{a}_{ij} \cdot \hat{v}_{ij} \,. \tag{7}$$

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

$$s_{\text{opt}} = \frac{\sum_{ij} P_{ij}}{\sum_{ij} 1} \,. \tag{8}$$

Otherwise it is solved by iteratively minimizing the sum of the squared 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 dynamic model of the 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 models, such as dynamic prisms, antiprisms, pyramids..., require the separate optimization of height and width parameters h and w (instead of a single s parameter, not in addition to it). Thanks to the Pythagorean theorem, this is easily done. To optimize h, instead of projecting the atom vectors onto the model

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

$$P_{ij} = \vec{a}_{ij} \cdot \hat{m}\,. \tag{9}$$

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

$$P_{ij} = \vec{a}_{ij} \cdot \hat{n}_{ij} \quad \text{where} \quad \vec{n}_{ij} = \vec{a}_{ij} - (\vec{a}_{ij} \cdot \hat{m}) \cdot \hat{m}\,. \tag{10}$$

As with the s parameter, the optimal values of h and w can usually be obtained analytically by computing the arithmetic mean of the P values. However, there are the special variants $\grave{h}$ and $\grave{w}$ which may have different proportionality constants for different belts. These, again, need to be solved iteratively.

Parameters φ for rotations (used e.g. in twisted prisms) are also straightforward, but computationally more expensive, as they require vector operations during iterations. In each cycle of such an iteration, the model vectors are incrementally rotated around $\hat{m}$. Subsequently, the sum of the squared distances between atom and model vectors is measured. If the fit deteriorated compared to the last cycle, the increment is multiplied with $-\frac{1}{2}$. Once the magnitude of the increment falls below $10^{-(4+n)}$, where n is the loop count of the overall optimization (ModelFit.loop_count), the iteration is terminated. To make the computation slightly more efficient, the atom and model vectors are first transformed into two-dimensional cartesian and polar coordinates, respectively (as evident from the Pythagorean theorem, the component in the direction of $\hat{m}$ doesn't affect the result). In the built-in models, φ is always balanced by a counterrotation $\varphi*$ in a different belt in order to minimize the dependence on 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 $\tilde{h}$, $\tilde{w}$ or $\tilde{\varphi}$, 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 $\tilde{h}$ and $\tilde{w}$, sin for $\tilde{\varphi}$). The frequency f and offset σ of these functions are determined when a model 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):

$$g(p) = \begin{cases} \tilde{h},\ \tilde{w} & \rightarrow \cos\left(\frac{2\,\pi \cdot f \cdot (p+\sigma)}{n}\right) \\ \tilde{\varphi} & \rightarrow \sin\left(\frac{2\,\pi \cdot f \cdot (p+\sigma)}{n}\right) \end{cases} \tag{11}$$

The divergent choice of basic trigonometric functions may seem odd, but in practice makes it easier to think about these, due to their transversal ($\tilde{h}$, $\tilde{w}$) and longitudinal ($\tilde{\varphi}$) 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 models 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 newly calculated and applied to the model in each iteration.

After all shape parameters have been optimized, the orientation is adjusted again, taking into account the new shape of the model. This process is repeated for up to ten cycles or until the δ value ceases to improve by more than 0.000003 between cycles.

3.5 Generic Distortion Values

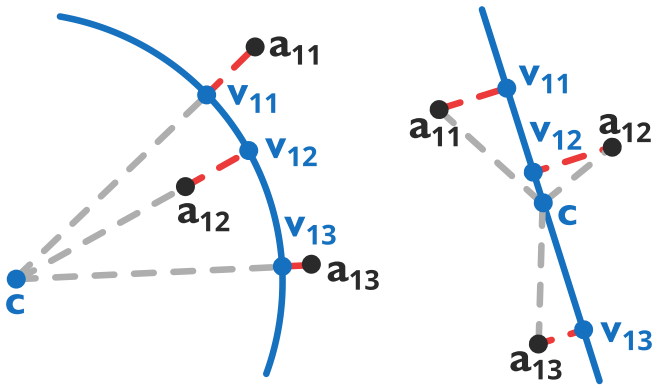

Figure 5: Measuring the value of $\delta_{\text{spherical}}$ (left) and δ_{linear} (or δ_{planar}, right). Red dotted lines contribute to the numerator and grey dotted lines to the denominator of equation 1.

For each atom arrangement, Polynator calculates three generic distortion values: δ_{linear}, δ_{planar} and $\delta_{\text{spherical}}$. A 'model' for this kind of fit is not constrained by specific proportions, but by an overall shape which must contain all its vertices. These shapes are respectively a line, a plane and the surface of a sphere. The most closely fitting line and plane normal are easily obtained as eigenvectors of the covariance matrix of all atom vectors. For each atom vector, the corresponding 'model vector' is the nearest point on that line or plane. For $\delta_{\text{spherical}}$, the most closely fitting sphere surface centered at the centroid has a radius equal to the average distance of each atom vector from the centroid. Note that $\delta_{\text{spherical}}$ does not necessarily refer to the most closely fitting sphere surface with freely selected centering. This becomes obvious when evaluating the case of four randomly distributed atom vectors: absent degenerate cases, it is always possible to find a perfectly fitting sphere surface by centering it on the circumcenter of the four vectors, which generally does not coincide with the centroid.

3.6 Convex Hull and Minimal Bounding Sphere

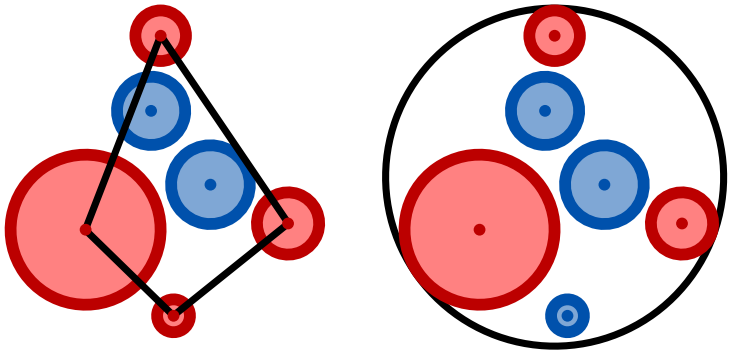

Figure 6: Two-dimensional convex hull (left) and minimal bounding circle (right) of a set of circles with a variety of radii. Circles are colored red if they lie on the respective boundary and blue otherwise.

Polynator is capable of computing the convex hull and the minimal bounding sphere of an atom arrangement or model. The convex hull is defined as the smallest convex shape which encompasses a set of points, or, in this case, atomic coordinates. It is used for finding the bounding edges, which are used as graph edges for coordination environments constructed with the *gap method* and *fixed number* options. It also gives a useful metric for the volume of an atom arrangement, especially in inorganic crystal structures.

The minimal bounding sphere is defined as the smallest sphere containing a set of objects. It is primarily useful as a metric for the size of small or relatively rigid molecules. Polynator computes the minimal bounding sphere treating atoms not as points, but as spheres. Each atom sphere is centered at the atomic coordinates and its radius is the atomic radius (as given in the *atomic radii* panel or otherwise taken from a default table). The volume of the minimal bounding sphere is typically much greater than that of the convex hull. A modified, iterative version of Welzl's algorithm [1] is used to determine the center and radius of the minimal bounding sphere. According to [2], it should technically be possible to construct a set of spheres for which this algorithm fails, but we have never observed this in practice.

3.7 Graph Tracing

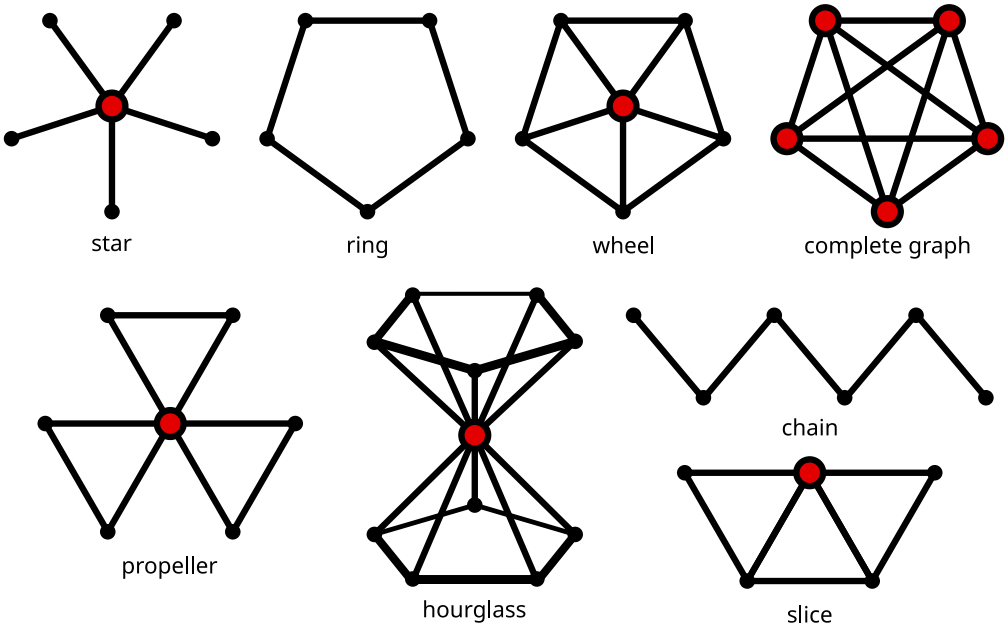

Figure 7: Primitive graph types used by Polynator. Universal vertices are highlighted in red.

As explained in section 3.2.3, Graph theory is a field of mathematics concerned with networks of vertices which are connected by edges. Polynator uses it to identify and isolate specific subgraphs in graphs of larger molecules or infinite frameworks. For this purpose, it constructs a simple, undirected graph by connecting the atoms in a given structure via one of the methods discussed in section 4.2. The graph tracing algorithm then finds all crystallographically unique subgraphs in this full graph that are isomorphic to the specified subgraph. In other words, it implements a substructure search without preceeding knowledge of the structures to be searched. Fig. 7 shows a number of primitive graph types which can be traced directly using atoms as vertices. If the target has a universal vertex, each unique atom in the crystal structure is checked by evaluating all combinations of the appropriate number of neighbors of that atom. If the strategy aims at a star, propeller or complete graph, this evaluation is purely based on the neighbor vertex degrees. For slices, wheels and hourglasses, in addition to that, it is necessary to check how many disjoint subgraphs would result if the traced graph would be isolated and the universal vertex removed. If the target is a chain or ring, candidates are obtained by iteratively adding members starting at each unique atom.

More complex graphs are traced in a bottom-up, multi-step process, with each step using the subgraphs found in previous steps as vertices. For example, Polynator uses a two-step process to search the graph of a crystal structure[1] for all subgraphs constituting truncated tetrahedra. In the first step, all six-membered rings are identified. These rings are treated as vertices in the second step. An edge is established between any two such vertices if the corresponding six-membered rings share exactly two atoms. The algorithm then looks for tetrahedra (complete graphs with four vertices) in the abstract graph constructed this way. Each finding is checked by looking at the degree (number of next neighbors) of each atom. In a truncated tetrahedron, each atom will have degree 3. No other graph which can be traced this way has this property.

Unfortunately, it is not always sufficient to check only the atom degrees. In some instances, as illustrated in fig. 8, compiling the degrees of all vertices does not unambiguously identify the target graph. If this is the case, it is required to check the shell IDs. This is done by iterating over all vertices in the candidate subgraph, producing a list of numbers for each of them. This list contains the number of vertices which are one edge away from the sample vertex, then the number of vertices which are two edges away, and so on until each vertex in the subgraph is registered. This yields a relatively compact identifier which is not always strictly unique to a graph, but in practice sufficient to differentiate between graphs given that were traced from specific components using the bottom-up method explained above.

[1] More precisely, an excerpt from a crystal structure, often a contiguous molecule or a $3 \times 3 \times 3$ cell.

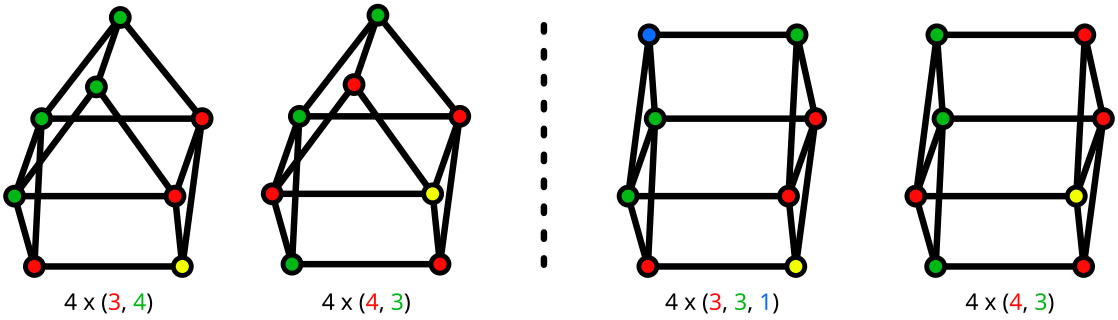

Figure 8: Shell IDs of the gyrobifastigium (left) and the orthobifastigium (right). The vertices of each graph can be split into two equivalence classes, each containing four vertices. The graphs can be distinguished by counting the number of vertices in a distance of one, two or three edges from the yellow vertex. Those vertices are colored red, green and blue, respectively.

4 Using Polynator

4.1 The Graphical User Interface (GUI)

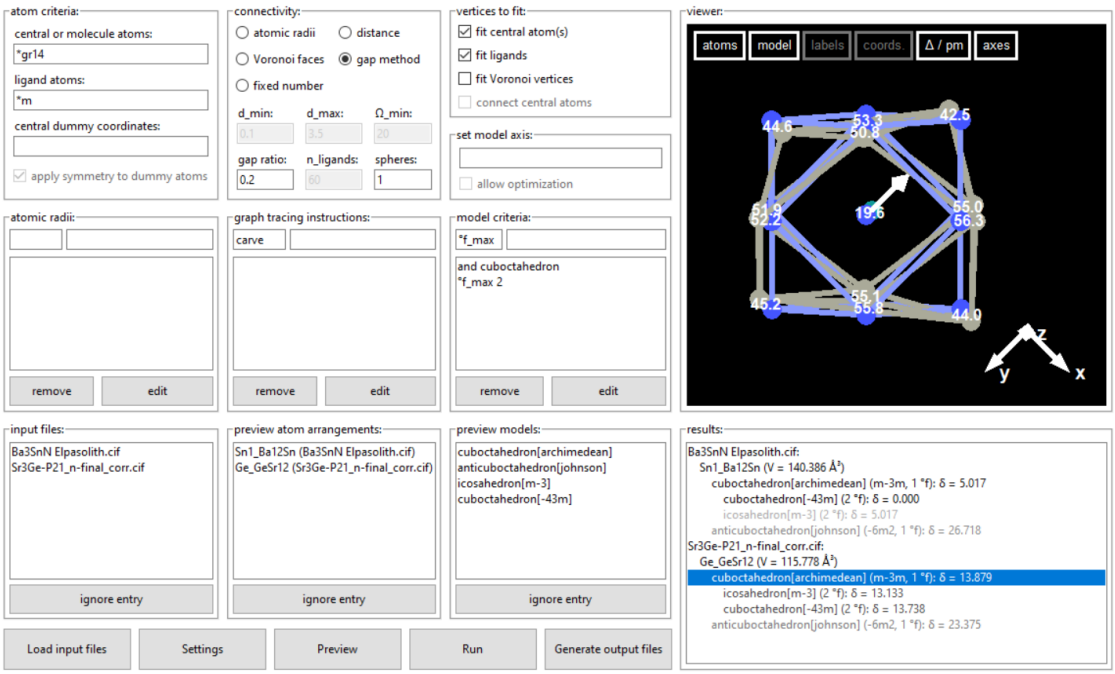

Figure 9: Screenshot of the main window of the GUI.

The GUI comprises a total of five windows:

- The main window is opened when starting the program. It allows the user to find atom arrangements, fit models and generate output files.

- The settings window is accessible from the main window via the *Settings* button. It allows for modifications to some aspects of the program.
- The custom model construction window allows the user to create new models. It is accessible via *Settings* → *customize: models*.
- The symmetry relations window allows the user to define new symmetry relations between models. This can be useful to properly link up custom models with existing models. Symmetry relations help to properly pair up atoms with model vertices in some cases. This window is accessible via *Settings* → *customize: model relations*.
- The custom graph template window allows the user to define new graphs to trace (see section 4.4). It is accessible via *Settings* → *customize: graph templates*.

Each window contains several framed, labelled panels. Each window has a *screen* panel at the top right position. On the main window, this allows for the visualization of unit cells, graph templates, atom arrangements, models and fit results by clicking on entries of the *input files*, *graph templates*, *preview atom arrangements*, *preview models*, or *results* panels, respectively. The visualizations for models include the deformations caused by free parameters. These can be toggled separately at the bottom left corner of the *screen* panel. Visualizations of results show an overlay of atom arrangement and fitted model, each of which can be hidden separately by clicking on the appropriate boxes at the top of the panel. They also show the distances between paired atom and model vectors (Δ/pm).

Each window has some space on the bottom for the display of comments and error messages. When hovering over a widget, a comment about that widget is displayed there in most cases. When clicking on an entry in the preview models panel, some information about that model is displayed in the comment bar. These features can be disabled in the settings menu.

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 or .xyz files on your computer. A click on *Run* starts Polynator's main process. However, depending on the crystal structures and the type of atom arrangement which you

are interested in, some additional input may be required, as explained following within this chapter.

4.2 Establishing Atomic Bonds

Constructing an atom arrangement to be evaluated generally requires connecting those atoms by establishing bonds first. In the *connectivity* panel, the user may choose from among five different procedures to achieve this, although the *gap ratio* and *fixed number* options are limited to coordination environments with ligands surrounding a single central atoms.

- *atomic radii*: With this procedure, two atoms are connected if the distance between them is smaller than the sum of their atomic radii. A fixed table of covalent radii is used by default, but radii for individual atoms or elements can be adjusted using the *atomic radii* panel.

- *distance*: Two atoms are connected if the distance between them is larger than d_{min} and smaller than d_{max}. These values can be freely adjusted in the lower part of the *connectivity* panel.

- *Voronoi faces*: This procedure is based on a Voronoi diagram. Two atoms are connected if their Voronoi polyhedra share a face and the solid angle subtended by that face is greater than the threshold Ω_{min}, which can be freely adjusted in the lower part of the *connectivity* panel. A similar algorithm is implemented in *ChemEnv* by Waroquiers et al. [1], who took inspiration from O'Keeffe [2]. The Voronoi procedure generally yields reasonable results for non-planar coordination environments. It is, however, by far the most computationally expensive option.

- *gap ratio*: For this procedure, the atoms in the vicinity of a central atom are sorted by their distance d from that central atom. If the gap in d between two consecutive ligands is larger than a threshold value, this is considered the beginning of a new coordination sphere. The threshold value is calculated as the product of the smallest central–ligand distance times a coefficient with a default value of 0.2 (*gap ratio*). The gap method often allows for easy access to any of the first few coordination spheres. However, for more disordered coordination environments, where coordination spheres aren't as neatly separated, it becomes much less useful.

- *fixed number*: The *Voronoi method* is based on the construction of a Voronoi polyhedron around the central atom. Atoms are considered ligands if their respective Voronoi polyhedron shares a face with the Voronoi polyhedron of the central atom and this shared face subtends a solid angle greater than a given threshold (20° by default) from the perspective of the central atom. A similar algorithm is implemented in *ChemEnv* by Waroquiers et al. [1], who took inspiration from O'Keeffe [2]. Note that all methods to select ligands come with their own biases and will work better for some types of coordination environments than for others. That said, the Voronoi method generally yields reasonable results as long as the coordination environment is non-planar.

  Atomic radii allow for more selectivity 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 in either central atom or ligand atom criterion box, then enter a number into the respective *set radius* field and press enter.

4.3 Coordination Environments

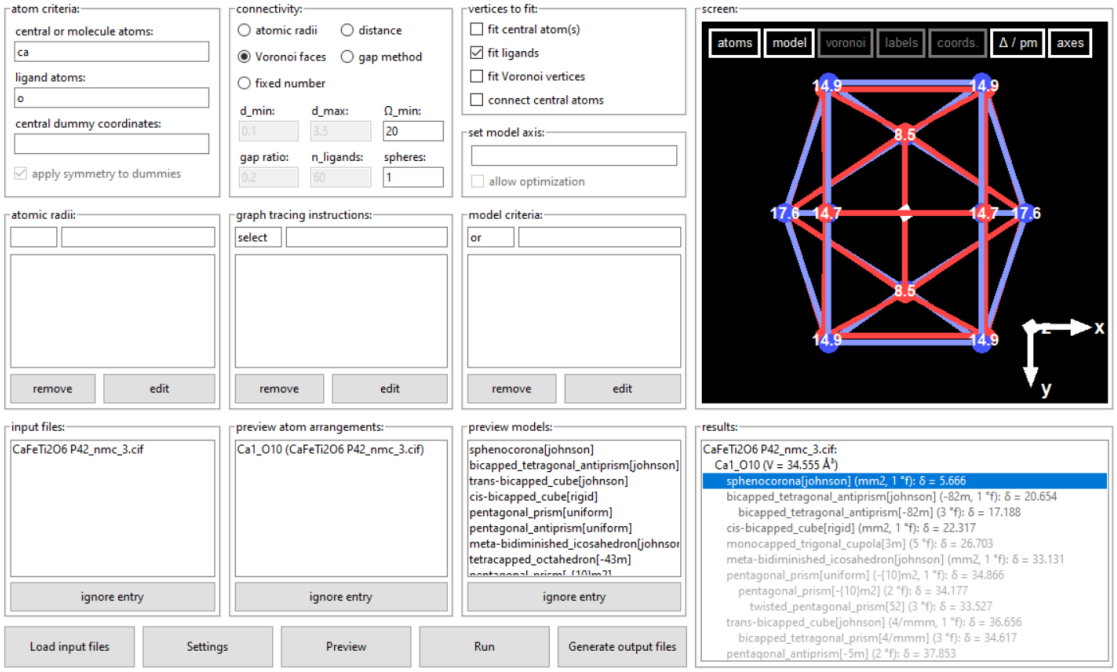

Figure 10: Evaluating a coordination environment in Polynator. As a result of the user-given atom criteria, only calcium and oxygen atoms are valid as central atoms and ligands, respectively. Atoms are connected based on shared Voronoi faces. The central calcium atom does not contribute to the fit and is therefore not shown on the screen.

The *atom criteria* panel allows for the selction of specific central and ligand atoms. A valid entry may be an element symbol, a specific atom name (e.g. Ca1) or a wildcard (e.g. '*M' for any metal, see tab. 2). Prefixing an entry with a minus sign (e.g. '-Ca1' or '-*M') excludes atoms that fit this entry. Prefixing an entry with an exclamation mark (e.g. '!Ca1' or '!*M') requires each atom arrangement to contain at least one of each requested atom. You may also leave these fields empty, in which case all possible central and ligand atoms will be evaluated.

Table 2: 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 and actinoids)
*RE	all rare earth elements including Sc, Y, lanthanoids and actinoids.
*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)

By default, any atom with a distance from the central atom larger than $d_{\mathrm{min}} = 0.1$ Å and smaller than $d_{\mathrm{max}} = 3.5$ Å will qualify as a ligand. These values can be freely adjusted in the *connectivity* panel. The same is true for the maximal coordination number $\mathrm{CN}_{\mathrm{max}}$. If necessary, the most distant ligands in excess of this number will be dropped until the number of ligands is equal to $\mathrm{CN}_{\mathrm{max}}$.

As alternatives to these simple cap values, the *fit settings* panel gives access to two algorithms which select the ligands for a coordination environment according to specific rules. The *gap method* looks for gaps in the distance distribution of ligands from the central atom. If a gap between two consecutive ligands is larger than a threshold value, this is considered the beginning of a new coordination sphere. The threshold value is calculated as the product of the smallest central–ligand distance times a coefficient with a default value of 0.2 (*gap size*). The gap method often allows easy access to any of the first few coordination spheres. However, for more disordered coordination environments, it becomes much less useful, as coordination spheres aren't as neatly separated anymore. The *Voronoi method* is based on the construction of a Voronoi polyhedron around the central atom. Atoms are considered ligands if their respective Voronoi polyhedron shares a face with the Voronoi polyhedron of the central atom and this shared face subtends a solid angle greater than a given threshold (20° by default) from the perspective of the central atom. A similar algorithm is implemented in *ChemEnv* by Waroquiers et al. [1], who took inspiration from O'Keeffe [2]. Note that all methods to select ligands come with their own biases and will work better

for some types of coordination environments than for others. That said, the Voronoi method generally yields reasonable results as long as the coordination environment is non-planar.

Atomic radii allow for more selectivity 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 in either central atom or ligand atom criterion box, then enter a number into the respective *set radius* field and press enter.

Polynator will by default not include the central atom in the arrangement of atoms which are to be fitted. To include the central atom, check the box *fits include central atoms* in the *fit settings* panel. In this case, the central atom is fitted against the centroid of all atom vectors and will also factor into the centering of the model. This panel also gives you the option to fit the vertices of a Voronoi polyhedron constructed around the central atom instead of the actual ligands.

4.4 Molecules and Cages

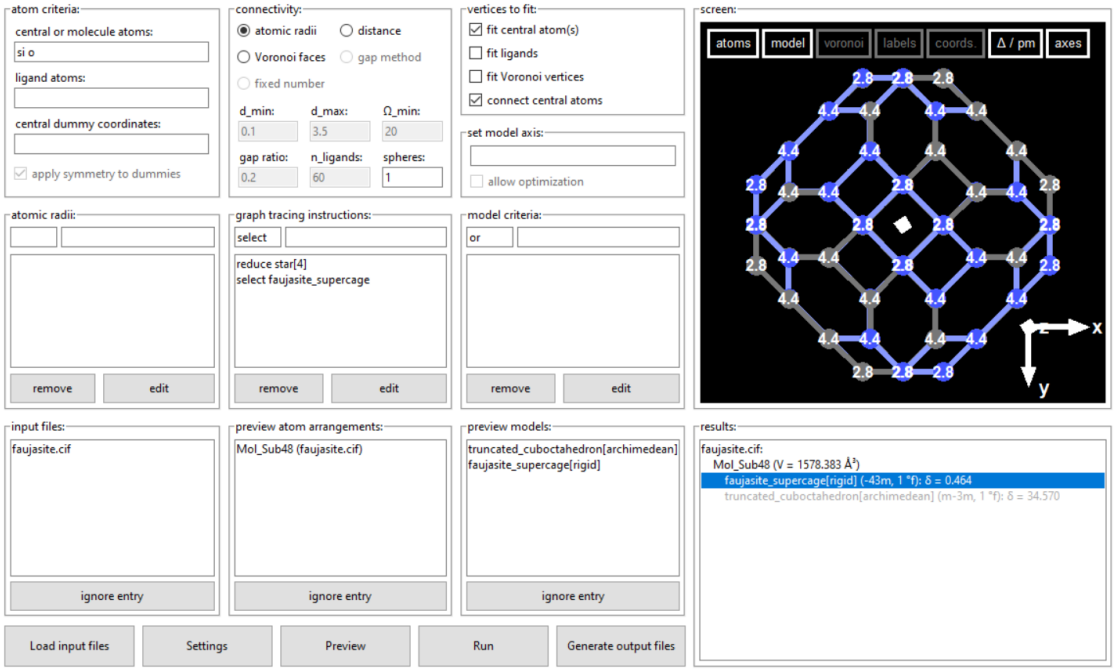

Figure 11: Analyzing a faujasite supercage in Polynator. Atoms are connected based on their default atomic radii. The cage is formed by 48 SiO_4 tetrahedra, each of which is collapsed into its respective centroid by the 'reduce_star[4]' instruction. This results in an infinite framework of abstract vertices representing the tetrahedra. To isolate one cage, Polynator is instructed to trace and select a 'faujasite_supercage' graph.

To find molecules and empty cages in a crystal structure, select the *connect central atoms* option in the *vertices to fit* panel. This will connect the selected atoms based on the chosen *connectivity* method. You may also choose from among a number of *graph tracing instructions* to conveniently isolate a particular subunit from a larger molecule or an infinite framework of connected atoms. This allows you to isolate e.g. a sodalite cage from the infinite framework of silicon atoms in a zeolite structure or to obtain all hexagonal rings from an organic molecule. While most instructions will only look for one specific shape, some are intentionally designed to find a slightly wider variety. For example, the instruction for fullerenes will find any polyhedron with only pentagonal and hexagonal faces where each atom has degree 3. Keep in mind that appropriate atom connectivity rules are a prerequisite for finding the desired shape! Additional graph templates can be defined via *Settings* → *graph templates*.

4.5 Graph Tracing

Polynator is able to search a crystal structure for specific shapes on a graph theoretical basis (see section 3.7). A prerequisite for this is the establishment of bonds via the *connectivity criteria* panel. It is generally recommended to use the *atomic radii* or *distance* options for this purpose. The connected atoms form a graph, i.e. a set of vertices (atoms) connected by edges (bonds). This graph, which comprises all generated atoms, can be traced in order to find specific subgraphs, e.g. six-membered rings, icosahedra, supertetrahedra, etc. Computation times for graph tracing operations can vary greatly, ranging from a few milliseconds to several minutes on a regular PC. This is due to the number of combinations scaling factorially with the density of connections in a crystal structure. For this reason, it is advisable to make sure no unintended bonds are created and no unintended atoms included.

To utilize this feature, enter the name of a target graph in the top right field of the *graph templates* panel and confirm your choice by pressing Enter, thus transferring the request to the box below. A graph template instructs Polynator's graph tracing algorithm to search the graphs of all input structures for unique instances of the target subgraph. In the *select* mode, the atoms in a successfully traced subgraph are processed as the central atoms in an atom arrangement. This mode can also be used without checking the *connect central atoms* box. In that case, a coordination environment is discarded if the ligands do not form any of the specified subgraphs. The *block* mode blocks successfully traced graphs so that no subgraphs of a blocked graph can be found in the *select* mode. For example, when tracing tetragonal pyramids in the *select* mode, if the crystal structure contains an octahedron, the tracing algorithm will normally find a number of tetragonal pyramids, since they are just octahedra with one of the vertices missing. However, if there is a additional tracing request for octahedra ($\equiv$ trigonal antiprisms) in the *block* mode, the pseudo-pyramids will be ignored. In the *reduce* mode, the atoms in each successfully traced subgraph are collapsed into a single dummy atom. This mode is applied before the others and the graph modified this way is then available for further tracing operations, including nested reductions. This can help to simplify an atom arrangement, for example by reducing a tetrahedral unit to a single pseudo-atom.

There are a number of parametrized graph types, which appear with a question mark in square brackets in the dropdown menu of the *graph templates* panel. To register such an entry, the question mark has to be replaced by an integer. For chains and rings, the integer refers to the number of vertices. For stars, it is the number of arms and for pyramids, bipyramids, hourglasses, prisms, antiprisms and elongated bipyramids, it denotes the number of atoms in the base. Supertetrahedra contain both central atoms and ligands of the constituent tetrahedra, but no bonds between ligands (i.e. the basic building block is a star[4]). The integer value is equivalent to the integer in the established notation of T2, T3 etc. for supertetrahedra.

4.6 Selecting Specific Models

By default, all available models with the respectively appropriate number of vertices will be fitted to each atom arrangement. The name of each model contains some clarifying information in the square brackets. Platonic, Archimedean, Catalan and Johnson solids, as well as uniform prisms and antiprisms are marked as such. Dynamic models have their point group in square brackets if they represent the set of all shapes with that point group and that topology. If 'rigid' or 'dynamic' is in square brackets, the model has a specific shape which is in most cases explained in the comment line upon selecting such a model. The *model criteria* panel allows you to select or exclude specific models. This panel will also accept fragments of valid entries. This is in contrast to atom criterion panels discussed above (otherwise 'N' would also find Nb, Ni, Zn etc.). A model may have more than one valid name. These synonyms are displayed in the comment line upon selecting a model in the *model preview* panel. Models can also be filtered by tags (see tab. 3) and by point group. You may select the '°f_max' mode and enter an integer to exclude all models with more degrees of freedom than that number (not counting the three translational and two rotational degrees of freedom available to every model). For example, entering '1' will exclude all dynamic models, leaving only rigid models such as the Platonic and Archimedean solids. The *model exclusion criteria* in the *Settings* menu serves a similar function, but will exclude unwanted models permanently.

Table 3: Model tags. In contrast to earlier versions, tags do not bestow any properties onto a model. However, they may hint at specific behaviors. For example, models with the *#prolate* or *#oblate* tags use strategies 2 and 3 for their assignment, respectively.

#rigid	*#dynamic*	*#symmetry_aligned*
#prolate	*#oblate*	*#constrained_parameters*
#molecule	*#cage*	*#occupied*
#pseudopolyhedron	*#chiral*	*#essential*
#equilateral	*#equidistant*	*#planar*
#regular_polygon	*#platonic*	*#archimedean*
#johnson	*#catalan*	*#deltahedron*
#fullerene	*#frank_kasper*	*#capped_cube*
#pyramid	*#bipyramid*	*#heterobipyramid*
#scalenohedron	*#prism*	*#antiprism*
#twisted_prism	*#frustum*	*#antifrustum*
#equator-capped_prism	*#axis-capped_prism*	*#fully_capped_prism*
#capped_frustum	*#capped_antifrustum*	

4.7 Setting a Model Axis

The user has the option of manually entering a model axis (see chapter 5, fig. 3). This bypasses the automatic belt assignment step and forces Polynator to assign belts according to this axis. This might be useful if an automatic belt assignment appears suboptimal. An axis is entered in the form of fractional coordinates, separated by commas or spaces. Unless the *allow optimization* box is ticked, this axis will remain unchanged throughout the entire fitting process.

4.8 The Settings Menu

The settings menu comprises several options to customize Polynator's behaviour. Changes made in this menu are remembered between sessions (they are stored in .cfg files located in the same folder as the main .exe or .py file.). The available options are as follows:

- Output behavior: Checkboxes allow the user to select which types of output files to produce. They also give the options to overwrite existing output files, to automatically produce outputs after every fit and to have only the best fitting model for each atom arrangement appear in the output.

- Skipping redundant models: If a model turns out to fit perfectly, versions of that model with additional degrees of freedom are not evaluated if the *skip redundant models* box is checked.

- Fit metric: Distortion values can be displayed either in the native δ metric or as CS(h)M-type values S (See section 3.1). The fitting procedure is not altered by either choice, since both values are minimized in the same fashion.

- Point group notation: The user is given the choice to have point group symbols displayed in Hermann-Mauguin (default) or Schönflies notation.

- Maximal δ value: Provides an upper threshold for the distortion. Models with a higher value do simply not appear in the output or the results box.

- Variable tax: The internal parameter δ_{taxed} is obtained by adding a small number (the variable tax) to the neutral δ value of a fitted model for each free variable that model has. The value of δ_{taxed} is not displayed anywhere, does not appear in the output files and does not influence the fitting procedure. However, it is consulted when deciding on the best fitting model, it influences the grayscale in the *results* box and determines the order of the list of models in the .out files.

- Model filter: The *active models* panel makes it possible to customize the set of models Polynator is actively using without having to specify preferences in the *model criteria* panel of the main window every time. For example, if 'exotic' model polyhedra are generally not useful to you, it may be beneficial to enter and confirm the exclusion criterion *not #essential* (see tab. 3 for a list of such model tags).

4.9 The Custom Model Construction Menu

The custom model construction window is accessible via *Settings* $\rightarrow$ *customize: models*. It allows for the creation of new models. The central component of this is worked out in the *belts* panel, where the user can add or remove belts (see fig. 3) with a number of vertices n. There is also a number of optional parameters which can be applied to a belt (see section 3.4). These parameters are perhaps best understood by selecting various model polyhedra in the *preview models* box at the bottom of the main window and observing them in the *screen* 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 models. The comment panel at the bottom of the window will guide you to some extent.

In order to guarantee a valid optimization procedure, some limits are placed on the customizability of a model. Firstly, the centroid of each belt must rest on the model axis. This means it is not possible to define e.g. a capped pentagonal pyramid (with the cap on a triangular face). Allowing imbalanced belts would introduce a nontrivial optimization problem for the centering of the model. For similar reasons, the centroid of all model vertices must rest on the origin at all times. Secondly, the frequency of each modulated parameter in a belt must be a product of one or more prime factors of the number of vertices n in that belt. Otherwise, imbalances and coupling between the optimization of orientation and shape would arise. If a model contains one or more s parameters, the average distance from the origin to the model vertices scaled by that parameter must be exactly 1.0. This ensures that the optimized parameter value matches the actual size of the model. Finally, if a belt contains a φ parameter, there must either be another belt with the same number of vertices and the same φ parameter in counterrotation (e.g. '-phi1'), or another belt with a φ parameter entitled 'phi*'. This eliminates or at least minimizes coupling between the optimization of orientation and those torsion angles, which would lead to slow convergence.

The new model also needs a name. Additionally, you have the option of assigning a point group, a symmetry operation pertaining to the model axis, a list of parent models, i.e. preexisting models with higher symmetry or fewer degrees of freedom, as well as a list of search terms or categories for the new model. However, these have no functionality other than being displayed at various points and allowing the user to search for them.

Lastly, constraints ('bundles') may be added. These are useful for dynamic models which do not have all of the degrees of freedom their point group would allow. Defining these is not always a simple task and may require some calculations. Some guidelines will be given here, but it is recommended to look at existing models with constrained parameters (They all share the *#constrained_parameters* tag). There are three types of models with constrained parameters:

- The first type is based on a rigid base polyhedron, which is modified by a number of constrained parameters which depend on a single variable $v1$. There is also a scaling parameter that is independent from $v1$. Modifying the value of $v1$ may change the shape of the model in a variety of ways. However, the average distance of the vertices from the center must be invariant under such modifications (otherwise calulations will be inaccurate, potentially in a subtle way). The parameters for this type of model typically work together to manipulate the angular components of the spherical vertex coordinates. This type is best suited for models with a cubic point group. Models of this type include the pyritohedral icosahedron and the elpasolite cuboctahedron.

- Some models have rigid components (e.g. vertically oriented regular polygons), but are not altogether rigid. This includes the fulvalene and biphenyl skeletons, among others. In these cases, the $\grave{h}$ parameter type is useful for constraining the vertical expansion of the rigid parts. However, to fully preserve the shape of such a rigid part while allowing it to change size, the value of the $\grave{h}$ parameter must be coupled with that of a $\grave{w}$ parameter. The parameter values are linear functions of the variable $v1$ in these cases.

- The third type of model is entirely managed by a number of variables $v1$, $v2$, etc. Each parameter tends to depend on more than one variable with this type of model, which makes them the most computationally expensive models defined in Polynator. There is currently only one built-in example; the truncated hexagonal trapezohedron, which requires constrained parameters to keep its pentagonal faces planar.

4.10 The Custom Model Relations Menu

The custom model relations window is accessible via *Settings* → *customize: model relations*. It allows the user to establish relations between models, especially custom models. Two models are related if they have the same number of vertices and the *child model*, having more degrees of freedom, can perfectly match any shape of the *parent model*. Model relations allow Polynator to pair atoms with model vertices more efficiently and more accurately. To establish a new relation, enter the names of child and parent model, as well as an index map, then press Enter or click on *add*. An index map is a short string of integers mapping the vertices of the parent model onto the child model. The indices of both parent and child

model can be viewed in the screens on the left of the window. Instead of a specific index map, it is possible to enter '*automatic', which prompts an algorithm to find valid maps automatically. However, it cannot be guaranteed that all valid maps, or any, will be found.

4.11 The Custom Graph Templates Menu

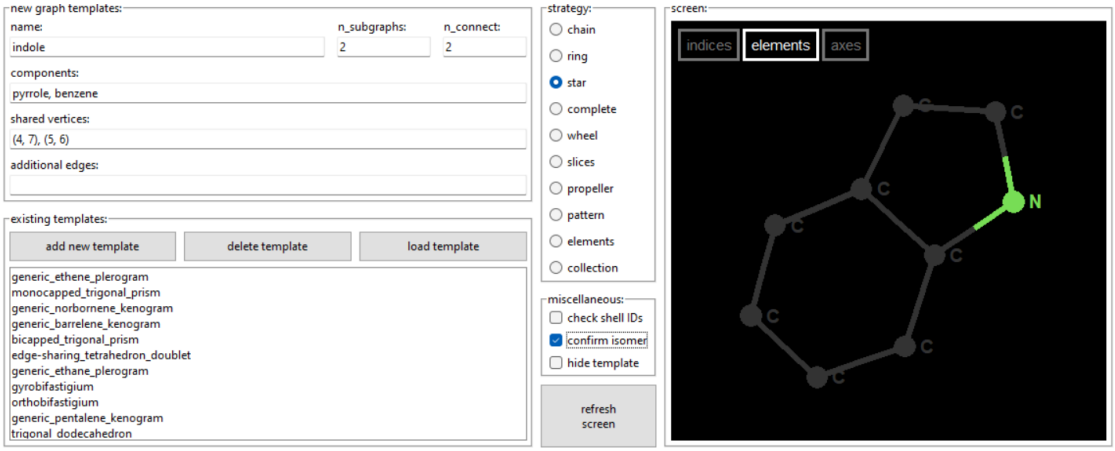

Figure 12: A template for the kenogram of an indole molecule is constructed from a pyrrole and a benzene component.

The custom graph templates window is accessible via *Settings* → *customize: graph templates*. It allows the user to construct templates for graphs of any size and complexity from simple components. These templates can then be used to trace graphs, as explained in section 4.5. It should be noted that being able to construct a graph is very different from devising an efficient algorithm for finding it as a subgraph of a potentially much larger supergraph (see section 3.7). This is the main reason why Polynator uses the somewhat clunky method of constructing graphs from smaller components according to specific rules instead of just letting the user construct graphs from scratch by connecting a number of vertices. The way a graph is constructed determines how it will be traced.

As mentioned above, graphs are constructed by connecting smaller component graphs. Which previously defined graph templates are used as components depends on the entry for *components*. Up to two different component types may be given. The strategy determines the shape of the underlying graph, which has component graphs as vertices. An edge between two such vertices is established

if and only if the corresponding component graphs share exactly the number of vertices (atoms) given as the *n_ connect* entry. The underlying graph has to conform to one of the primitive graph types shown in fig. 7. The *n_ subgraphs* entry determines the total number of component graphs. The *shared_ vertices* and *additional_ edges* entries make it possible to construct a unique and contiguous graph from the components. They each take a list of index tuples, with the indices referring to vertices of the component graphs. The indices in a *shared_ vertices* tuple must belong to different components. The corresponding vertices are merged to form a single vertex in the composite graph. The indices in a *additional_ edges* tuple must also belong to different components and the tuple must contain exactly two indices. The corresponding vertices are connected by an edge in the composite graph.

Other than the seven basic strategies based on the graph types shown in fig. 7, it is possible to define *pattern*-type instructions. A *pattern* is traced by joining any number of connected components. This makes it more open-ended than instructions based on the typically very precise basic strategies. A *collection* functions as a way to combine various tracing targets under one label. A collection may have any number of components and those components are not required to share any specific qualities.

The *elements* option makes it possible to specify elements for some or all of the vertices in a graph. This includes wildcards (see tab. 2). It is highly recommended to use this only on primitive graphs (e.g. star[4], ring[6]) and to construct more complex labelled graphs from those labelled primitives. False positives with regards to element positions may be traced unless *confirm isomer* is selected. For example, when tracing an imidazole kenogram based on a template without *confirm isomer*, pyrazole molecules may also turn up. This is also true for composite graphs based on labelled components. If *confirm isomer* was unselected in fig. 12, any isoindole molecules would also be found when tracing based on the displayed template.

5 Output Files

Output files for all evaluated objects can be generated via *Generate output files*. They will be put into a subdirectory of the folder holding the input files that were

evaluated. There are seven types of output files, the generation of which can be toggled in the *Settings* menu: two types of data tables, general output files, atom specific output files, two types of minimal .cif files and .log files.

5.1 Data Tables (.csv)

Two data tables may be generated as single .csv files which contain the most relevant information about all atom arrangements and fitted models, respectively. The real_atom_arrangements.csv file contains the composition, volume and number of atoms for each atom arrangement, as well as δ values referring to a the closest line, plane, or sphere containing all model vectors (see section 3.5). This file can be generated without fitting any models. The model_fits.csv file includes the distortion value, the volumes of the convex hulls of atom arrangements and model polyhedra, the averaged linear distance between paired atom and model vectors, the radial and angular portions of said distance and the free and constrained parameter values for each fitted model.

5.2 General Output Files (.out)

General output files are purely text-based. One such file is generated for each atom arrangement. It contains general information about the atom arrangement at the top and a section with information about each individual fit after that. These sections are separated from each other by wide horizontal lines. Each section is further divided into paragraphs.

The paragraph at the very top contains general information for the evaluated atom arrangement, such as the number of atoms, chemical composition and volume. The excentricity vector is the distance between the central atom and the centroid of all atoms. The three special δ values are explained in section 3.5. The second paragraph contains a Python dictionary with all input instructions to document the setup for this batch of results.

The first model-specific paragraph gives general information such as the name, point group, distortion value and volume of the model polyhedron. The third block holds some statistical information. This includes averages for the linear difference measures from the previous block. The standard deviation given for the length of the difference vector 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 δ value for a central projection of all atom and model vectors onto a unit sphere and for a cylindrical projection onto the model axis. The model constructor gives a Python dictionary of the model belts and their parameters. For more information on this entry, see sections 5.2 and 5.4. It might help to try out the custom model construction window. Lastly, the .out file lists the parameter values broken down into free and constrained parameters. These can be quite useful to measure e.g. the average height of a coordination environment or the torsion angle between the hexagonal rings in a biphenyl unit. However, it is important to be aware of prefactors! For instance, φ parameters often apply to two counterrotating belts, as marked by a minus sign before one of the 'phi' parameter names in the 'model constructor' block above. In that case, the value given for the φ parameter must be doubled to obtain the correct torsion angle between the two belts.

5.3 Atom Specific Output Files (.aso)

These are structured similarly to the .out files, but they contain information on the level of individual atoms and model vertices. Each file starts by listing the coordinates of all fitted atoms and their distance from the central atom (or from the centroid if a molecule or cage was evaluated). If Voronoi vertices were fitted, a similar list is generated for them.

The file continues with model-specific vectors and related information. All coordinates given are fractional (the same format you would find in a .cif file). Each atom vector comes with a vertical component (parallel to the model axis) and a horizontal component (orthogonal to the model axis). The 'angular difference' refers to the angle a given ligand vector is displaced from its model counterpart, from the perspective of the model center. 'Radial difference' means the difference between the distances of that atom vector and its model counterpart from the centroid. The 'angular difference' is just the angle between atom and model vertex from the perpective of the centroid. It is split into the spherical coordinate components 'phi difference' and 'theta difference'.

5.4 Minimal Coordinate Files (.cif)

The minimal .cif files Polynator creates are intended mostly for visualization and perhaps verification in an external program. They have space group $P1$ regardless of the original space group and contain only the atoms and model vertices involved in the respective fit. There are two versions of these files: fractional and cartesian. 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 have unconvenient placement and overlaps between translated coordination environments, which can be avoided by using the cartesian version.

5.5 Report Files (.log)

The .log file mostly tries to record if something went wrong during the computations. If you observe unexpected behaviour, it might be worthwhile to look at this file.

6 Navigating the Code

6.1 General Structure

Polynator is written in a loosely object-oriented style. While most of the code consists of class definitions, there are also some global constants (located at the beginning of the script) and independent functions (towards the end). There are 19 classes relevant to the backend and 25 other classes which manage the graphical user interface (GUI). Other than the modules imported from the standard library, all of the code is contained in the polynator_gui.py, polynator_main.py and polynator_model_constants.py files. This short guide will ignore the GUI and focus exclusively on the backend.

6.2 Classes

The MainProcess class orchestrates the backend at the highest level. It holds general information such as the user input and the blueprints used to construct models. Its run_full method starts the process by instantiating Structure objects. What was just the Structure class up to version 1.5 has now been split into the BasicStructure, CoordinationEnvironmentStructure and Molecu-

larStructure classes, the latter two inheriting from the former. Each of these processes the information read from one input .cif or .xyz file by the InputFileParser class. The create_supercell method performs symmetry operations on the raw atom vectors to generate atoms filling at least a $3 \times 3 \times 3$ supercell. The data for coordination environments or molecules is collected by the create_coordination_environments or create_molecules methods. Each set of atoms is then further processed by an AtomArrangement object instantiated by the MainProcess.create_atom_arrangements method. An AtomArrangement object contains information about a single molecule, coordination environment or Voronoi polyhedron and is responsible for administering models to it. After retrieving the appropriate ModelPrecursor containers from the MainProcess.dict_model_precursors dictionary, it starts to assign the atom vectors to the belts of the first model with its assign_to_belts method. Only after this assignment step is complete, a ModelFit object is instantiated by the AtomArrangement.run_assignment method. The actual fitting procedure is then carried out by this object.

The information about each atom generated by the Structure class is stored in Atom containers, which inherit from the abstract Vertex class. A Vertex has at least two attributes: an .index integer for identification and a .neighbors list to store connections with neighboring vertices. The BasicGraphTracer and SymmetricGraphTracer classes trace graphs (see section 3.7) with the help of the Subgraph class, which inherits from the Vertex class and stores a number of Atoms. To make the process of connecting Vertices more efficient, the CellSubdivision class implements a spatial hash, which is utilized by the Structure and GraphTracer classes.

Voronoi polyhedra are constructed by the VoronoiDiagram class and its subordinate DelaunayTetrahedron and VoronoiFace classes using a flip-based incremental insertion algorithm for the construction of the dual Delaunay tetrahedralization. Convex hulls are constructed by the ConvexHull class and its subordinate ConvexFace class via a divide and conquer algorithm. Minimal bounding spheres are constructed by the get_minimal_bounding_sphere function.

6.3 Global Constants

A notable constant is the DICT_MODEL_BLUEPRINTS, which has model names as keys and tuples of dictionaries as values. Each tuple contains the information necessary to construct and manipulate the model vertices. This is split up into belt dictionaries, each holding information such as the number of atoms 'n', initial height and width 'h_init' and 'w_init', as well as a number of free or constrained parameters. DICT_MODEL_BITS_N_BOBS holds auxiliary information about each model, such as tags, point group symmetry and the number of variables. It notably also contains the information required for the computation of constrained parameters. It is found in the entries for 'parameter_bundles' and 'bundle_domains'. DICT_MODEL_EDGES contains tuples encoding the edges between model vertices

DICT_MODEL_NAMES_BY_GRAPH_ID and DICT_MODEL_GRAPH_AUTOMORPHISMS contain the necessary information to assign model vertices using the graph theoretical method outlined in section 3.2.3. DICT_SYMMETRY_RELATIONS encodes relationships between models. This means mostly symmetry relations, but also other relations between topologically equivalent models with different degrees of freedom (e.g. between a rigid and a dynamic trigonal prism). It allows a derivative model to inherit its belt assignment from a previously fitted parent model (not to be confused with class inheritance). This is encoded in the encapsulated lists of lists of integers. Each integer is the index of an atom vector in the flattened belts_real attribute of the parent ModelFit object.

DICT_GRAPH_TEMPLATES holds information utilized by the Graph classes to construct and trace various molecular subunits and cages in a larger network of atoms. Each set of instructions is encoded in a dictionary. Each such dictionary (also found as the target_dict attribute of a Graph object) has entries specifying the number of atoms and vertices, as well as the strategy (see section 3.7). If the traced graph is based on a combination of component graphs, the dictionary will include additional entries. The component_clusters entry is always a tuple of two other graph names (except if the strategy is 'collection').

6.4 Model Parameters

All parameter types other than 'sc' have a version with an '_init' suffix, which signals a fixed parameter, as opposed to a free or dependent one. These '_init' values are applied when the model prototypes are constructed by the ModelPrecursor.construct_prototype function. They are useful for the construction of many rigid models, including the rigid base models for dynamic models with constrained parameters. The modulating parameters described in section 3.4 go along with suffixes '_frq' and '_off' for the frequency and offset of the sinusoidal wave function, respectively. If a model requires more than one modulated parameter of the same type in the same belt, the type name of the second parameter, as well as its '_frq' and '_off' supplements are prefixed with one or more additional leading '∼' characters. As discussed in section 3.4, some models require multiple parameters to be constrained as a function of a single variable. If a blueprint contains one or more such 'parameter_bundles', they are defined in the dict_info section, along with a 'bundle_domains' dictionary, which defines the domain of each bundle variable as a tuple with the minimal, maximal and starting value, respectively.

References

[1] D. Waroquiers, J. George, M. Horton, S. Schenk, K. A. Persson, G.-M. Rignanese, X. Gonze, G. Hautier, *Acta Crystallogr. B*, **2020**, 76, 683–695.

[2] M. O'Keeffe, *Acta Crystallogr. A*, **1979**, 35, 772–775.

[3] H. Zabrodsky, S. Peleg, D. Avnir, *J. Am. Chem. Soc.*, **1992**, 114, 7843–7851.

[4] M. Pinsky, D. Avnir, *Inorg. Chem.*, **1998**, 37, 5575.

[5] W. Kabsch, *Acta Crystallogr. A*, **1976**, 30, 513.