crystract

Introduction

This vignette demonstrates the core workflow for using the `crystract` package. The package is designed for the **batch processing** of Crystallographic Information Files (.cif) to extract structural and chemical information from the repeating atomic arrangement that defines a crystal. It calculates key geometric properties like bond lengths and angles and compiles the results into a structured format suitable for large-scale data analysis.

We will first demonstrate the main batch-processing function, `analyze_cif_files`. Then, to illustrate how it works, we will walk through the analysis of a single file step-by-step, explaining the underlying functions, the crystallographic principles, and the mathematical formulas they employ.

Setup: Loading the Package

First, we load the `crystract` package.

```r
library(crystract)
```

1. The Core `crystract` Workflow

The primary goal of `crystract` is to automate the analysis of many CIF files at once. While the package provides granular functions for each step of the crystallographic analysis, the most common use case is to leverage the main wrapper function for an end-to-end pipeline.

1.1 The Full Pipeline for Batch Processing

The `analyze_cif_files()` function is the cornerstone of the package. It performs the entire sequence of operations—from reading files to calculating bond angles with error propagation—for one or more files. It takes a vector of file paths and returns a single `data.table` where each row corresponds to a processed CIF file. Complex results (like coordinates, distances, and angles) are stored in list-columns for easy access and analysis.

This is the recommended function for batch processing. The code block below shows a commented-out example of how you would load all CIF files from the package's example data directory. For this demonstration, we will proceed by analyzing just a single example file.

```r
# Find the path to the single example CIF file included in the package
cif_path <- system.file("extdata", "ICSD422.cif", package = "crystract")

# --- Example for batch processing a directory ---
# To load all CIF files from the package's example directory, you would
# uncomment and run this:
# extdata_dir <- system.file("extdata", package = "crystract")
# all_cif_paths <- list.files(path = extdata_dir, pattern = "\\.cif$", full.names = TRUE)
# analysis_results_batch <- analyze_cif_files(all_cif_paths)

# For this vignette, we will run the pipeline on just our single example file
analysis_results <- analyze_cif_files(cif_path)

# Let's inspect the structure of the output table.
# It's a single row containing all our results in nested data.tables.
str(analysis_results, max.level = 2)
#> Classes 'data.table' and 'data.frame':   1 obs. of  17 variables:
#>  $ file_name          : chr "ICSD422.cif"
#>  $ database_code      : chr "ICSD 422"
#>  $ chemical_formula   : chr "Si1 Sr2"
#>  $ structure_type     : chr "TiNiSi#MgSrSi"
#>  $ space_group_name   : chr "P n m a"
#>  $ space_group_number : chr "62"
#>  $ unit_cell_metrics  :List of 1
#>   ..$ :Classes 'data.table' and 'data.frame':    1 obs. of  12 variables:
#>   .. ..- attr(*, ".internal.selfref")=<externalptr>
#>  $ atomic_coordinates :List of 1
#>   ..$ :Classes 'data.table' and 'data.frame':    3 obs. of  11 variables:
#>   .. ..- attr(*, ".internal.selfref")=<externalptr>
#>  $ symmetry_operations:List of 1
#>   ..$ :Classes 'data.table' and 'data.frame':    8 obs. of  3 variables:
#>   .. ..- attr(*, ".internal.selfref")=<externalptr>
#>  $ transformed_coords :List of 1
#>   ..$ :Classes 'data.table' and 'data.frame':    12 obs. of  4 variables:
#>   .. ..- attr(*, ".internal.selfref")=<externalptr>
#>  $ expanded_coords    :List of 1
#>   ..$ :Classes 'data.table' and 'data.frame':    324 obs. of  4 variables:
#>   .. ..- attr(*, ".internal.selfref")=<externalptr>
#>  $ distances          :List of 1
#>   ..$ :Classes 'data.table' and 'data.frame':    969 obs. of  6 variables:
#>   .. ..- attr(*, ".internal.selfref")=<externalptr>
#>  $ bonded_pairs       :List of 1
#>   ..$ :Classes 'data.table' and 'data.frame':    14 obs. of  9 variables:
#>   .. ..- attr(*, ".internal.selfref")=<externalptr>
#>   .. ..- attr(*, "sorted")= chr [1:3] "Atom1" "Atom2" "Distance"
#>  $ brunner_pairs      :List of 1
#>   ..$ :Classes 'data.table' and 'data.frame':    24 obs. of  6 variables:
#>   .. ..- attr(*, ".internal.selfref")=<externalptr>
#>  $ hoppe_pairs        :List of 1
#>   ..$ :Classes 'data.table' and 'data.frame':    22 obs. of  3 variables:
#>   .. ..- attr(*, ".internal.selfref")=<externalptr>
#>  $ neighbor_counts    :List of 1
#>   ..$ :Classes 'data.table' and 'data.frame':    3 obs. of  2 variables:
#>   .. ..- attr(*, ".internal.selfref")=<externalptr>
#>  $ bond_angles        :List of 1
#>   ..$ :Classes 'data.table' and 'data.frame':    30 obs. of  5 variables:
#>   .. ..- attr(*, ".internal.selfref")=<externalptr>
#>   .. ..- attr(*, "sorted")= chr [1:3] "CentralAtom" "Neighbor1" "Neighbor2"
#>  - attr(*, ".internal.selfref")=<externalptr>
```

As shown in the structure output, the result is a tidy `data.table` containing all extracted and calculated information. We can easily access the nested data frames for further analysis. Note that the output includes results from multiple bonding algorithms, such as `bonded_pairs` (Minimum Distance), `brunner_pairs`, and `hoppe_pairs`, which are described later.

To get the final `bonded_pairs` table (which includes propagated errors):

```r
# The result is a list-column, so we access the element with [[1]]
final_bonds <- analysis_results$bonded_pairs[[1]]

print(head(final_bonds))
#> Key: <Atom1, Atom2, Distance>
#>     Atom1         Atom2 Distance  DeltaX DeltaY  DeltaZ     dcut     dmin
#>    <char>        <fctr>    <num>   <num>  <num>   <num>    <num>    <num>
#> 1:    Si1 Sr1_4_0_-1_-1 3.184477 -0.0932    0.5  0.1797 3.479899 3.163544
#> 2:    Si1  Sr1_4_0_0_-1 3.184477 -0.0932   -0.5  0.1797 3.479899 3.163544
#> 3:    Si1  Sr2_1_0_0_-1 3.261366  0.2347    0.0  0.2776 3.479899 3.163544
#> 4:    Si1 Sr2_3_-1_-1_0 3.465249  0.2731    0.5 -0.0720 3.479899 3.163544
#> 5:    Si1  Sr2_3_-1_0_0 3.465249  0.2731   -0.5 -0.0720 3.479899 3.163544
#> 6:    Si1   Sr1_1_0_0_0 3.163544  0.1010    0.0 -0.3203 3.479899 3.163544
#>    DistanceError
#>            <num>
#> 1:   0.008313329
#> 2:   0.008313329
#> 3:   0.014072755
#> 4:   0.009293087
#> 5:   0.009293087
#> 6:   0.014160750
```

1.2 A Step-by-Step Walkthrough

To understand what `analyze_cif_files()` does under the hood, this section breaks down the process. We will use a single CIF file to demonstrate each function individually, explaining the crystallographic concepts and the structure of the output at each stage.

1.2.1 Loading CIF Data We use the package's `read_cif_files()` function to load data into memory. For this demonstration, we use an example CIF file for Strontium Silicide (Sr_2Si) from the Inorganic Crystal Structure Database (ICSD), specifically entry 422. This entry can be found online here: ICSD 422.

```r
# The path was defined in the previous section:
cif_data_list <- read_cif_files(cif_path)

# We'll work with the content of the first file
cif_content <- cif_data_list[[1]]

# Let's look at the first few lines of the raw data
knitr::kable(
  head(cif_content),
  caption = "First 6 lines of the raw CIF data."
)
```

Table 1: First 6 lines of the raw CIF data.

V1
#(C) 2023 by FIZ Karlsruhe - Leibniz Institute for Information Infrastructure. All rights reserved.
data_422-ICSD
_database_code_ICSD 422
_audit_creation_date 1980-01-01
_audit_update_record 2000-07-15
_chemical_name_common 'Strontium silicide (2/1)'

1.2.2 Extracting Metadata and Unit Cell Parameters A crystal structure is defined by its **unit cell**, the smallest repeating parallelepiped that can be used to build the entire crystal through translation, and the arrangement of atoms within it. We first extract this high-level information.

```
database_code <- extract_database_code(cif_content)
chemical_formula <- extract_chemical_formula(cif_content)
space_group_name <- extract_space_group_name(cif_content)
space_group_number <- extract_space_group_number(cif_content)

cat("Database Code:", database_code, "\n")
#> Database Code: ICSD 422
cat("Chemical Formula:", chemical_formula, "\n")
#> Chemical Formula: Si1 Sr2
cat("Space Group:", space_group_name, "(No.", space_group_number, ")\n")
#> Space Group: P n m a (No. 62 )
```

Next, `extract_unit_cell_metrics()` extracts the six parameters that define the shape and size of the unit cell: the lengths of its three axes (a, b, c) and the angles between them (α, β, γ). Their experimental uncertainties (a_{err}, etc.) are also extracted.

```
unit_cell_metrics <- extract_unit_cell_metrics(cif_content)
print(unit_cell_metrics)
#>    _cell_length_a _cell_length_b _cell_length_c _cell_angle_alpha
#>             <num>          <num>          <num>             <num>
#> 1:           8.11           5.15           9.54                90
#>    _cell_angle_beta _cell_angle_gamma _cell_length_a_error _cell_length_b_error
#>               <num>             <num>                <num>                <num>
#> 1:               90                90                   NA                   NA
#>    _cell_length_c_error _cell_angle_alpha_error _cell_angle_beta_error
#>                   <num>                   <num>                  <num>
#> 1:                   NA                      NA                     NA
#>    _cell_angle_gamma_error
#>                      <num>
#> 1:                      NA
```

1.2.3 Extracting Atomic and Symmetry Data Instead of listing every atom in the unit cell, a CIF file efficiently describes the structure using only the **asymmetric unit**: the smallest set of unique atoms. All other atoms in the unit cell can be generated by applying the crystal's **symmetry operations** to this unique set.

```r
# Extract the coordinates of the unique atoms in the asymmetric unit
atomic_coordinates <- extract_atomic_coordinates(cif_content)
print("Asymmetric Atomic Coordinates:")
#> [1] "Asymmetric Atomic Coordinates:"
print(atomic_coordinates)
#>     Label WyckoffSymbol WyckoffMultiplicity Occupancy OccupancyError    x_a
#>    <char>        <char>               <num>     <num>          <num>  <num>
#> 1:    Sr1             c                   4         1             NA 0.6529
#> 2:    Sr2             c                   4         1             NA 0.5192
#> 3:    Si1             c                   4         1             NA 0.2539
#>      y_b    z_c x_error y_error z_error
#>    <num>  <num>   <num>   <num>   <num>
#> 1:  0.25 0.0769  0.0006      NA  0.0005
#> 2:  0.25 0.6748  0.0006      NA  0.0005
#> 3:  0.25 0.1028  0.0016      NA  0.0014

# Extract the symmetry operations
symmetry_operations <- extract_symmetry_operations(cif_content)
print("Symmetry Operations (first 6 of 8):")
#> [1] "Symmetry Operations (first 6 of 8):"
print(head(symmetry_operations))
#>          x      y      z
#>    <char> <char> <char>
#> 1:  x+1/2      y -z+1/2
#> 2:      x -y+1/2      z
#> 3: -x+1/2  y+1/2  z+1/2
#> 4:     -x     -y     -z
#> 5: -x+1/2     -y  z+1/2
#> 6:     -x  y+1/2     -z
```

Understanding the `atomic_coordinates` Table:

- `Label`: This is a unique identifier for each atom in the asymmetric unit. The number (e.g., `Sr1`, `Sr2`) distinguishes atoms of the same element that occupy crystallographically distinct sites (i.e., they have different local environments and cannot be transformed into one another by a symmetry operation).
- `x_a`, `y_b`, `z_c`: These are the **fractional coordinates** of the atom. They describe the atom's position as a fraction of the length of the unit cell axes a, b, and c, respectively. A value of $(0.5, 0.5, 0.5)$ represents the exact center of the unit cell.
- `x_error`, `y_error`, `z_error`: The experimental standard uncertainties associated with each coordinate.

1.2.4 Generating the Full Crystal Structure Now we use the asymmetric atoms and symmetry operations to computationally build the full crystal structure. This is a two-step process.

Formula Context: Symmetry Operations

A symmetry operation is an affine transformation that maps an initial fractional coordinate (x, y, z) to a new coordinate (x', y', z'). It consists of a rotation/reflection matrix $\mathbf{W}$ and a translation vector $\mathbf{w}$.

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} \quad \text{or} \quad \mathbf{x}' = \mathbf{W}\mathbf{x} + \mathbf{w}$$

The `apply_symmetry_operations` function applies all of the crystal's symmetry operations to each atom in the asymmetric unit, generating the complete set of atoms within the primary unit cell.

Formula Context: Supercell Expansion

To find all nearest neighbors of an atom, we must also consider atoms in adjacent unit cells. The `expand_transformed_coords` function generates a **supercell** (in this case, 3x3x3) by translating the primary unit cell atoms by integer vectors (i, j, k), where i, j, k each range from -1 to 1. An atom at fractional coordinate (x, y, z) generates new coordinates:

$$(x_{exp}, y_{exp}, z_{exp}) = (x + i, y + j, z + k)$$

```
# Apply symmetry to generate all atoms in the primary unit cell
transformed_coords <- apply_symmetry_operations(atomic_coordinates, symmetry_operations)
print("Unique atoms in full unit cell (first 6 of 12):")
#> [1] "Unique atoms in full unit cell (first 6 of 12):"
print(head(transformed_coords))
#>     Label    x_a   y_b    z_c
#>    <char>  <num> <num>  <num>
#> 1:  Sr1_1 0.1529  0.25 0.4231
#> 2:  Sr1_2 0.6529  0.25 0.0769
#> 3:  Sr1_3 0.8471  0.75 0.5769
#> 4:  Sr1_4 0.3471  0.75 0.9231
#> 5:  Sr2_1 0.0192  0.25 0.8252
#> 6:  Sr2_2 0.5192  0.25 0.6748

# Expand into a 3x3x3 supercell for neighbor calculations
expanded_coords <- expand_transformed_coords(transformed_coords)
print("Atoms in supercell (first 6 of 324):")
#> [1] "Atoms in supercell (first 6 of 324):"
print(head(expanded_coords))
#>             Label     x_a   y_b     z_c
#>            <char>   <num> <num>   <num>
#> 1: Sr1_1_-1_-1_-1 -0.8471 -0.75 -0.5769
#> 2: Sr1_2_-1_-1_-1 -0.3471 -0.75 -0.9231
#> 3: Sr1_3_-1_-1_-1 -0.1529 -0.25 -0.4231
#> 4: Sr1_4_-1_-1_-1 -0.6529 -0.25 -0.0769
#> 5: Sr2_1_-1_-1_-1 -0.9808 -0.75 -0.1748
#> 6: Sr2_2_-1_-1_-1 -0.4808 -0.75 -0.3252
```

Understanding the Generated Atom Labels:

- **`transformed_coords` Label (e.g., `Sr1_1`)**: The label `Sr1_1` means "the atom generated by applying the 1st symmetry operation to the asymmetric atom `Sr1`". `Sr1_2` comes from the 2nd symmetry operation, and so on.
- **`expanded_coords` Label (e.g., `Sr1_1_-1_-1_-1`)**: This label provides the full history. `Sr1_1_-1_-1_-1` is the atom that was generated from asymmetric atom `Sr1` via the 1st symmetry operation, and then translated by $i = -1, j = -1, k = -1$ (i.e., moved one unit cell length in the negative direction along all three axes a, b, and c).

1.2.5 Calculating Interatomic Distances **Formula Context: Interatomic Distance in a Triclinic System**

Because unit cell axes are not always orthogonal, the simple Pythagorean theorem is insufficient. The distance d between two atoms at fractional coordinates (x_{f1}, y_{f1}, z_{f1}) and (x_{f2}, y_{f2}, z_{f2}) requires the full crystallographic distance formula, which accounts for the unit cell metric tensor:

$$d = \begin{pmatrix} a^2(x_{f1} - x_{f2})^2 + b^2(y_{f1} - y_{f2})^2 + c^2(z_{f1} - z_{f2})^2 \\ + 2ab(x_{f1} - x_{f2})(y_{f1} - y_{f2}) \cos\gamma \\ + 2ac(x_{f1} - x_{f2})(z_{f1} - z_{f2}) \cos\beta \\ + 2bc(y_{f1} - y_{f2})(z_{f1} - z_{f2}) \cos\alpha \end{pmatrix}^{1/2}$$

The `calculate_distances` function computes the distances from each central atom (from the asymmetric unit) to all other atoms in the generated supercell.

```r
distances <- calculate_distances(atomic_coordinates, expanded_coords, unit_cell_metrics)
print("Calculated Distances (shortest 6):")
#> [1] "Calculated Distances (shortest 6):"
print(head(distances[order(Distance)]))
#>     Atom1          Atom2 Distance  DeltaX DeltaY  DeltaZ
#>    <fctr>         <fctr>    <num>   <num>  <num>   <num>
#> 1:    Si1    Sr1_1_0_0_0 3.163544  0.1010    0.0 -0.3203
#> 2:    Sr1    Si1_1_0_0_0 3.163544 -0.1010    0.0 -0.3203
#> 3:    Sr1 Si1_4_0_-1_-1 3.184477 -0.0932    0.5  0.1797
#> 4:    Sr1  Si1_4_0_0_-1 3.184477 -0.0932   -0.5  0.1797
#> 5:    Si1 Sr1_4_0_-1_-1 3.184477 -0.0932    0.5  0.1797
#> 6:    Si1  Sr1_4_0_0_-1 3.184477 -0.0932   -0.5  0.1797
```

Understanding the `distances` Table Columns:

- `Atom1`: The central atom from the original asymmetric unit (e.g., `Sr1`, `Si1`).
- `Atom2`: The neighboring atom from the supercell, identified by its fully descriptive label.
- `Distance`: The calculated interatomic distance in Angstroms (Å).
- `DeltaX`, `DeltaY`, `DeltaZ`: The vector components $(x_{f1} - x_{f2})$, $(y_{f1} - y_{f2})$, etc., in fractional coordinates.

1.2.6 Identifying Bonds and Neighbors

Formula Context: Bonding and Coordination Number

Identifying which atoms are "bonded" is key to determining the **coordination number (CN)**, the count of nearest neighbors. `crystract` implements several methods for this task:

- **Minimum Distance Method**: The default method used here. It defines a custom distance cutoff for each central atom i:

$$d_i^{\text{cut}} = (1 + \delta) d_i^{\text{min}}$$

  Here, d_i^{min} is the shortest distance from atom i to any other atom, and δ is a tolerance parameter (default is 0.1). It is robust and requires no element-specific presets.
- **Brunner's Method**: Uses modified ionic radii to determine cutoffs.
- **Hoppe's Method (ECoN)**: Weights neighbor contributions by distance, yielding a continuous, non-integer CN.

The `minimum_distance` function filters the `distances` table to identify bonded pairs. `calculate_neighbor_counts` then summarizes these to find the integer CN for each central atom.

```r
# Identify bonded pairs using a tolerance of 10%
bonded_pairs <- minimum_distance(distances, delta = 0.1)
print("Bonded Pairs (first 6):")
#> [1] "Bonded Pairs (first 6):"
print(head(bonded_pairs))
```

```
#>      Atom1          Atom2 Distance  DeltaX DeltaY  DeltaZ     dcut     dmin
#>     <fctr>         <fctr>    <num>   <num>  <num>   <num>    <num>    <num>
#> 1:     Sr1 Si1_4_0_-1_-1 3.184477 -0.0932    0.5  0.1797 3.479899 3.163544
#> 2:     Sr1  Si1_4_0_0_-1 3.184477 -0.0932   -0.5  0.1797 3.479899 3.163544
#> 3:     Sr1   Si1_1_0_0_0 3.163544 -0.1010    0.0 -0.3203 3.479899 3.163544
#> 4:     Sr1   Si1_2_0_0_0 3.245310  0.3990    0.0 -0.0259 3.479899 3.163544
#> 5:     Sr2  Si1_3_0_-1_0 3.465249  0.2731    0.5  0.0720 3.587503 3.261366
#> 6:     Sr2   Si1_1_0_0_0 3.261366 -0.2347    0.0  0.2776 3.587503 3.261366

# Calculate neighbor counts based on the bonded pairs
neighbor_counts <- calculate_neighbor_counts(bonded_pairs)
print("Neighbor Counts:")
#> [1] "Neighbor Counts:"
print(neighbor_counts)
#>       Atom NeighborCount
#>     <fctr>         <int>
#> 1:     Sr1             4
#> 2:     Sr2             3
#> 3:     Si1             7
```

1.2.7 Calculating Bond Angles Formula Context: Bond Angle Calculation

To calculate a bond angle A-B-C (with B at the vertex), the fractional coordinates must first be converted to an orthogonal Cartesian system.

1. **Fractional to Cartesian Conversion:** A fractional coordinate ($\mathbf{x}_f$) is converted to a Cartesian coordinate ($\mathbf{x}_c$) via a transformation matrix $\mathbf{M}$:

$$\begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} = \mathbf{M} \cdot \begin{pmatrix} x_f \\ y_f \\ z_f \end{pmatrix} \quad \text{where} \quad \mathbf{M} = \begin{pmatrix} a & b\cos\gamma & c\cos\beta \\ 0 & b\sin\gamma & \frac{c(\cos\alpha - \cos\beta\cos\gamma)}{\sin\gamma} \\ 0 & 0 & \frac{V}{ab\sin\gamma} \end{pmatrix}$$

   where V is the unit cell volume.

2. **Angle via Dot Product:** With atoms in Cartesian space, the angle θ between vectors $\vec{u}$ (from B to A) and $\vec{v}$ (from B to C) is:

$$\theta = \arccos\left(\frac{\vec{u}\cdot\vec{v}}{|\vec{u}||\vec{v}|}\right)$$

The `calculate_angles` function implements this for all possible bond angles around each central atom.

```
bond_angles <- calculate_angles(
  bonded_pairs,
  atomic_coordinates,
  expanded_coords,
  unit_cell_metrics
)
print("Calculated Bond Angles (first 6):")
#> [1] "Calculated Bond Angles (first 6):"
print(head(bond_angles))
#>    CentralAtom     Neighbor1     Neighbor2     Angle
#>         <char>        <fctr>        <fctr>     <num>
#> 1:         Si1 Sr1_4_0_-1_-1  Sr1_4_0_0_-1 107.92071
#> 2:          Si1 Sr1_4_0_-1_-1  Sr2_1_0_0_-1  72.62529
#> 3:          Si1 Sr1_4_0_-1_-1 Sr2_3_-1_-1_0  69.97350
#> 4:          Si1 Sr1_4_0_-1_-1  Sr2_3_-1_0_0 149.23688
#> 5:          Si1 Sr1_4_0_-1_-1   Sr1_1_0_0_0 125.55190
#> 6:          Si1 Sr1_4_0_-1_-1   Sr1_2_0_0_0  73.87978
```

1.2.8 Error Propagation Finally, `crystract` propagates the experimental uncertainties from cell parameters and atomic coordinates to the calculated distances and angles.

Formula Context: Error Propagation

The uncertainty, σ_f, in a calculated value f that depends on several variables p_i (each with uncertainty σ_{p_i}) is found using the sum of squares of the partial derivatives. A critical simplification made here is assuming the input variables (cell parameters, atomic coordinates) are uncorrelated. The general formula for the variance (σ_f^2) is:

$$\sigma_f^2 = \sum_i \left(\frac{\partial f}{\partial p_i} \sigma_{p_i} \right)^2$$

Uncertainty in Interatomic Distance (σ_d) The uncertainty in the distance, σ_d, is found by applying the general formula to the distance d. The input parameters p_i are the 12 variables that define the distance: $\{a, b, c, \alpha, \beta, \gamma, x_{f1}, y_{f1}, z_{f1}, x_{f2}, y_{f2}, z_{f2}\}$. Letting $\Delta x = x_{f1} - x_{f2}$, etc., the partial derivatives are, for example:

$$\frac{\partial d}{\partial a} = \frac{1}{2d} \left(2a(\Delta x)^2 + 2b(\Delta x)(\Delta y) \cos \gamma + 2c(\Delta x)(\Delta z) \cos \beta \right)$$

Uncertainty in Bond Angle (σ_θ) Propagating error to the bond angle θ is a more complex, multi-step process involving propagating initial errors first to Cartesian coordinates, and then from those Cartesian uncertainties to the final angle via the chain rule.

```
# Propagate errors for interatomic distances
bonded_pairs_with_error <- propagate_distance_error(
  bonded_pairs,
  atomic_coordinates,
  unit_cell_metrics
)
print("Bonded Pairs with Distance Error (first 6):")
#> [1] "Bonded Pairs with Distance Error (first 6):"
print(head(bonded_pairs_with_error))
#> Key: <Atom1, Atom2, Distance>
#>     Atom1          Atom2 Distance  DeltaX DeltaY  DeltaZ     dcut     dmin
#>    <char>         <fctr>    <num>   <num>  <num>   <num>    <num>    <num>
#> 1:    Si1 Sr1_4_0_-1_-1 3.184477 -0.0932    0.5  0.1797 3.479899 3.163544
#> 2:    Si1  Sr1_4_0_0_-1 3.184477 -0.0932   -0.5  0.1797 3.479899 3.163544
#> 3:    Si1  Sr2_1_0_0_-1 3.261366  0.2347    0.0  0.2776 3.479899 3.163544
#> 4:    Si1 Sr2_3_-1_-1_0 3.465249  0.2731    0.5 -0.0720 3.479899 3.163544
#> 5:    Si1  Sr2_3_-1_0_0 3.465249  0.2731   -0.5 -0.0720 3.479899 3.163544
#> 6:    Si1   Sr1_1_0_0_0 3.163544  0.1010    0.0 -0.3203 3.479899 3.163544
#>    DistanceError
#>            <num>
#> 1:   0.008313329
#> 2:   0.008313329
#> 3:   0.014072755
#> 4:   0.009293087
#> 5:   0.009293087
#> 6:   0.014160750

# Propagate errors for bond angles
bond_angles_with_error <- propagate_angle_error(
  bond_angles,
  atomic_coordinates,
  expanded_coords,
  unit_cell_metrics
)
print("Bond Angles with Angle Error (first 6):")
#> [1] "Bond Angles with Angle Error (first 6):"
print(head(bond_angles_with_error))
#> Key: <CentralAtom, Neighbor1, Neighbor2>
#>    CentralAtom     Neighbor1     Neighbor2     Angle AngleError
#>         <char>        <fctr>        <fctr>     <num>      <num>
#> 1:         Si1 Sr1_4_0_-1_-1  Sr1_4_0_0_-1 107.92071  0.3991814
#> 2:         Si1 Sr1_4_0_-1_-1  Sr2_1_0_0_-1  72.62529  0.2625336
#> 3:         Si1 Sr1_4_0_-1_-1 Sr2_3_-1_-1_0  69.97350  0.1320523
#> 4:         Si1 Sr1_4_0_-1_-1  Sr2_3_-1_0_0 149.23688  0.4492550
#> 5:         Si1 Sr1_4_0_-1_-1   Sr1_1_0_0_0 125.55190  0.2121372
#> 6:         Si1 Sr1_4_0_-1_-1   Sr1_2_0_0_0  73.87978  0.2610475
```

2. Tools for Post-Processing and Analysis

After running the main analysis pipeline, `crystract` provides several helper functions to filter and refine the results, allowing you to isolate data based on chemical identity or key crystallographic properties.

2.1 Filtering by Chemical Identity

The `filter_atoms_by_symbol()` function provides an interactive way to filter results (like bond or angle tables) to focus on the coordination environment around a specific type of element.

```
# In an interactive R session, you would run this:
filtered_bonds <- filter_atoms_by_symbol(
  data_table = bonded_pairs_with_error,
  atom_col = "Atom1" # Filter based on the central atom
)
```

If you were to type `Si` and press Enter, the function would return a new `data.table` containing only the rows where the central atom (`Atom1`) is Silicon.

```
# Manually simulate the user typing "Si" at the prompt
user_input <- "Si"
symbols_to_keep <- trimws(strsplit(user_input, ",")[[1]])
patterns <- sapply(symbols_to_keep, function(sym) paste0("(^", sym, "$)|(^", sym, "[^A-Za-z])"))
full_pattern <- paste(patterns, collapse = "|")
filtered_bonds_by_symbol <- bonded_pairs_with_error[grepl(full_pattern, get("Atom1"))]

print("Original table contains bonds centered on both Sr and Si atoms:")
```

```
#> [1] "Original table contains bonds centered on both Sr and Si atoms:"
print(bonded_pairs_with_error$Atom1)
#>  [1] "Si1" "Si1" "Si1" "Si1" "Si1" "Si1" "Si1" "Sr1" "Sr1" "Sr1" "Sr1" "Sr2"
#> [13] "Sr2" "Sr2"
cat("\n")
print("Filtered table now only contains bonds centered on 'Si1':")
#> [1] "Filtered table now only contains bonds centered on 'Si1':"
print(filtered_bonds_by_symbol$Atom1)
#> [1] "Si1" "Si1" "Si1" "Si1" "Si1" "Si1" "Si1"
```

2.2 Filtering by Wyckoff Position

For many crystallographic studies, it is crucial to analyze atoms based on their specific site symmetry, described by their **Wyckoff position** (e.g., "2a", "6c"). The `filter_by_wyckoff_symbol()` function is designed for this purpose.

```
# 1. In our example, all asymmetric atoms occupy the Wyckoff site 'c' with multiplicity 4 ("4c").
print("Atomic coordinates showing Wyckoff information:")
#> [1] "Atomic coordinates showing Wyckoff information:"
print(atomic_coordinates[, .(Label, WyckoffSymbol, WyckoffMultiplicity)])
#>     Label WyckoffSymbol WyckoffMultiplicity
#>    <char>        <char>               <num>
#> 1:    Sr1             c                   4
#> 2:    Sr2             c                   4
#> 3:    Si1             c                   4
cat("\n")

# 2. Filter bonds where the central atom is on the "4c" Wyckoff site.
bonds_from_4c_site <- filter_by_wyckoff_symbol(
  data_table = bonded_pairs_with_error,
  atomic_coordinates = atomic_coordinates,
  atom_col = "Atom1",
  wyckoff_symbols = "4c"
)

print(paste("Number of rows in original bond table:", nrow(bonded_pairs_with_error)))
#> [1] "Number of rows in original bond table: 14"
print(paste("Number of rows after filtering for site '4c':", nrow(bonds_from_4c_site)))
#> [1] "Number of rows after filtering for site '4c': 14"
```

2.3 Filtering Ghost Distances Using Atomic Radii

A common issue in crystallography is **site-occupancy disorder**, where a single crystallographic site is statistically co-occupied by different atoms (e.g., a mix of small Boron and larger Silicon atoms). Standard calculations can produce physically meaningless, short "ghost" distances between atoms that never exist in that configuration simultaneously.

The `filter_ghost_distances()` function cleans the distance table by removing these artifacts. It uses a built-in table of covalent radii to establish a minimum plausible bond length. Any calculated distance falling below this physical threshold (multiplied by a tolerance factor) is filtered out.

```
# We use the full, unfiltered distance table for this demonstration.
# The function returns a list with two tables: 'kept' and 'removed'.
# A distance is kept if: Distance > (Radius1 + Radius2) * tolerance
filtered_result <- filter_ghost_distances(
    distances = distances,
    atomic_coordinates = atomic_coordinates,
    tolerance = 0.4 # Default tolerance
)

kept_distances <- filtered_result$kept
removed_distances <- filtered_result$removed

cat("Total distances calculated:", nrow(distances), "\n")
#> Total distances calculated: 969
cat("Distances kept after filtering:", nrow(kept_distances), "\n")
#> Distances kept after filtering: 969
cat("Ghost distances removed:", nrow(removed_distances), "\n\n")
#> Ghost distances removed: 0

# For a well-ordered structure like Sr2Si, no ghost bonds are expected.
# The 'removed' table should be empty, confirming the filter's behavior.
print("Removed ghost distances (should be empty for this well-ordered example):")
#> [1] "Removed ghost distances (should be empty for this well-ordered example):"
print(removed_distances)
#> Empty data.table (0 rows and 4 cols): Atom1,Atom2,Distance,min_allowed_dist
```

2.4 Filtering by Element Exclusion

To focus an analysis on a specific part of a structure (e.g., a host framework without guest atoms), `filter_by_elements()` allows for the complete removal of bonds involving certain elements.

```
# Let's filter our bond table to exclude any bonds involving Strontium ("Sr").
# Since all bonds in this structure are Si-Sr, the result should be an empty table.
bonds_without_sr <- filter_by_elements(
    distances = bonded_pairs_with_error,
    atomic_coordinates = atomic_coordinates,
    elements_to_exclude = "Sr"
)

cat("Number of bonds in original table:", nrow(bonded_pairs_with_error), "\n")
#> Number of bonds in original table: 14
cat("Number of bonds after excluding 'Sr':", nrow(bonds_without_sr), "\n")
#> Number of bonds after excluding 'Sr': 0
```

2.5 Calculating Weighted Average Network Distance

After filtering the data, a common goal is to compute a single summary statistic that represents the structure. The `calculate_weighted_average_network_distance()` function calculates a representative bond length for a specified atomic network (defined by a set of Wyckoff sites).

Formula Context: Weighted Average Network Distance To accurately reflect the contribution of each bond type in a complex crystal, the average distance (d_{weighted}) is calculated using a true weighted mean. The weight for each bond accounts for the crystallographic probability of that bond occurring.

$$d_{\text{weighted}} = \frac{\sum_{(i,j)\in P} w_{ij} \cdot d_{ij}}{\sum_{(i,j)\in P} w_{ij}} \quad \text{where} \quad w_{ij} = (m_i \cdot o_i) \cdot (m_j \cdot o_j)$$

In this formula: - P is the set of all valid atom pairs after all filtering steps. - d_{ij} is the distance between atom i and atom j. - w_{ij} is the statistical weight of the bond between i and j. - m_i and o_i are the site **multiplicity** and **occupancy** of atom i, respectively. The product $m_i \cdot o_i$ gives the effective number of atoms of type i at that site in the unit cell.

This weighting scheme ensures that bonds between high-multiplicity or high-occupancy sites contribute more to the final average, providing a more physically meaningful representation of the overall structure.

```r
# Calculate the weighted average distance for the entire Sr2Si network.
# First, we define the Wyckoff sites belonging to the network. Here, it's just "4c".
network_wyckoff_sites <- "4c"

# The function uses the full distance table and coordinate info.
weighted_avg_dist <- calculate_weighted_average_network_distance(
    distances = distances,
    atomic_coordinates = atomic_coordinates,
    wyckoff_symbols = network_wyckoff_sites
)

cat("Weighted average network distance for the '4c' sites:", weighted_avg_dist, "Å\n")
#> Weighted average network distance for the '4c' sites: 11.65494 Å
```

Conclusion

This vignette has demonstrated the core workflow of the `crystract` package, from the high-level, automated processing of multiple CIF files with `analyze_cif_files` to a detailed, step-by-step breakdown of the underlying calculations. We have seen how the package extracts fundamental data, builds a complete crystal model, calculates key geometric properties like bond distances and angles, and rigorously propagates experimental uncertainties into these final results.

Furthermore, with a powerful suite of post-processing tools such as `filter_atoms_by_symbol`, `filter_by_wyckoff_symbol`, `filter_ghost_distances`, and `calculate_weighted_average_network_distance`, you can easily refine, clean, and analyze the generated datasets to focus on the specific chemical or crystallographic features of interest.

The goal of `crystract` is to provide a robust and accessible platform for crystallographic data mining in R. We encourage you to apply this workflow to your own CIF files and explore the rich structural information that can be uncovered.