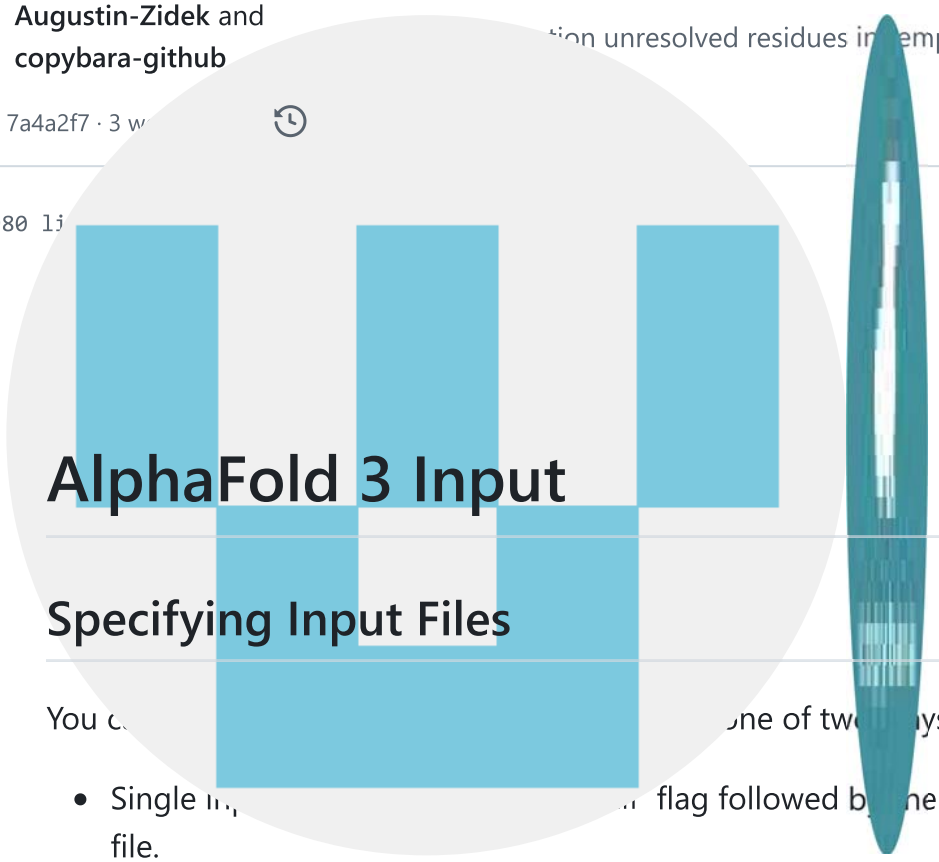

AlphaFold 3 Input

Specifying Input Files

You c one of tw ys:

- Single i flag followed b e path to a single JSON file.
- Multiple input files: Use the `--input_dir` flag followed by the path to a directory of JSON files.

Input Format

AlphaFold 3 uses a custom JSON input format differing from the [AlphaFold Server JSON input format]. See [below] for more information.

- Specifying custom multiple sequence alignment (MSA) for protein and RNA chains.
- Specifying custom structural templates for protein chains.
- Specifying ligands using [Chemical Component Dictionary (CCD)] codes.
- Specifying ligands using SMILES.
- Specifying ligands by defining them using the CCD mmCIF format and supplying them via the [user-provided CCD].

- Specifying covalent bonds between entities.
- Specifying multiple random seeds.

AlphaFold Server JSON Compatibility

The AlphaFold Server uses a separate JSON format from the one used here in the AlphaFold 3 codebase. In particular, the JSON format used in the AlphaFold 3 codebase offers more flexibility and control in defining custom ligands, branched glycans, and covalent bonds between entities.

We provide a converter in `run_alphafold.py` which automatically detects the input JSON format, denoted `dialect` in the converter code. The converter denotes the AlphaFoldServer JSON as `alphafoldserver`, and the JSON format defined here in the AlphaFold 3 codebase as `alphafold3`. If the detected input JSON format is `alphafoldserver`, then the converter will translate that into the JSON format `alphafold3`.

Multiple Inputs

The top-level of the `alphafoldserver` JSON format is a list, allowing specification of multiple inputs in a single JSON. In contrast, the `alphafold3` JSON format requires exactly one input per JSON file. Specifying multiple inputs in a single `alphafoldserver` JSON is fully supported.

Note that the converter distinguishes between `alphafoldserver` and `alphafold3` JSON formats by checking if the top-level of the JSON is a list or not. In particular, if you pass in a `alphafoldserver`-style JSON without a top-level list, then this is considered incorrect and `run_alphafold.py` will raise an error.

Glycans

If the JSON in `alphafoldserver` format specifies glycans, the converter will raise an error. This is because translating glycans specified in the `alphafoldserver` format to the `alphafold3` format is not currently supported.

Random Seeds

The `alphafoldserver` JSON format allows users to specify `"modelSeeds": []`, in which case a seed is chosen randomly for the user. On the other hand, the `alphafold3` format requires users to specify a seed.

The converter will choose a seed randomly if `"modelSeeds": []` is set when translating from `alphafoldserver` JSON format to `alphafold3` JSON format. If seeds are specified in the `alphafoldserver` JSON format, then those will be preserved in the translation to the `alphafold3` JSON format.

Ions

While AlphaFold Server treats ions and ligands as different entity types in the JSON format, AlphaFold 3 treats ions as ligands. Therefore, to specify e.g. a magnesium ion, one would specify it as an entity of type `ligand` with `ccdCodes: ["MG"]`.

Sequence IDs

The `alphafold3` JSON format requires the user to specify a unique identifier (`id`) for each entity. On the other hand, the `alphafoldserver` does not allow specification of an `id` for each entity. Thus, the converter automatically assigns one.

The converter iterates through the list provided in the `sequences` field of the `alphafoldserver` JSON format, assigning an `id` to each entity using the following order ("reverse spreadsheet style"):

```
A, B, ..., Z, AA, BA, CA, ..., ZA, AB, BB, CB, ..., ZB, ...
```

For any entity with `count > 1`, an `id` is assigned arbitrarily to each "copy" of the entity.

Top-level Structure

The top-level structure of the input JSON is:

```
{
  "name": "Job name goes here",
  "modelSeeds": [1, 2],  # At least one seed required.
  "sequences": [
    {"protein": {...}},
    {"rna": {...}},
    {"dna": {...}},
    {"ligand": {...}}
  ],
  "bondedAtomPairs": [...],  # Optional.
  "userCCD": "...",  # Optional, mutually exclusive with userCCDPath.
  "userCCDPath": "...",  # Optional, mutually exclusive with userCCD.
  "dialect": "alphafold3",  # Required.
  "version": 3  # Required.
}
```

The fields specify the following:

- `name: str`: The name of the job. A sanitised version of this name is used for naming the output files.

- `modelSeeds: list[int]` : A list of integer random seeds. The pipeline and the model will be invoked with each of the seeds in the list. I.e. if you provide *n* random seeds, you will get *n* predicted structures, each with the respective random seed. You must provide at least one random seed.
- `sequences: list[Protein | RNA | DNA | Ligand]` : A list of sequence dictionaries, each defining a molecular entity, see below.
- `bondedAtomPairs: list[Bond]` : An optional list of covalently bonded atoms. These can link atoms within an entity, or across two entities. See more below.
- `userCCD: str` : An optional string with user-provided chemical components dictionary. This is an expert mode for providing custom molecules when SMILES is not sufficient. This should also be used when you have a custom molecule that needs to be bonded with other entities - SMILES can't be used in such cases since it doesn't give the possibility of uniquely naming all atoms. It can also be used to provide a reference conformer for cases where RDKit fails to generate a conformer. See more below.
- `userCCDPath: str` : An optional path to a file that contains the user-provided chemical components dictionary instead of providing it inline using the `userCCD` field. The path can be either absolute, or relative to the input JSON path. The file must be in the CCD mmCIF format, and could be either plain text, or compressed using gzip, xz, or zstd.
- `dialect: str` : The dialect of the input JSON. This must be set to `alphafold3` . See AlphaFold Server JSON Compatibility for more information.
- `version: int` : The version of the input JSON. This must be set to 1 or 2. See AlphaFold Server JSON Compatibility and versions below for more information.

Versions

The top-level `version` field (for the `alphafold3` dialect) can be either `1` , `2` , or `3` . The following features have been added in respective versions:

- `1` : the initial AlphaFold 3 input format.
- `2` : added the option of specifying external MSA and templates using newly added fields `unpairedMsaPath` , `pairedMsaPath` , and `mmcifPath` .
- `3` : added the option of specifying external user-provided CCD using newly added field `userCCDPath` .

Sequences

The `sequences` section specifies the protein chains, RNA chains, DNA chains, and ligands. Every entity in `sequences` must have a unique ID. IDs don't have to be sorted alphabetically.

Protein

Specifies a single protein chain.

```
{
  "protein": {
    "id": "A",
    "sequence": "PVLSCGEWQL",
    "modifications": [
      {"ptmType": "HY3", "ptmPosition": 1},
      {"ptmType": "P1L", "ptmPosition": 5}
    ],
    "unpairedMsa": ...,  # Mutually exclusive with unpairedMsaPath.
    "unpairedMsaPath": ...,  # Mutually exclusive with unpairedMsa.
    "pairedMsa": ...,  # Mutually exclusive with pairedMsaPath.
    "pairedMsaPath": ...,  # Mutually exclusive with pairedMsa.
    "templates": [...]
  }
}
```

The fields specify the following:

- `id: str | list[str]` : An uppercase letter or multiple letters specifying the unique IDs for each copy of this protein chain. The IDs are then also used in the output mmCIF file. Specifying a list of IDs (e.g. `["A", "B", "C"]`) implies a homomeric chain with multiple copies.
- `sequence: str` : The amino-acid sequence, specified as a string that uses the 1-letter standard amino acid codes.
- `modifications: list[ProteinModification]` : An optional list of post-translational modifications. Each modification is specified using its CCD code and 1-based residue position. In the example above, we see that the first residue won't be a proline (`P`) but instead `HY3` .
- `unpairedMsa: str` : An optional multiple sequence alignment for this chain. This is specified using the A3M format (equivalent to the FASTA format, but also allows gaps denoted by the hyphen `-` character). See more details below.
- `unpairedMsaPath: str` : An optional path to a file that contains the multiple sequence alignment for this chain instead of providing it inline using the `unpairedMsa` field. The path can be either absolute, or relative to the input JSON path. The file must be in the A3M format, and could be either plain text, or compressed using gzip, xz, or zstd.
- `pairedMsa: str` : We recommend *not* using this optional field and using the `unpairedMsa` for the purposes of pairing. See more details below.
- `pairedMsaPath: str` : An optional path to a file that contains the multiple sequence alignment for this chain instead of providing it inline using the `pairedMsa` field. The path can be either absolute, or relative to the input JSON path. The file must be

in the A3M format, and could be either plain text, or compressed using gzip, xz, or zstd.

- `templates: list[Template]` : An optional list of structural templates. See more details below.

RNA

Specifies a single RNA chain.

```
{
  "rna": {
    "id": "A",
    "sequence": "AGCU",
    "modifications": [
      {"modificationType": "2MG", "basePosition": 1},
      {"modificationType": "5MC", "basePosition": 4}
    ],
    "unpairedMsa": ...,  # Mutually exclusive with unpairedMsaPath.
    "unpairedMsaPath": ...  # Mutually exclusive with unpairedMsa.
  }
}
```

The fields specify the following:

- `id: str | list[str]` : An uppercase letter or multiple letters specifying the unique IDs for each copy of this RNA chain. The IDs are then also used in the output mmCIF file. Specifying a list of IDs (e.g. `["A", "B", "C"]`) implies a homomeric chain with multiple copies.
- `sequence: str` : The RNA sequence, specified as a string using only the letters `A` , `C` , `G` , `U` .
- `modifications: list[RnaModification]` : An optional list of modifications. Each modification is specified using its CCD code and 1-based base position.
- `unpairedMsa: str` : An optional multiple sequence alignment for this chain. This is specified using the A3M format. See more details below.
- `unpairedMsaPath: str` : An optional path to a file that contains the multiple sequence alignment for this chain instead of providing it inline using the `unpairedMsa` field. The path can be either absolute, or relative to the input JSON path. The file must be in the A3M format, and could be either plain text, or compressed using gzip, xz, or zstd.

DNA

Specifies a single DNA chain.

```
{
  "dna": {
    "id": "A",
    "sequence": "GACCTCT",
    "modifications": [
      {"modificationType": "6OG", "basePosition": 1},
      {"modificationType": "6MA", "basePosition": 2}
    ]
  }
}
```

The fields specify the following:

- `id: str | list[str]` : An uppercase letter or multiple letters specifying the unique IDs for each copy of this DNA chain. The IDs are then also used in the output mmCIF file. Specifying a list of IDs (e.g. `["A", "B", "C"]`) implies a homomeric chain with multiple copies.
- `sequence: str` : The DNA sequence, specified as a string using only the letters `A` , `C` , `G` , `T` .
- `modifications: list[DnaModification]` : An optional list of modifications. Each modification is specified using its CCD code and 1-based base position.

Ligands

Specifies a single ligand. Ligands can be specified using 3 different formats:

1. CCD code(s). This is the easiest way to specify ligands. Supports specifying covalent bonds to other entities. CCD from 2022-09-28 is used. If multiple CCD codes are specified, you may want to specify a bond between these and/or a bond to some other entity. See the bonds section below.
2. SMILES string. This enables specifying ligands that are not in CCD. If using SMILES, you cannot specify covalent bonds to other entities as these rely on specific atom names - see the next option for what to use for this case.
3. User-provided CCD + custom ligand codes. This enables specifying ligands not in CCD, while also supporting specification of covalent bonds to other entities and backup reference coordinates for when RDKit fails to generate a conformer. This offers the most flexibility, but also requires careful attention to get all of the details right.

```
{
  "ligand": {
    "id": ["G", "H", "I"],
    "ccdCodes": ["ATP"]
  }
},
```

```
  {
    "ligand": {
      "id": "J",
      "ccdCodes": ["LIG-1337"]
    }
  },
  {
    "ligand": {
      "id": "K",
      "smiles": "CC(=O)OC1C[NH+]2CCC1CC2"
    }
  }
```

The fields specify the following:

- `id: str | list[str]` : An uppercase letter (or multiple letters) specifying the unique ID of this ligand. This ID is then also used in the output mmCIF file. Specifying a list of IDs (e.g. `["A", "B", "C"]`) implies a ligand that has multiple copies.
- `ccdCodes: list[str]` : An optional list of CCD codes. These could be either standard CCD codes, or custom codes pointing to the user-provided CCD.
- `smiles: str` : An optional string defining the ligand using a SMILES string. The SMILES string must be correctly JSON-escaped.

Each ligand may be specified using CCD codes or SMILES but not both, i.e. for a given ligand, the `ccdCodes` and `smiles` fields are mutually exclusive.

SMILES string JSON escaping

The SMILES string must be correctly JSON-escaped, in particular the backslash character must be escaped as two backslashes, otherwise the JSON parser will fail with a `JSONDecodeError` . For instance, the following SMILES string `CCC[C@@H](O)CC\C=C\C=C\C#CC#C\C=C\CO` has to be specified as:

```
  {
    "ligand": {
      "id": "A",
      "smiles": "CCC[C@@H](O)CC\\C=C\\C=C\\C#CC#C\\C=C\\CO"
    }
  }
```

You can JSON-escape the SMILES string using the `jq` command-line tool which should be easily installable on most Linux systems:

```
jq -R . <<< 'CCC[C@@H](O)CC\C=C\C=C\C#CC#C\C=C\CO'  # Replace with your SMI
```

Alternatively, you can use this Python code:

```
import json

smiles = r'CCC[C@@H](O)CC\C=C\C=C\C#CC#C\C=C\CO'  # Replace with your SMILE
print(json.dumps(smiles))
```

Reference structure construction with SMILES

For some ligands and some random seeds, RDKit might fail to generate a conformer, indicated by the `Failed to construct RDKit reference structure` error message. In this case, you can either provide a reference structure for the ligand using the user-provided CCD Format, or try increasing the number of RDKit conformer iterations using the `--conformer_max_iterations=...` flag.

Ions

Ions are treated as ligands, e.g. a magnesium ion would simply be a ligand with `ccdCodes: ["MG"]`.

Multiple Sequence Alignment

Protein and RNA chains allow setting a custom Multiple Sequence Alignment (MSA). If not set, the data pipeline will automatically build MSAs for protein and RNA entities using Jackhmmer/Nhmmer search over genetic databases as described in the paper.

RNA Multiple Sequence Alignment

RNA `unpairedMsa` can be either:

1. Unset (or set explicitly to `null`). AlphaFold 3 won't build MSA for this RNA chain.
2. Set to an empty string (`""`). AlphaFold 3 won't build MSA and will run MSA-free for this RNA chain.
3. Set to a non-empty A3M string. AlphaFold 3 will use the provided MSA for this RNA chain.

Protein Multiple Sequence Alignment

For protein chains, the situation is slightly more complicated due to paired and unpaired MSA (see MSA Pairing below for more details).

The following combinations are valid for a given protein chain:

1. Both `unpairedMsa` and `pairedMsa` fields are unset (or explicitly set to `null`), AlphaFold 3 will build both MSAs automatically. This is the recommended option.
2. The `unpairedMsa` is set to to a non-empty A3M string, `pairedMsa` set to an empty string (`""`). AlphaFold 3 won't build MSA, will use the `unpairedMsa` as is and run `pairedMSA`-free.
3. The `pairedMsa` is set to to a non-empty A3M string, `unpairedMsa` set to an empty string (`""`). AlphaFold 3 won't build MSA, will use the `pairedMsa` and run `unpairedMSA`-free. **This option is not recommended**, see MSA Pairing below.
4. Both `unpairedMsa` and `pairedMsa` fields are set to an empty string (`""`). AlphaFold 3 will not build the MSA and the MSA input to the model will be just the query sequence (equivalent to running completely MSA-free).
5. Both `unpairedMsa` and `pairedMsa` fields are set to a custom non-empty A3M string, AlphaFold 3 will use the provided MSA instead of building one as part of the data pipeline. This is considered an expert option.

Note that both `unpairedMsa` and `pairedMsa` have to either be *both* set (i.e. non-`null`), or both unset (i.e. both `null`, explicitly or implicitly). Typically, when setting `unpairedMsa`, you will set the `pairedMsa` to an empty string (`""`). For example this will run the protein chain A with the given MSA, but without any templates (template-free):

```
{
  "protein": {
    "id": "A",
    "sequence": ...,
    "unpairedMsa": "The A3M you want to run with",
    "pairedMsa": "",
    "templates": []
  }
}
```

When setting your own MSA, you have to make sure that:

1. The MSA is in the A3M format. This means adhering to the FASTA format while also allowing lowercase characters denoting inserted residues and hyphens (`-`) denoting gaps in sequences.
2. The first sequence is exactly equal to the query sequence.
3. If all insertions are removed from MSA hits (i.e. all lowercase letters are removed), all sequences have exactly the same length as the query (they form an exact rectangular matrix).

MSA Pairing

MSA pairing matters only when folding multiple chains (multimers), since we need to find a way to concatenate MSAs for the individual chains along the sequence dimension. If done naively, by simply concatenating the individual MSA matrices along the sequence dimension and padding so that all MSAs have the same depth, one can end up with rows in the concatenated MSA that are formed by sequences from different organisms.

It may be desirable to ensure that across multiple chains, sequences in the MSA that are from the same organism end up in the same MSA row. AlphaFold 3 internally achieves this by looking for the UniProt organism ID in the `pairedMsa` and pairing sequences based on this information.

We recommend users do the pairing manually or use the output of an appropriate software and then provide the MSA using only the `unpairedMsa` field. This method gives exact control over the placement of each sequence in the MSA, as opposed to relying on name-matching post-processing heuristics used for `pairedMsa`.

When setting `unpairedMsa` manually, the `pairedMsa` must be explicitly set to an empty string (`""`).

For instance, if there are two chains `DEEP` and `MIND` which we want to be paired on organism A and C, we can achieve it as follows:

```
> query
DEEP
> match 1 (organism A)
D--P
> match 2 (organism B)
DD-P
> match 3 (organism C)
DD-P
```

```
> query
MIND
> match 1 (organism A)
M--D
> Empty hit to make sure pairing is achieved
----
> match 2 (organism C)
MIN-
```

The resulting MSA when chains are concatenated will then be:

```
> query
DEEPMIND
> match 1 + match 1
```

```
D--PM--D
> match 2 + padding
DD-P----
> match 3 + match 2
DD-PMIN-
```

Structural Templates

Structural templates can be specified only for protein chains:

```
"templates": [
  {
    "mmcif": ...,  # Mutually exclusive with mmcifPath.
    "mmcifPath": ...,  # Mutually exclusive with mmcif.
    "queryIndices": [0, 1, 2, 4, 5, 6],
    "templateIndices": [0, 1, 2, 3, 4, 8]
  }
]
```

The fields specify the following:

- `mmcif: str` : A string containing the single chain protein structural template in the mmCIF format.
- `mmcifPath: str` : An optional path to a file that contains the mmCIF with the structural template instead of providing it inline using the `mmcifPath` field. The path can be either absolute, or relative to the input JSON path. The file must be in the mmCIF format, and could be either plain text, or compressed using gzip, xz, or zstd.
- `queryIndices: list[int]` : 0-based indices in the query sequence, defining the mapping from query residues to template residues.
- `templateIndices: list[int]` : 0-based indices in the template sequence, specifying the mapping from query residues to template residues defined in the mmCIF file. Note that unresolved mmCIF residues must be taken into account when specifying template indices.

A template is specified as an mmCIF string containing a single chain with the structural template together with a 0-based mapping that maps query residue indices to the template residue indices. The mapping is specified using two lists of the same length. E.g. to express a mapping `{0: 0, 1: 2, 2: 5, 3: 6}`, you would specify the two indices lists as:

```
"queryIndices":    [0, 1, 2, 3],
"templateIndices": [0, 2, 5, 6]
```

Note that mmCIFs can have residues with missing atom coordinates (present in residue tables but missing in the `_atom_site` table) – these must be taken into account when specifying template indices. E.g. to align residues 4–7 in a template with unresolved residues 1, 2, 3 and resolved residues 4, 5, 6, 7, you need to set the template indices to 3, 4, 5, 6 (since 0-based indexing is used). An example of a protein with unresolved residues 1–20 can be found here: https://www.rcsb.org/structure/8UXY.

You can provide multiple structural templates. Note that if an mmCIF containing more than one chain is provided, you will get an error since it is not possible to determine which of the chains should be used as the template.

You can run template-free (but still run genetic search and build MSA) by setting templates to `[]` and either explicitly setting both `unpairedMsa` and `pairedMsa` to `null`:

```
"protein": {
  "id": "A",
  "sequence": ...,
  "pairedMsa": null,
  "unpairedMsa": null,
  "templates": []
}
```

Or you can simply fully omit them:

```
"protein": {
  "id": "A",
  "sequence": ...,
  "templates": []
}
```

You can also run with pre-computed MSA, but let AlphaFold 3 search for templates. This can be achieved by setting `unpairedMsa` and `pairedMsa`, but keeping templates unset (or set to `null`). The profile given as an input to Hmmsearch when searching for templates will be built from the provided `unpairedMsa`:

```
"protein": {
  "id": "A",
  "sequence": ...,
  "unpairedMsa": ...,
  "pairedMsa": ...,
  "templates": null
}
```

Or you can simply fully omit the `templates` field thus setting it implicitly to `null`:

```
  "protein": {
    "id": "A",
    "sequence": ...,
    "unpairedMsa": ...,
    "pairedMsa": ...,
  }
```

Bonds

To manually specify covalent bonds, use the `bondedAtomPairs` field. This is intended for modelling covalent ligands, and for defining multi-CCD ligands (e.g. glycans). Defining covalent bonds between or within polymer entities is not currently supported.

Bonds are specified as pairs of (source atom, destination atom), with each atom being uniquely addressed using 3 fields:

- **Entity ID** (`str`): this corresponds to the `id` field for that entity.
- **Residue ID** (`int`): this is 1-based residue index *within* the chain. For single-residue ligands, this is simply set to 1.
- **Atom name** (`str`): this is the unique atom name *within* the given residue. The atom name for protein/RNA/DNA residues or CCD ligands can be looked up in the CCD for the given chemical component. This also explains why SMILES ligands don't support bonds: there is no atom name that could be used to define the bond. This shortcoming can be addressed by using the user-provided CCD format (see below).

The example below shows two bonds:

```
  "bondedAtomPairs": [
    [["A", 145, "SG"], ["L", 1, "C04"]],
    [["J", 1, "O6"], ["J", 2, "C1"]]
  ]
```

The first bond is between chain A, residue 145, atom SG and chain L, residue 1, atom C04. This is a typical example for a covalent ligand. The second bond is between chain J, residue 1, atom O6 and chain J, residue 2, atom C1. This bond is within the same entity and is a typical example when defining a glycan.

All bonds are implicitly assumed to be covalent bonds. Other bond types are not supported.

Defining Glycans

Glycans are bound to a protein residue, and they are typically formed of multiple chemical components. To define a glycan, define a new ligand with all of the chemical components of the glycan. Then define a bond that links the glycan to the protein residue, and all bonds that are within the glycan between its individual chemical components.

For example, to define the following glycan composed of 4 components (CMP1, CMP2, CMP3, CMP4) bound to an asparagine in a protein chain A:

You will need to specify:

1. Protein chain A.
2. Ligand chain B with the 4 components.
3. Bonds ASN-CMP1, CMP1-CMP2, CMP2-CMP3, CMP2-CMP4.

User-provided CCD

There are two approaches to model a custom ligand not defined in the CCD:

1. If the ligand is not bonded to other entities, it can be defined using a SMILES string.
2. If it is bonded to other entities, or to be able to customise relevant features (such as bond orders, atom names and ideal coordinates used when conformer generation fails), it is necessary to define that particular ligand using the CCD mmCIF format.

Note that if a full CCD mmCIF is provided, any SMILES string input as part of that mmCIF is ignored.

Once defined, this ligand needs to be assigned a name that doesn't clash with existing CCD ligand names (e.g. `LIG-1`). Avoid underscores (_) in the name, as it could cause issues in the mmCIF format.

The newly defined ligand can then be used as a standard CCD ligand using its custom name, and bonds can be linked to it using its named atom scheme.

Conformer Generation

The data pipeline attempts to generate a conformer for ligands using RDKit. The `Mol` used to generate the conformer is constructed either from the information provided in the CCD mmCIF, or from the SMILES string if that is the only information provided.

If conformer generation fails, the model will fall back to using the ideal coordinates in the CCD mmCIF if these are provided. If they are not provided, the model will use the reference coordinates if the last modification date given in the CCD mmCIF is prior to the training cutoff date. If no coordinates can be found in this way, all conformer coordinates are set to zero and the model will output `NaN` (`null` in the output JSON) confidences for the ligand.

Note that sometimes conformer generation failures can be resolved by increasinging the number of RDKit conformer iterations using the `--conformer_max_iterations=...` flag.

User-provided CCD Format

The user-provided CCD must be passed either:

- In the `userCCD` field (in the root of the input JSON) as a string. Note that JSON doesn't allow newlines within strings, so newline characters (`\n`) must be used to delimit lines. Single rather than double quotes should also be used around strings like the chemical formula.
- In the `userCCDPath` field, as a path to a file that contains the user-provided chemical components dictionary. The path can be either absolute, or relative to the input JSON path. The file must be in the CCD mmCIF format, and could be either plain text, or compressed using gzip, xz, or zstd.

The main pieces of information used are the atom names and elements, bonds, and also the ideal coordinates (`pdbx_model_Cartn_{x,y,z}_ideal`) which essentially serve as a structural template for the ligand if RDKit fails to generate conformers for that ligand.

The user-provided CCD can also be used to redefine standard chemical components in the CCD. This can be useful if you need to redefine the ideal coordinates.

Below is an example user-provided CCD redefining component X7F, which serves to illustrate the required sections. For readability purposes, newlines have not been replaced by `\n`.

```
data_MY-X7F
#
_chem_comp.id MY-X7F
_chem_comp.name '5,8-bis(oxidanyl)naphthalene-1,4-dione'
_chem_comp.type non-polymer
_chem_comp.formula 'C10 H6 O4'
_chem_comp.mon_nstd_parent_comp_id ?
```

```
_chem_comp.pdbx_synonyms ?
_chem_comp.formula_weight 190.152
#
loop_
_chem_comp_atom.comp_id
_chem_comp_atom.atom_id
_chem_comp_atom.type_symbol
_chem_comp_atom.charge
_chem_comp_atom.pdbx_leaving_atom_flag
_chem_comp_atom.pdbx_model_Cartn_x_ideal
_chem_comp_atom.pdbx_model_Cartn_y_ideal
_chem_comp_atom.pdbx_model_Cartn_z_ideal
MY-X7F C02 C 0 N -1.418 -1.260 0.018
MY-X7F C03 C 0 N -0.665 -2.503 -0.247
MY-X7F C04 C 0 N 0.677 -2.501 -0.235
MY-X7F C05 C 0 N 1.421 -1.257 0.043
MY-X7F C06 C 0 N 0.706 0.032 0.008
MY-X7F C07 C 0 N -0.706 0.030 -0.004
MY-X7F C08 C 0 N -1.397 1.240 -0.037
MY-X7F C10 C 0 N -0.685 2.443 -0.057
MY-X7F C11 C 0 N 0.679 2.445 -0.045
MY-X7F C12 C 0 N 1.394 1.243 -0.013
MY-X7F O01 O 0 N -2.611 -1.301 0.247
MY-X7F O09 O 0 N -2.752 1.249 -0.049
MY-X7F O13 O 0 N 2.750 1.257 -0.001
MY-X7F O14 O 0 N 2.609 -1.294 0.298
MY-X7F H1 H 0 N -1.199 -3.419 -0.452
MY-X7F H2 H 0 N 1.216 -3.416 -0.429
MY-X7F H3 H 0 N -1.221 3.381 -0.082
MY-X7F H4 H 0 N 1.212 3.384 -0.062
MY-X7F H5 H 0 N -3.154 1.271 0.830
MY-X7F H6 H 0 N 3.151 1.241 -0.880
#
loop_
_chem_comp_bond.atom_id_1
_chem_comp_bond.atom_id_2
_chem_comp_bond.value_order
_chem_comp_bond.pdbx_aromatic_flag
O01 C02 DOUB N
O09 C08 SING N
C02 C03 SING N
C02 C07 SING N
C03 C04 DOUB N
C08 C07 DOUB Y
C08 C10 SING Y
C07 C06 SING Y
C10 C11 DOUB Y
C04 C05 SING N
C06 C05 SING N
C06 C12 DOUB Y
C11 C12 SING Y
C05 O14 DOUB N
C12 O13 SING N
```

```
C03 H1 SING N
C04 H2 SING N
C10 H3 SING N
C11 H4 SING N
O09 H5 SING N
O13 H6 SING N
#
```

Mandatory fields

Parsing the user-provided CCD needs only a subset of the fields that CCD uses. The mandatory fields are described below. Refer to CCD documentation for more detailed explanation of each field. Note that not all of these fields are input to the model, but they are necessary for the data pipeline to run – see the Model input fields section below.

Singular fields (containing just a single value)

- `_chem_comp.id` : The ID of the component. Must match the `_data` record and must not contain special CIF characters (like `_` or `#`).
- `_chem_comp.name` : Optional full name of the component. If unknown, set to `?`.
- `_chem_comp.type` : Type of the component, typically `non-polymer`.
- `_chem_comp.formula` : Optional component formula. If unknown, set to `?`.
- `_chem_comp.mon_nstd_parent_comp_id` : Optional parent component ID. If unknown, set to `?`.
- `_chem_comp.pdbx_synonyms` : Optional synonym IDs. If unknown, set to `?`.
- `_chem_comp.formula_weight` : Optional weight of the component. If unknown, set to `?`.

Per-atom fields (containing one record per atom)

- `_chem_comp_atom.comp_id` : Component ID.
- `_chem_comp_atom.atom_id` : Atom ID.
- `_chem_comp_atom.type_symbol` : Atom element type.
- `_chem_comp_atom.charge` : Atom charge.
- `_chem_comp_atom.pdbx_leaving_atom_flag` : Optional flag determining whether this is a leaving atom. If unset, assumed to be no (`N`) for all atoms.
- `_chem_comp_atom.pdbx_model_Cartn_x_ideal` : Ideal x coordinate.
- `_chem_comp_atom.pdbx_model_Cartn_y_ideal` : Ideal y coordinate.
- `_chem_comp_atom.pdbx_model_Cartn_z_ideal` : Ideal z coordinate.

Per-bond fields (containing one record per bond)

- `_chem_comp_bond.atom_id_1` : The ID of the first of the two atoms that define the bond.
- `_chem_comp_bond.atom_id_2` : The ID of the second of the two atoms that define the bond.
- `_chem_comp_bond.value_order` : The bond order of the chemical bond associated with the specified atoms.
- `_chem_comp_bond.pdbx_aromatic_flag` : Whether the bond is aromatic.

Model input fields

The following fields are used to generate input for the model:

- `_chem_comp_atom.atom_id` : Atom ID.
- `_chem_comp_atom.type_symbol` : Atom element type.
- `_chem_comp_atom.charge` : Atom charge.
- `_chem_comp_atom.pdbx_model_Cartn_x_ideal` : Ideal x coordinate. Only used if conformer generation fails.
- `_chem_comp_atom.pdbx_model_Cartn_y_ideal` : Ideal y coordinate. Only used if conformer generation fails.
- `_chem_comp_atom.pdbx_model_Cartn_z_ideal` : Ideal z coordinate. Only used if conformer generation fails.
- `_chem_comp_bond.atom_id_1` : The ID of the first of the two atoms that define the bond.
- `_chem_comp_bond.atom_id_2` : The ID of the second of the two atoms that define the bond.

Full Example

An example illustrating all the aspects of the input format is provided below. Note that AlphaFold 3 won't run this input out of the box as it abbreviates certain fields and the sequences are not biologically meaningful.

```
{
  "name": "Hello fold",
  "modelSeeds": [10, 42],
  "sequences": [
    {
      "protein": {
        "id": "A",
        "sequence": "PVLSCGEWQL",
        "modifications": [
          {"ptmType": "HY3", "ptmPosition": 1},
          {"ptmType": "P1L", "ptmPosition": 5}
        ],
```

```
      "unpairedMsa": ...,
    }
  },
  {
    "protein": {
      "id": "B",
      "sequence": "RPACQLW",
      "templates": [
        {
          "mmcif": ...,
          "queryIndices": [0, 1, 2, 4, 5, 6],
          "templateIndices": [0, 1, 2, 3, 4, 8]
        }
      ]
    }
  },
  {
    "dna": {
      "id": "C",
      "sequence": "GACCTCT",
      "modifications": [
        {"modificationType": "6OG", "basePosition": 1},
        {"modificationType": "6MA", "basePosition": 2}
      ]
    }
  },
  {
    "rna": {
      "id": "E",
      "sequence": "AGCU",
      "modifications": [
        {"modificationType": "2MG", "basePosition": 1},
        {"modificationType": "5MC", "basePosition": 4}
      ],
      "unpairedMsa": ...
    }
  },
  {
    "ligand": {
      "id": ["F", "G", "H"],
      "ccdCodes": ["ATP"]
    }
  },
  {
    "ligand": {
      "id": "I",
      "ccdCodes": ["NAG", "FUC"]
    }
  },
  {
    "ligand": {
      "id": "Z",
      "smiles": "CC(=O)OC1C[NH+]2CCC1CC2"
```

```
        }
      }
    ],
    "bondedAtomPairs": [
      [["A", 1, "CA"], ["G", 1, "CHA"]],
      [["I", 1, "O6"], ["I", 2, "C1"]]
    ],
    "userCCD": ...,
    "dialect": "alphafold3",
    "version": 3
  }
```