Gemmi Documentation

Release 0.3.8

Marcin Wojdyr

Jun 01, 2020

Contents

Gemmi is a library, accompanied by a set of programs, developed primarily for use in **macromolecular crystallography** (MX). For working with:

- macromolecular models (content of PDB, PDBx/mmCIF and mmJSON files),
- refinement restraints (CIF files),
- reflection data (MTZ and mmCIF formats),
- data on a 3D grid (electron density maps, masks, MRC/CCP4 format)
- crystallographic symmetry.

Parts of this library can be useful in structural bioinformatics (for symmetry-aware analysis of protein models), and in other molecular-structure sciences that use CIF files (we have the fastest open-source CIF parser).

Gemmi is open-source (MPL) and portable – it runs on Linux, Windows, MacOS and even inside a web browser if compiled to WebAssembly. It is written in C++11, with Python (2 and 3) bindings, and with partial C and Fortran 2003 interface.

Occasionally, the project gets sidetracked into visualization of the PDB data.

Gemmi is a joint project of Global Phasing Ltd and CCP4.

The project is named after Gemmi Pass. The name can also be expanded as *GEneral MacroMolecular I/o*.

Source code repository: https://github.com/project-gemmi/gemmi

---

Important: As of 2020 the library is under intensive development and not everything is documented yet. Just ask questions.

CHAPTER 1

Contents

1.1 Installation

1.1.1 C++ library

It is a header-only library. You need to ensure that the `include` directory is in your include path when compiling your program. For example:

```
git clone https://github.com/project-gemmi/gemmi.git
c++ -std=c++11 -Igemmi/include -O2 my_program.cpp
```

If you want Gemmi to uncompress gzipped files on the fly (i.e. if you `#include <gemmi/gz.hpp>`) you will also need to link your program with the zlib library.

If a file name is passed to Gemmi (through `std::string`) it is assumed to be in ASCII or UTF-8.

1.1.2 Python 2.7/3.x module

From source

To install the gemmi module you need pip, git and not too old C++ compiler (GCC 4.8+, Clang 3.4+, MSVC 2015+, ICC 17+):

```
pip install gemmi
```

Alternatively, to install the latest version directly from the repository:

```
pip install git+https://github.com/project-gemmi/gemmi.git
```

or clone the project (or download a zip file) and from the top-level directory do:

```
pip install .
```

If gemmi is already installed, uninstall the old version first (`pip uninstall`) or add option `--upgrade`.

On Windows Python 3.5+ should automatically find an appropriate compiler (MSVC 2015+) . If the compiler is not installed, pip shows a message with a download link. For Python 2.7 pip prefers MSVC 2008, which is too old to compile gemmi. You may still use MSVC 2015, 2017 or 2019, but before invoking pip you need to set the compiler environment with one of these commands:

```
"%VS140COMNTOOLS%\..\..\VC\vcvarsall.bat" x64
"%VS140COMNTOOLS%\..\..\VC\vcvarsall.bat"
```

If you'd like to use PyPy instead of CPython – PyPy2.7 >= 5.7 is supported (although only occasionally tested – open an issue if it doesn't work).

Binaries

If you use the CCP4 suite, you can find gemmi there.

If you use Anaconda Python, you can install package conda from conda-forge:

```
conda install -c conda-forge gemmi
```

These distribution channels may have a previous version of gemmi.

1.1.3 Fortran and C bindings

The Fortran bindings are in early stage and are not documented yet. They use the ISO_C_BINDING module introduced in Fortran 2003. You may see the `fortran/` directory to know what to expect. The bindings and usage examples can be compiled with CMake:

```
cmake -D USE_FORTRAN=1 .
make
```

The C bindings are used only for making Fortan bindings, but they should be usable on their own. If you use cmake to build the project you get a static library `libcgemmi.a` that can be used from C, together with the `fortran/*.h` headers.

1.1.4 Gemmi program

The library comes with a command-line program. To build it from source:

```
git clone https://github.com/project-gemmi/gemmi.git
cd gemmi
cmake .
make
```

1.1.5 Credits

This project is using code from a number of third-party open-source projects.

Projects used in the C++ library and included under `include/gemmi/third_party/`:

- PEGTL – library for creating PEG parsers. License: MIT.
- sajson – high-performance JSON parser. License: MIT.
- PocketFFT – FFT library. License: 3-clause BSD.
- stb_sprintf – locale-independent snprintf() implementation. License: Public Domain.
- tinydir – directory (filesystem) reader. License: 2-clause BSD.

Code derived from the following projects is used in the library:

- ksw2 – sequence alignment in `seqalign.hpp` is based on the ksw_gg function from ksw2. License: MIT.
- Larch – calculation of f' and f'' in `fprime.hpp` is based on CromerLiberman code from Larch. License: 2-clause BSD.

Projects included under `third_party/`, not used in the library itself, but used in command-line utilities, python bindings or tests that are distributed with the library:

- The Lean Mean C++ Option Parser – command-line option parser. License: MIT.
- doctest – testing framework. License: MIT.
- linalg.h – linear algebra library. License: Public Domain.
- zlib – a subset of the zlib library for uncompressing gz files, used as a fallback when the zlib library is not found in the system. License: zlib.

Not distributed with Gemmi:

- pybind11 – used for creating Python bindings. License: 3-clause BSD.
- cctbx – used in tests and in scripts that generated space group data. License: 3-clause BSD.

Email me if I forgot about something.

1.2 CIF Parser

This section covers working with CIF files on a syntactic level.

Higher-level functions that understand semantics of:

- small molecule or inorganic CIF files,
- macromolecular PDBx/mmCIF,
- and monomer/ligand cif files as used for macromolecular restraints

are documented in section *Molecular models*.

1.2.1 What are STAR, CIF, DDL, mmCIF?

(in case someone comes here when looking for a serialization format)

STAR is a human-readable data serialization format (think XML or JSON) that happens to be known and used only in molecular-structure sciences.

CIF (Crystallographic Information File) – a file format used in crystallography – is a restricted derivative of STAR. It is restricted in features (to make implementation easier), but also imposes arbitrary limits – for example on the line length.

DDL is a schema language for STAR/CIF.

All of them (STAR, CIF and DDL) have multiple versions. We will be more specific in the following sections.

The STAR/CIF syntax is relatively simple, but may be confusing at first. (Note that the initial version of STAR was published by Sydney Hall in 1991 – before XML and long before JSON and YAML, not to mention TOML).

Key-value pairs have the form:

```
_year 2017  # means year = 2017
```

Blocks (sections) begin with the `data_` keyword with a block name glued to it:

```
data_tomato   # [tomato]
_color red    # color = "red"
```

Importantly, unlike XML/JSON/YAML/TOML, STAR is designed to concisely represent tabular data. This example from TOML docs:

```
# TOML
points = [ { x = 1, y = 2, z = 3 },
           { x = 7, y = 8, z = 9 },
           { x = 2, y = 4, z = 8 } ]
```

could be expressed as a so-called *loop* in STAR (keyword `loop_` followed by column names followed by values):

```
# STAR/CIF
loop_
_points.x _points.y _points.z
1 2 3
7 8 9
2 4 8
```

Typically, long tables (loops) make most of the CIF content:

```
1    N N   . LEU A 11  ? 0.5281 0.5618 0.5305 -0.0327 -0.0621 0.0104
2    C CA  . LEU A 11  ? 0.5446 0.5722 0.5396 -0.0317 -0.0632 0.0080
# many, many lines...
5331 S SD  . MET C 238 ? 2.2952 2.3511 2.3275 -0.0895 0.0372  -0.0230
5332 C CE  . MET C 238 ? 1.5699 1.6247 1.6108 -0.0907 0.0388  -0.0244
```

The dot and question mark in the example above are two null types. In the CIF spec: `?` = *unknown* and `.` = *not applicable*. In mmCIF files `.` is used for mandatory items, `?` for not mandatory.

The CIF syntax has a serious flaw resulting from historical trade-offs: a string that can be interpreted as a number does not need to be quoted. Therefore, the type of `5332` above is not certain: the JSON equivalent can be either `5332` or `"5332"`.

Note: "STAR File" is trademarked by IUCr, and it used to be patented.

The mmCIF format (by mmCIF we mean what is more formally called PDBx/mmCIF) is the CIF syntax + a huge dictionary (ontology/schema) in DDL2. The dictionary defines relations between columns in different tables, which makes it resemble a relational database (it was designed at the height of popularity of RDBMSs).

Where are the specs?

International Tables for Crystallography Vol. G (2006) describes all of the STAR, CIF 1.1, DDL1 and DDL2. If you don't have access to it – IUCr website has specs of CIF1.1 and DDLs. As far as I can tell all versions of the STAR spec are behind paywalls.

Later versions of the formats (hardly used as of 2017) are described in these articles: STAR (2012), DDLm and dREL (2012), and CIF 2.0 (2016). Only the last one is freely available.

PDBx/mmCIF is documented at mmcif.pdb.org.

1.2.2 What is parsed?

The parser supports CIF 1.1 spec and some extras.

Currently, it is available as:

- C++11 header-only library, and
- Python (2 and 3) extension module.

We use it to read:

- mmCIF files (both coordinates and structure factors)
- CIF files from Crystallography Open Database (COD)
- Chemical Component Dictionary from PDB
- DDL1 and DDL2 dictionaries from IUCr and PDB
- monomer library a.k.a. Refmac dictionary

The parser handles:

- all constructs of CIF 1.1 (including *save frames*),
- the `global_` and `stop_` keywords from STAR – needed for Refmac monomer library and `mmcif_nmr-star.dic`, respectively.

It could be extended to handle also the new features of CIF 2.0 or even the full STAR format, but we don't have a good reason to do this. The same goes for DDLm/dREL.

The parser does not handle CIF 1.1 conventions, as they are not part of the syntax: line wrapping (`eol;\eol`), Greek letters (`\m` -> μ), accented letters (`\'o` -> ó), special alphabetic characters (`\%A` -> Å) and other codes (`\\infty` -> ∞).

CIF parsers in the small-molecules field need to deal with incorrect syntax. The papers about iotbx.cif and COD::CIF::Parser enumerate 12 and 10 common error categories, respectively. Nowadays the problem is less severe, especially in the MX community that embraced the CIF format later. So we've decided to add only the following rules to relax the syntax:

- names and lines can have any length like in STAR (the CIF spec imposes the limit of 2048 characters, but some mmCIF files from PDB exceed it, e.g. 3j3q.cif),
- quoted strings may contain non-ascii characters (if nothing has changed one entry in the PDB has byte A0 corresponding to non-breaking space),
- a table (loop) can have no values if it is followed by another loop or block end (because Refmac monomer library),
- block name (*blockcode*) can be empty, i.e. the block can start with bare `data_` keyword (RELION writes such files),
- unquoted strings cannot start with keywords (STAR spec is ambiguous about this – see StarTools doc for details; this rule is actually about simplifying not relaxing),
- missing value in a key-value pair is optionally allowed if whitespace after the tag ends with a new-line character. More specifically, the parsing step allows for missing value in such case, but the next validation step (which can be skipped when using using low-level functions, such as `parse_file()`) throws an error.

1.2.3 Getting started

C++

CIF parser is implemented in header files, so you do not need to compile Gemmi. It has a single dependency: PEGTL (also header-only), which is included under the `include/gemmi/third_party` directory. All you need is to make sure that Gemmi headers are in your project's include path, and compile your program as C++11 or later.

Let us start with a simple example. This little program reads mmCIF file and shows weights of the chemical components:

```
#include <iostream>
#include <gemmi/cif.hpp>

namespace cif = gemmi::cif;

int main() {
  cif::Document doc = cif::read_file("1mru.cif");
  for (cif::Block& block : doc.blocks)
    for (auto cc : block.find("_chem_comp.", {"id", "formula_weight"}))
      std::cout << cc[0] << " weights " << cc[1] << std::endl;
}
```

To compile it on Unix system you need to fetch Gemmi source code and run a compiler:

```
git clone https://github.com/project-gemmi/gemmi.git
c++ -std=c++11 -Igemmi/include -O2 my_program.cpp
```

Python

Python module for Python 2.7 and 3.x can be installed with pip, as described in the *Installation* section. After installation `pydoc gemmi.cif` should list all classes and methods.

To start with a simple example, here is a program that says hello to each element found in mmCIF:

```
import sys
from gemmi import cif

greeted = set()
for path in sys.argv[1:]:
    try:
        doc = cif.read_file(path)  # copy all the data from mmCIF file
        block = doc.sole_block()  # mmCIF has exactly one block
        for element in block.find_loop("_atom_site.type_symbol"):
            if element not in greeted:
                print("Hello " + element)
                greeted.add(element)
    except Exception as e:
        print("Oops. %s" % e)
        sys.exit(1)
```

More complex examples are shown in the *Examples* section.

Internally, Python bindings use pybind11.

1.2.4 DOM and SAX

The terms DOM and SAX originate from XML parsing, but they became also names of general parsing styles. Gemmi can parse CIF files in two ways that correspond to DOM and SAX:

- The usual way is to parse a file or a string into a document (DOM) that can be easily accessed and manipulated.
- Alternatively, from C++ only, one can define own PEGTL Actions for to the grammar rules from `cif.hpp`. These actions will be triggered while reading a CIF file.

This documention covers the DOM parsing only. The hierarchy in the DOM reflects the structure of CIF 1.1:

- Document contains blocks.
- Block can contain name-value pairs, loops and frames.
- Frame can contain name-value pairs and loops.
- Loop (*m*×*n* table) contains *n* column names and *m*×*n* values.

Names are often called *tags*. The leading _ is usually treated as part of the tag, not just a syntactic feature. So we store tag string with the underscore (`_my_tag`), and function that take tags as arguments expect strings starting with _. The case of tags is preserved.

Values have types. CIF 1.1 defines four base types:

- char (string)
- uchar (ughh.. case-insensitive string)
- numb (number that cannot be recognized as number on the syntax level)
- null (one of two possible nulls: `?` and `.`)

Additionally, DDL2 dictionaries specify subtypes. For example, `int` and `float` are mmCIF subtypes of `numb`.

Since in general it is not possible to infer type without the corresponding dictionary, the DOM stores raw strings (including quotes). They can be later converted to required type using the following helper functions:

- `as_string()` – gets unquoted string,
- `as_number()` – gets floating-point number,
- `as_int()` – gets integer,
- `as_char()` – gets single character,
- `is_null()` – check if the value is null (i.e. `?` or `.`),

and we have also

- `quote()` – the opposite of `as_string()` – add quotes appropriate for the content of the string (usually, no quotes are necessary and no quotes are added).

C++

All these helper functions are defined in `gemmi/cifdoc.hpp` except for `as_number()` which is in `gemmi/numb.hpp`. They take a string as an argument and work as expected, for example:

```
double rfree = cif::as_number(raw_rfree_string); // NaN if it's '?' or '.'
```

Python

```
>>> from gemmi import cif
>>> cif.as_number('123')
123.0
>>> cif.as_int('123')
123
>>> cif.quote('two words')
"'two words'"
>>> cif.as_string(_)
'two words'
>>> cif.as_number(_)
nan
```

1.2.5 Reading a file

We have a few reading functions that read a file (or a string, or a stream) and return a document (DOM) – an instance of class `Document`.

C++

The reading functions are in the `gemmi::cif` namespace:

```
Document read_file(const std::string& filename)
Document read_memory(const char* data, const size_t size, const char* name)
Document read_cstream(std::FILE *f, size_t bufsize, const char* name)
Document read_istream(std::istream &is, size_t bufsize, const char* name)
```

Parameter `name` is used only when reporting errors. Parameter `bufsize` determines the buffer size and only affects performance. Regardless of the buffer size, the last two options are slower than `read_file()` – they were not optimized for.

Additional header `<gemmi/gz.hpp>` is needed to transparently open a gzipped file (by uncompressing it first into a memory buffer) if the filename ends with `.gz`:

```
// in this and all the next examples: namespace cif = gemmi::cif;
cif::Document doc = cif::read(gemmi::MaybeGzipped(path));
```

If you use it, you must also link the program with zlib. On Unix systems it usually means adding `-lz` to the compiler invocation.

And if the `path` above is `-`, the standard input is read.

Python

```
from gemmi import cif

# read and parse a CIF file
doc = cif.read_file('components.cif')

# the same, but if the filename ends with .gz it is uncompressed on the fly
doc = cif.read('../tests/1pfe.cif.gz')
```

(continues on next page)

(continued from previous page)

```
# read content of a CIF file from string
doc = cif.read_string('data_this _is valid _cif content')
```

Low-level functions

The `read_file()` call is equivalent to the following sequence:

```
path = 'components.cif'
doc = cif.Document()
doc.source = path
doc.parse_file(path)
doc.check_for_missing_values()
doc.check_for_duplicates()
```

The last two functions check for, respectively, missing values in tag-value pairs and duplicated names. It is possible to read erroneous CIF files by skipping these checks.

Analogically, function `read_string()` can be replaced by a similar sequence with `Document.parse_string()` doing the main job:

```
>>> doc = cif.Document()
>>> doc.parse_string('data_this _tag_has_no_value\n')
>>> doc[0].find_value('_tag_has_no_value')
''
>>> try: doc.check_for_missing_values()
... except RuntimeError: print('missing value')
...
missing value
```

1.2.6 Writing a file

Reading and writing a file does not preserve whitespaces. Instead, we have a few choices for "styling" of the output:

- `Style::Simple` writes out the DOM structure adding blank lines between mmCIF categories,
- `Style::NoBlankLines` does not add blank lines,
- `Style::PreferPairs` writes single-row loops as pairs,
- `Style::Pdbx` additionally puts `#` (empty comments) between categories, mimicking the peculiar formatting of PDBx/mmCIF files in the official wwPDB archive. It enables diff-ing original and modified files with option `--ignore-space-change`.
- `Style::Indent35` writes values from pairs from 35th column,

C++

The functions writing `cif::Document` to C++ stream is in a separate header `gemmi/to_cif.hpp`:

```
void write_cif_to_stream(std::ostream& os, const Document& doc, Style style)
```

Python

In Python, the function that writes the document to a file is a method of the `Document` class:

```
>>> doc.write_file('1pfe-modified.cif')
```

It can take the style as optional, second argument:

```
>>> doc.write_file('1pfe-modified.cif', cif.Style.PreferPairs)
>>> doc.write_file('1pfe-styled.cif', cif.Style.Pdbx)
```

The `Document` class also has a method `as_string()` which returns the text that would be written by `write_file()`.

1.2.7 Document

`Document` is made of blocks with data. The blocks can be iterated over, accessed by index or by name (each CIF block must have a unique name).

As it is common for cif files to contain only a single block, gemmi has a method `sole_block()` that returns the first block if the document has only one block; otherwise it throws an exception.

At last, is also has a member variable `source` that contains the path of the file from which the document was read (if it was read from a file).

C++

`Document` has the two member variables:

```
std::string source;  // filename or the name passed to read_memory()
std::vector<Block> blocks;
```

Each `Block` corresponds to a data block. To access a block with known name use:

```
Block* Document::find_block(const std::string& name)
```

To access the only block in the file you may use:

```
Block& Document::sole_block()
```

A new `Document` instance can be created with default constructor. To modify a document you need to access directly its member variables. With one exception: when adding new blocks you can use a function that additionally checks if the new name is unique:

```
Block& Document::add_new_block(const std::string& name, int pos=-1)
```

Python

`Document` can be iterated, accessed by block index and by block name:

```
>>> doc = cif.read_file("components.cif")
>>> len(doc)
25219
>>> doc[0]
```

(continues on next page)

(continued from previous page)

```
<gemmi.cif.Block 000>
>>> doc[-1]
<gemmi.cif.Block ZZZ>
>>> doc['MSE']
<gemmi.cif.Block MSE>
```

It has two other ways of accessing a block:

```
>>> # The function block.find_block(name) is like block[name] ...
>>> doc.find_block('MSE')
<gemmi.cif.Block MSE>
>>> # ... except when the block is not found:
>>> doc.find_block('no such thing')  # -> None
>>> # doc['no such thing'] # -> KeyError

>>> # Get the only block; throws exception if the document has more blocks.
>>> cif.read("../tests/1pfe.cif.gz").sole_block()
<gemmi.cif.Block 1PFE>
```

Blocks can be inserted (by default – appended after existing blocks) using one of the two functions:

- `Document.add_new_block(name, pos=-1)`
- `Document.add_copied_block(block, pos=-1)`

As an example, here is how to start a new document:

```
>>> d = cif.Document()
>>> block_one = d.add_new_block('block-one')
>>> # populate block_one
```

To delete a block use `Document.__delitem__` (for example: `del doc[1]`).

Document has also one property

```
>>> doc.source
'components.cif'
```

1.2.8 Block

Each block has a name and a list of items. Each item is one of:

- name-value pair (Pair),
- table, a.k.a loop (Loop)
- or save frame (Block – the same data structure as for block).

C++

Each block contains:

```
std::string name;
std::vector<Item> items;
```

where `Item` is implemented as an unrestricted (C++11) union that holds one of Pair, Loop or Block.

Python

Each block has a name:

```
>>> doc = cif.read("../tests/1pfe.cif.gz")
>>> block = doc.sole_block()
>>> block.name
'1PFE'
```

and a list of items (class Item):

```
>>> for item in block:
...     if item.line_number > 1670:
...         if item.pair is not None:
...             print('pair', item.pair)
...         elif item.loop is not None:
...             print('loop', item.loop)
...         elif item.frame is not None:
...             print('frame', item.frame)
...
pair ['_ndb_struct_conf_na.entry_id', '1PFE']
pair ['_ndb_struct_conf_na.feature', "'double helix'"]
loop <gemmi.cif.Loop 8 x 25>
loop <gemmi.cif.Loop 5 x 43>
loop <gemmi.cif.Loop 3 x 3>
loop <gemmi.cif.Loop 83 x 10>
```

1.2.9 Frame

(Very few people need it, skip this section.)

The *named save frames* (keyword `save_`) from the STAR specification are used in CIF files only as sub-sections of a block. The only place where they are enountered are mmCIF dictionaries.

The save-frame is stored as `Block` and can be accessed with:

```
Block* Block::find_frame(std::string name)
```

```
>>> frame = block.find_frame('my_frame')
```

Additionally, in C++ one may iterate over all Block's items, check each item type and handle all the save frames:

```
for (cif::Item& item : block.items)
  if (item.type == cif::ItemType::Frame)
    // doing something with item.frame which is a (nested) Block
    cif::Block& frame = item.frame;
```

1.2.10 Pairs and Loops

The functions in this section can be considered low-level, because they are specific to either name-value pairs or to loops.

Warning: Assuming what is a pair and what is in loop is a common source of bugs when handling mmCIF files, so instead of these functions we recommend using abstractions introduced in the next sections. For example, when working with proteins one could assume that anisotropic ADP values are in a loop, but wwPDB has entries with anisotropic ADP for one atom only – as name-value pairs. On the other hand, one could think that R-free is always given as name-value, but in entries from joint X-ray and neutron refinement it is in a loop. Be careful.

The next sections introduce function that work with both pairs and loops.

C++

Pair is simply defined as:

```
using Pair = std::array<std::string, 2>;
```

A pair with a particular tag can be located using:

```
const Pair* Block::find_pair(const std::string& tag) const
```

or, if you want just the value:

```
const std::string* Block::find_value(const std::string& tag) const
```

Both functions return `nullptr` if the tag is not found (and they do not search in CIF loops).

To add a pair to the block, or modify an existing one, use:

```
void Block::set_pair(const std::string& tag, std::string value)
```

The value needs quoting, the passed argument needs to be already quoted (you may pass `cif::quote(value)`).

Loop is defined as:

```
struct Loop {
  std::vector<std::string> tags;
  std::vector<std::string> values;
  // and a number of functions
};
```

To get values corresponding to a tag in a loop (table) you may use:

```
Column Block::find_loop(const std::string& tag)
```

`struct Column`, which is documented further on, has method `get_loop()` which gives access to `struct Loop`.

A new loop can be added using function:

```
Loop& Block::init_loop(const std::string& prefix, std::vector<std::string> tags)
```

Then it can be populated by either setting directly `tags` and `values`, or my using Loop's methods such as `add_row()` or `set_all_values()`.

Python

Accessing name-value pairs:

```
>>> # (1) tag and value
>>> block.find_pair('_cell.length_a')
['_cell.length_a', '39.374']
>>> block.find_pair('_no_such_tag')  # return None

>>> # (2) only value
>>> block.find_value('_cell.length_b')
'39.374'
>>> block.find_value('_cell.no_such_tag')  # returns None

>>> # (3) Item
>>> item = block.find_pair_item('_cell.length_c')
>>> item.pair
['_cell.length_c', '79.734']
>>> item.line_number
72
>>> block.find_pair_item('_nothing')  # return None
```

To add a name-value pair, replacing current item if it exists, use function `set_pair`:

```
>>> block.set_pair('_year', '2030')
```

If the value needs quoting, it must be passed quoted:

```
>>> block.set_pair('_title', cif.quote('Goldilocks and the Three Bears'))
```

Now we can create a CIF file can from scratch:

```
>>> d = cif.Document()
>>> d.add_new_block('oak')
<gemmi.cif.Block oak>
>>> _.set_pair('_nut', 'acorn')
>>> print(d.as_string().strip())
data_oak
_nut acorn
```

To access values in loop:

```
>>> block.find_loop('_atom_type.symbol')
<gemmi.cif.Column _atom_type.symbol length 6>
>>> list(_)
['C', 'CL', 'N', 'O', 'P', 'S']
```

To add a row to an existing table (loop) use `add_row`:

```
>>> loop = block.find_loop('_atom_type.symbol').get_loop()
>>> loop.add_row(['Au'], pos=0)
>>> loop.add_row(['Zr'])  # appends
>>> list(block.find_loop('_atom_type.symbol'))
['Au', 'C', 'CL', 'N', 'O', 'P', 'S', 'Zr']
```

`set_all_values` sets all the data in a table. It takes as an argument a list of lists of string. The lists of strings correspond to columns.

```
>>> loop = block.find_loop('_citation_author.citation_id').get_loop()
>>> loop.tags
['_citation_author.citation_id', '_citation_author.name', '_citation_author.ordinal']
>>> loop.set_all_values([['primary']*2, [cif.quote('Alice A.'), cif.quote('Bob B.')],␣
↪['1', '2']])
>>> for row in block.find_mmcif_category('_citation_author.'):
...   print(list(row))
['primary', "'Alice A.'", '1']
['primary', "'Bob B.'", '2']
```

To add a new loop (replacing old one if it exists) use `init_loop`:

```
>>> loop = block.init_loop('_ocean_', ['id', 'name'])
>>> # empty table is invalid in CIF, we need to add something
>>> loop.add_row(['1', cif.quote('Atlantic Ocean')])
```

In the above example, if the block already has tags `_ocean_id` and/or `_ocean_name` and

- if they are in a table: the table will be cleared and re-used,
- if they are in name-value pairs: the pairs will be removed and a table will be created at the position of the first pair.

To reorder items, use `block.move_item(old_pos, new_pos)`. The current position of a tag can be obtained using `block.get_index(tag)`.

```
>>> block.get_index('_entry.id')
0
>>> block.get_index('_ocean_id')
385
>>> block.move_item(0, -1)  # move first item to the end
>>> block.get_index('_entry.id')
385
>>> block.get_index('_ocean_id')
384
>>> block.move_item(385, 0)  # let's move it back (385 == -1 here)
```

1.2.11 Column

`Column` is a lightweight proxy class for working with both loop columns and name-value pairs.

It was returned from `find_loop` above, but it is also returned from a more general function `find_values()`, which searches for a given tag in both loops and pairs.

C++

The C++ signature of `find_values` is:

```
Column Block::find_values(const std::string& tag)
```

`Column` has a few member functions:

```
// Number of rows in the loop. 0 means that the tag was not found.
int length() const;

// Returns pointer to the column name in the DOM.
// The name is the same as argument to find_loop() or find_values().
std::string* get_tag();

// Returns underlying Item (which contains either Pair or Loop).
Item* item();

// Returns pointer to the DOM structure containing the whole table.
Loop* get_loop() const;

// Get raw value (no bounds checking).
std::string& operator[](int n);

// Get raw value (after bounds checking).
std::string& at(int n);

// Short-cut for cif::as_string(column.at(n)).
std::string str(int n) const;
```

`Column` also provides support for C++11 range-based `for`:

```
// mmCIF _chem_comp_atom is usually a table, but not always
for (const std::string &s : block.find_values("_chem_comp_atom.type_symbol"))
  std::cout << s << std::endl;
```

If the column represents a name-value pair, `Column::get_loop()` return `nullptr` (and `Column::get_length()` returns 1).

Python

```
>>> block.find_values('_cell.length_a')  # name-value pair
<gemmi.cif.Column _cell.length_a length 1>
>>> block.find_values('_atom_site.Cartn_x')  # column in a loop
<gemmi.cif.Column _atom_site.Cartn_x length 342>
```

Column's special method `__bool__` tells if the tag was found. `__len__` returns the number of corresponding values. `__iter__`, `__getitem__` and `__setitem__` get or set a raw string (i.e. string with quotes, if applicable).

As an example, let us shift all the atoms by 1A in the x direction:

```
>>> col_x = block.find_values('_atom_site.Cartn_x')
>>> for n, x in enumerate(col_x):
...   col_x[n] = str(float(x) + 1)
```

To get the actual string content (without quotes) one may use the method `str`:

```
>>> column = block.find_values('_chem_comp.formula')
>>> column[7]
"'H2 O'"
>>> column.str(7)  # short-cut for cif.as_string(column[7])
'H2 O'
```

If the tag is found in a loop, method `get_loop` returns a reference to this `Loop` in the DOM. Otherwise it returns `None`.

```
>>> column.get_loop()
<gemmi.cif.Loop 12 x 7>
```

1.2.12 Table

Usually we want to access multiple columns at once, so the library has another abstraction (`Table`) that can be used with multiple tags.

`Table` is returned by `Block.find()`. Like column, it is a lightweight, iterable view of the data, but it is for querying multiple related tags at the same time.

The first form of `find()` takes a list of tags:

```
Table Block::find(const std::vector<std::string>& tags)
```

```
>>> block.find(['_entity_poly_seq.entity_id', '_entity_poly_seq.num', '_entity_poly_
↪seq.mon_id'])
<gemmi.cif.Table 18 x 3>
```

Since tags in one loop tend to have a common prefix (category name), the library provides also a second form that takes the common prefix as the first argument:

```
Table Block::find(const std::string& prefix, const std::vector<std::string>& tags)

// These two calls are equivalent:

block.find({"_entity_poly_seq.entity_id", "_entity_poly_seq.num", "_entity_poly_seq.
↪mon_id"});
block.find("_entity_poly_seq.", {"entity_id", "num", "mon_id"});
```

```
>>> block.find('_entity_poly_seq.', ['entity_id', 'num', 'mon_id'])
<gemmi.cif.Table 18 x 3>
```

Note that `find` is not aware of dictionaries and categories, therefore the category name should end with a separator (dot for mmCIF files, as shown above).

In the example above, all the tags are required. If one of them is absent, the returned `Table` is empty. Tags (all except the first one) can be marked as *optional* by adding prefix `?`:

```
Table table = block.find({"_required_tag", "?_optional_tag"})
```

```
>>> table = block.find(['_required_tag', '?_optional_tag'])
```

In such case the returned table may contain either one or two columns. Before accessing column corresponding to an optional tag one must check if the column exists with `Table::has_column()` (or, alternatively, with equivalent function `Table::Row::has()` which will be introduced later):

```
bool Table::has_column(int n) const
```

```
>>> block.find('_entity_poly_seq.', ['entity_id', '?num', '?bleh'])
<gemmi.cif.Table 18 x 3>
>>> _.has_column(0), _.has_column(1), _.has_column(2)
(True, True, False)
```

The `Table` has functions to check its shape:

```
bool ok() const;  // true if the table is not empty
size_t width() const;  // number of columns
size_t length() const;  // number of rows
```

```
>>> table = block.find('_entity_poly_seq.', ['entity_id', 'num', 'mon_id'])
>>> # instead of ok() in Python we use __bool__()
>>> assert table, "table.__bool__() is expected to return True"
>>> table.width()  # number of columns
3
>>> len(table)  # number of rows
18
```

If Table's data is in Loop, the Loop class can be accessed using:

```
Loop* get_loop();  // nullptr for tag-value pairs
```

```
>>> table.loop     # None for tag-value pairs
<gemmi.cif.Loop 18 x 4>
```

If a prefix was specified when calling find, the prefix length is stored and the prefix can be retrived:

```
size_t prefix_length;
std::string get_prefix() const;
```

```
>>> table.prefix_length
17
>>> table.get_prefix()
'_entity_poly_seq.'
```

Table also has function `erase()` that deletes all tags and data associated with the table. See an example in the *CCD section* below.

Row-wise access

Most importantly, `Table` provides access to rows (`Table::Row`) that in turn provide access to value strings:

```
Row Table::operator[](int n)  // access Row
Row Table::at(int n)          // the same but with bounds checking
// and also begin(), end() and iterator that enable range-for.

// Returns the first row that has the specified string in the first column.
Row Table::find_row(const std::string& s)

// Makes sure that the table has only one row and returns it.
Row Table::one()
```

```
>>> table[0]
<gemmi.cif.Table.Row: 1 1 DG>

>>> # and of course it's iterable
>>> for row in table: pass

>>> # Returns the first row that has the specified string in the first column.
>>> table.find_row('2')
<gemmi.cif.Table.Row: 2 1 DSN>
```

as well as to the tags:

```
Row tags();  // pseudo-row that contains tags
```

```
>>> table.tags
<gemmi.cif.Table.Row: _entity_poly_seq.entity_id _entity_poly_seq.num _entity_poly_
↪seq.mon_id>
```

`Table::Row` has functions for accessing the values:

```
// Get raw value.
std::string& Table::Row::operator[](int n)  // no bounds checking
std::string& Table::Row::at(int n)          // with bounds checking
// and also begin(), end(), iterator, const_supports iterators.
```

```
>>> for row in table: print(row[-1], end=',')
DG,DC,DG,DT,DA,DC,DG,DC,DSN,ALA,N2C,NCY,MVA,DSN,ALA,NCY,N2C,MVA,
>>>
>>> row = table[9]
>>> for value in row: print(value, end=',')
2,2,ALA,
>>> row[2]
'ALA'
>>> row[-1]  # the same
'ALA'
>>> row.get(2) # the same, but returns None instead of IndexError
'ALA'
>>> row['mon_id']  # the same, but slightly slower than numeric index
'ALA'
>>> row['_entity_poly_seq.mon_id']  # the same
'ALA'
```

and a few convenience functions, including:

```
size_t Table::Row::size() const           // the width of the table
std::string Table::Row::str(int n) const  // short-cut for cif::as_string(row.at(n))
bool Table::Row::has(int n) const         // the same as Table::has_column(n)
```

```
>>> len(row)     # the same as table.width()
3
>>> row.str(2)  # if the value is in quotes, it gets un-quoted
'ALA'
>>> row.has(2)  # the same as table.has_column(2)
True
```

The tags and value can be modified. As an example, let us swap two tag names (these two tend to have identical values, so no one will notice):

```
  cif::Table::Row tags = block.find("_atom_site.", {"label_atom_id", "auth_atom_id"}).
↪tags();
  std::swap(tags[0], tags[1]);
```

```
>>> tags = block.find('_atom_site.', ['label_atom_id', 'auth_atom_id']).tags
>>> tags[0], tags[1] = tags[1], tags[0]
```

Column-wise access

`Table` gives also access to columns, represented by the previously introduced `Column`:

```
Column Table::column(int index)

// alternatively, specify tag name
Column Table::find_column(const std::string& tag)
```

```
>>> table.column(0)
<gemmi.cif.Column _entity_poly_seq.entity_id length 18>
>>> table.find_column('_entity_poly_seq.mon_id')
<gemmi.cif.Column _entity_poly_seq.mon_id length 18>
```

If the table is created in a function that uses prefix, the prefix can be omitted in `find_column`:

```
Table t = block.find("_entity_poly_seq.", {"entity_id", "num", "mon_id"});
Column col = t.find_column(2);
// is equivalent to
Column col = t.find_column("_entity_poly_seq.mon_id");
// is equivalent to
Column col = t.find_column("mon_id");
```

```
>>> table.find_column('mon_id')
<gemmi.cif.Column _entity_poly_seq.mon_id length 18>
```

C++ note: both `Column` and `Table::Row` have functions `begin()` and `end()` in const and non-const variants, returning `iterator` and `const_iterator` types, respectively. These types satisfy requirements of the BidirectionalIterator concept. Conversely, the iterator over the rows of `Table` is a minimalistic structure – just enough get the range-for work.

mmCIF categories

mmCIF files group data into categories. All mmCIF tags have a dot (e.g. `_entry.id`) and the category name is the part before the dot.

C++

We have two functions to work with categories. One returns a list of all categories in the block:

```
std::vector<std::string> Block::get_mmcif_category_names() const
```

The other returns a `Table` with all tags (and values) belonging to the specified category:

```
Table Block::find_mmcif_category(std::string cat)
```

Python

Python bindings have the same two functions:

```
>>> block.get_mmcif_category_names()[:3]
['_entry.', '_audit_conform.', '_database_2.']
>>> block.find_mmcif_category('_entry.')
<gemmi.cif.Table 1 x 1>
>>> _.tags[0], _[0][0]
('_entry.id', '1PFE')
>>> cat = block.find_mmcif_category('_database_2.')
>>> cat
<gemmi.cif.Table 4 x 2>
>>> list(cat.tags)
['_database_2.database_id', '_database_2.database_code']
>>> for row in cat:
...   print('%s: %s' % tuple(row))
PDB: 1PFE
NDB: DD0057
RCSB: RCSB019291
WWPDB: D_1000019291
>>> cat[3][1]
'D_1000019291'
```

Additionally, two Python-specific functions: `get_mmcif_category` and `set_mmcif_category` translate between an mmCIF category and Python dictionary:

```
>>> block.get_mmcif_category('_entry.')
{'id': ['1PFE']}
>>> sorted(block.get_mmcif_category('_database_2.').keys())
['database_code', 'database_id']
```

`?` and `.` are translated to `None` and `False`. To disable this translation and get "raw" strings, add argument `raw=True`.

```
>>> default = block.get_mmcif_category('_software')
>>> raw = block.get_mmcif_category('_software', raw=True)
>>> for name in ['name', 'version', 'citation_id']:
...     print(default[name][0], raw[name][0])
...
EPMR EPMR
False .
None ?
```

`set_mmcif_category()` takes a category name and a dictionary (that maps tags to lists) and creates or replaces the corresponding category:

```
>>> ocean_data = {'id': [1, 2, 3], 'name': ['Atlantic', 'Pacific', 'Indian']}
>>> block.set_mmcif_category('_ocean', ocean_data)
```

It also takes the optional `raw` parameter (default: `False`). Specify `raw=True` if the values are already quoted, otherwise they will be quoted twice.

```
>>> block.set_mmcif_category('_raw', {'a': ['?', '.', "'a b'"]}, raw=True)
>>> list(block.find_values('_raw.a'))
['?', '.', "'a b'"]
>>> block.set_mmcif_category('_nop', {'a': ['?', '.', "'a b'"]}, raw=False)
>>> list(block.find_values('_nop.a'))
["'?'", "'.'", '"\'a b\'"']
>>> block.set_mmcif_category('_ok', {'a': [None, False, 'a b']}, raw=False)
```

(continues on next page)

(continued from previous page)

```
>>> list(block.find_values('_ok.a'))
['?', '.', "'a b'"]
```

Finally, we have a variant of `init_loop` for working with mmCIF categories:

```
>>> loop = block.init_mmcif_loop('_ocean.', ['id', 'name'])
>>> loop.add_row(['1', 'Atlantic'])
```

The subtle difference from generic `init_loop` is that if the block has other name-value pairs in the same category (say, `_ocean.depth 8.5`) `init_loop` leaves them untouched, but `init_mmcif_loop` removes them. Additionally, like in other `_mmcif_` functions, the trailing dot in the category name may be omitted (but the leading underscore is required).

1.2.13 JSON

`cif::Document` can be stored in a JSON format, and it can be read from JSON file. This is in general true about CIF files - their content can be converted to JSON and back. In gemmi we have a number of options to customize the translation. In particular, both mmCIF and CIF-JSON flavours are supported. More details about the flavours are given in the description of *gemmi convert*.

C++

Header `gemmi/to_json.hpp` provides code for serializing `cif::Document` as JSON.

Such JSON files can be read back into the `cif::Document` structure using function from `gemmi/json.hpp`.

Python

`Document.as_json()` returns the document serialized in JSON string. To output mmJSON add argument `mmjson=True`.

mmJSON (possibly gzipped) can be read using function `cif.read_mmjson()`. In addition, the function `cif.read()` will also read mmJSON format if the file name ends with `.json` or `.json.gz`.

1.2.14 Design choices

Parser

Parsing formal languages is a well-researched topic in computer science. The first versions of lex and yacc - popular tools that generate lexical analyzers and parsers - were written in 1970's. Today many tools exist to translate grammar rules into C/C++ code. On the other hand many compilers and high-profile tools use hand-coded parsers - as it is more flexible.

Looking at other STAR and CIF parsers – some use parser generators (COD::CIF::Parser and starlib2 use yacc/bison, iotbx.cif uses Antlr3), others are hand-coded.

I had experience with flex/bison and Boost.Spirit (and I wanted to try also Lemon and re2c) but I decided to use PEGTL for this task (and I'm very happy with this choice). I was convinced by the TAOC++ JSON parser that is based on PEGTL and has a good balance of simplicity and performance.

PEGTL is a C++ library (not a generator) for creating PEG parsers. PEG stands for Parsing Expression Grammar – a simpler approach than the traditional Context Free Grammar.

As a result, our parser depends on a third-party (header-only) library, but the parser itself is pretty simple.

Data structures

The next thing is how we store the data read from file. We decided to rely on the C++ standard library where we can.

Generally, storage of such data involves (in C++ terms) some containers and a union/variant type for storing values of mixed types.

We use primarily `std::vector` as a container.

A custom structure `Item` with (unrestricted) union is used as a variant-like class.

Strings are stored in `std::string` and it is fast enough. Mainstream C++ standard libraries have short string optimization (SSO) for up to 15 or 22 characters, which covers most of the values in mmCIF files.

1.2.15 Performance

Gemmi has the fastest open-source CIF parser (at least in the hands of the author). While further improvement would be possible (some JSON parsers are much faster and parsing CIF and JSON is not that different), it is not a priority.

1.2.16 Directory walking

Many of the utilities and examples developed for this project work with archives of CIF files such as wwPDB or COD. To make it easier to iterate over all CIF files in a directory tree we provide a class `CifWalk`.

C++

```
#include <gemmi/dirwalk.hpp>

// ...
// throws std::runtime_error if top_dir doesn't exist
for (const std::string& cif_file : gemmi::CifWalk(top_dir)) {
  cif::Document doc = cif::read(gemmi::MaybeGzipped(cif_file));
  // ...
}
```

This header file contains also a more general `DirWalk` class, and classes specific to macromolecular files (`PdbWalk`, `MmCifWalk`, `CoorFileWalk`). The file type of each file is guessed from the file name.

Python

Since Python comes with the os.walk() function for iterating over files and directories, this functionality is less important here. Anyway, we provide bindings for CifWalk:

```
>>> import gemmi
>>> list(gemmi.CifWalk('../tests/'))[:2]
['../tests/1011031.cif', '../tests/1pfe.cif.gz']
```

We also have Python bindings for `CoorFileWalk` that picks macromolecular coordinate files.

When the user has no permission to read one of the traversed directories, the functions above raise an error (std::runtime_error / RuntimeError).

All these directory walking functions are powered by the tinydir library (a single-header library copied into `include/gemmi/third_party`).

1.2.17 Examples

The examples here use C++11 or Python 2.7/3.x. Full working code code can be found in the examples directory.

If you have the Python `gemmi` module installed you should also have the Python examples – try `python -m gemmi-examples` if not sure where they are.

The examples below can be run on one or more PDBx/mmCIF files. The ones that perform PDB-wide analysis are meant to be run on a local copy of the mmCIF archive (30GB+ gzipped, don't uncompress!).

mmCIF to XYZ

To start with something simple, let us convert mmCIF to the XYZ format:

```
#include <iostream>
#include <gemmi/cif.hpp>
#include <gemmi/util.hpp>  // for gemmi::join_str
//#include <boost/algorithm/string/join.hpp>  // for boost::algorithm::join

void convert_to_xyz(cif::Document doc) {
  cif::Table atoms = doc.sole_block().find("atom_site_.",
                           {"type_symbol", "Cartn_x", "Cartn_y", "Cartn_z"});
  std::cout << atoms.length() << "\n\n";
  for (auto row : atoms)
    std::cout << gemmi::join_str(row, " ") << "\n";
    //std::cout << boost::algorithm::join(row, " ") << "\n";
}
```

auth vs label

When you look at the list of atoms (`_atom_site.*`) in mmCIF files some columns seem to be completely redundant. Are they?

It is hard to manually find an example where `_atom_site.auth_atom_id` differs from `_atom_site.label_atom_id`, or where `_atom_site.auth_comp_id` differs from `_atom_site.label_comp_id`. So here is a small C++ program:

```
// Compare pairs of columns from the _atom_site table.
// Compiled with: g++-6 -O2 -Iinclude auth_label.cc -lstdc++fs -lz
#include <gemmi/gz.hpp>
#include <gemmi/cif.hpp>
#include <iostream>
#include <experimental/filesystem>  // just <filesystem> in C++17

namespace fs = std::experimental::filesystem;
namespace cif = gemmi::cif;

void print_differences(cif::Block& block, const std::string& name) {
  for (auto row : block.find("_atom_site.", {"auth_" + name, "label_" + name}))
```

(continues on next page)

(continued from previous page)

```
    if (row[0] != row[1])
      std::cout << block.name << ": " << name << "  "
                << row[0] << " -> " << row[1] << std::endl;
}

bool ends_with(const std::string& str, const std::string& suffix) {
  size_t sl = suffix.length();
  return str.length() >= sl && str.compare(str.length() - sl, sl, suffix) == 0;
}

int main(int argc, char* argv[]) {
  if (argc != 2)
    return 1;
  for (auto& p : fs::recursive_directory_iterator(argv[1])) {
    std::string path = p.path().u8string();
    if (ends_with(path, ".cif") || ends_with(path, ".cif.gz")) {
      cif::Document doc = cif::read(gemmi::MaybeGzipped(path));
      // What author's atom names were changed?
      print_differences(doc.sole_block(), "atom_id");
      // What author's residue names were changed?
      print_differences(doc.sole_block(), "comp_id");
    }
  }
  return 0;
}
```

We compile it, run it, and come back after an hour:

```
$ g++-6 -O2 -Iinclude examples/auth_label.cpp -lstdc++fs -lz
$ ./a.out pdb_copy/mmCIF
3D3W: atom_id  O1 -> OD
1TNI: atom_id  HN2 -> HN3
1S01: comp_id  CA -> UNL
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
2KI7: atom_id  H -> H1
4E1U: atom_id  H101 -> H103
4WH8: atom_id  H3 -> H391
4WH8: atom_id  H4 -> H401
3V2I: atom_id  UNK -> UNL
```

(continues on next page)

(continued from previous page)

```
1AGG: atom_id  H3 -> H
1AGG: atom_id  H3 -> H
1AGG: atom_id  H3 -> H
```

So, as of April 2017, only a single author's residue name was changed, and atom names were changed in 7 PDB entries.

Amino acid frequency

After seeing a paper in which amino acid frequency was averaged over 5000+ proteomes (Ala 8.76%, Cys 1.38%, Asp 5.49%, etc) we may want to compare it with the frequency in the PDB database. So we write a little script

```
#!/usr/bin/env python
from __future__ import print_function
from collections import Counter
from gemmi import cif

# To keep this example small we moved handling of command-line args to util.py.
# When a directory is given as an argument get_file_paths_from_args()
# yields all the cif(.gz) paths under this directory.
from util import get_file_paths_from_args

# Check amino-acid frequency in the PDB database
totals = Counter()
for path in get_file_paths_from_args():
    # read file (uncompressing on the fly) and get the only block
    block = cif.read(path).sole_block()
    # find table with the sequence
    seq = block.find('_entity_poly_seq.', ['entity_id', 'mon_id'])
    # convert table with chain types (protein/DNA/RNA) to dict
    entity_types = dict(block.find('_entity_poly.', ['entity_id', 'type']))
    # and count these monomers that correspond to a protein chain
    aa_counter = Counter(row.str(1) for row in seq
                         if 'polypeptide' in entity_types[row.str(0)])
    totals += aa_counter
    # print residue counts for each file
    print(block.name, *('%s:%d' % c for c in aa_counter.most_common()))
# finally, print the total counts as percentages
f = 100.0 / sum(totals.values())
print('TOTAL', *('%s:%.2f%%' % (m, c*f) for (m, c) in totals.most_common(20)))
```

We can run this script on our copy of the PDB database (mmCIF format) and get such an output:

```
200L ALA:17 LEU:15 LYS:13 ARG:13 THR:12 ASN:12 GLY:11 ILE:10 ASP:10 VAL:9 GLU:8 SER:6␣
↪TYR:6 GLN:5 PHE:5 MET:5 PRO:3 TRP:3 HIS:1
...
4ZZZ LEU:40 LYS:33 SER:30 ASP:25 GLY:25 ILE:25 VAL:23 ALA:19 PRO:18 GLU:18 ASN:17␣
↪THR:16 TYR:16 GLN:14 ARG:11 PHE:10 HIS:9 MET:9 CYS:2 TRP:2
TOTAL LEU:8.90% ALA:7.80% GLY:7.34% VAL:6.95% GLU:6.55% SER:6.29% LYS:6.18% ASP:5.55%␣
↪THR:5.54% ILE:5.49% ARG:5.35% PRO:4.66% ASN:4.16% PHE:3.88% GLN:3.77% TYR:3.45%␣
↪HIS:2.63% MET:2.14% CYS:1.38% TRP:1.35%
```

On my laptop it takes about an hour, using a single core. Most of this hour is spent on tokenizing the CIF files and copying the content into a DOM structure, what could be largely avoided given that we use only sequences not atoms. But it is not worth to optimize one-off scripts. The same goes for using multiple processor cores.

Custom PDB search

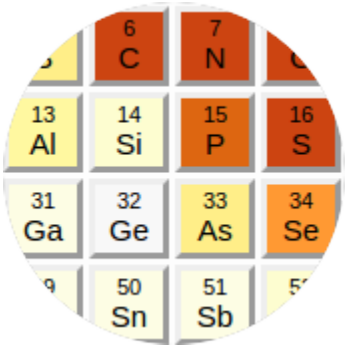
We may need to go through a local copy of the PDB archive to find entries according to criteria that cannot be queried in RCSB/PDBe/PDBj web interfaces. For example, if for some reason we need PDB entries with large number of anisotropic B-factors, we may write a quick script:

```
#!/usr/bin/env python
"Find PDB entries with more than 50,000 anisotropic B-factors."

from __future__ import print_function
from gemmi import cif
from util import get_file_paths_from_args

for path in get_file_paths_from_args():
    block = cif.read(path).sole_block()
    anis = block.find_values("_atom_site_anisotrop.id")
    if len(anis) > 50000:
        print(block.name, len(anis))
```

and then run it for an hour or so.

```
$ ./examples/simple_search.py pdb_copy/mmCIF
5AEW 60155
5AFI 98325
3AIB 52592
...
```

Search PDB by elements



Let say we want to be able to search the PDB by specifying a set of elements present in the model. First we write down elements present in each PDB entry:

```
block = cif.read(path).sole_block()
elems = set(block.find_loop("_atom_site.type_symbol"))
print(name + ' ' + ' '.join(elems))
```

This example ended up overdone a bit and it was put into a separate repository.

Here is a demo: https://project-gemmi.github.io/periodic-table/

Solvent content vs resolution

Let say that we would like to generate a plot of solvent content as a function of $d_{\min}$, similar to the plots by C. X. Weichenberger and B. Rupp in Acta Cryst D and CNN), but less smoothed and with duplicate entries included.

In macromolecular crystallography solvent content is conventionally estimated as:

$$V_S = 1 - 1.230 / V_M$$

where V_M is Matthews coefficient defined as V_M=V/m (volume of the asymmetric unit over the molecular weight of all molecules in this volume).

Weichenberger and Rupp calculated V_S themselves, but we will simply use the values present in mmCIF files. So first we extract V_M, V_S and $d_{\min}$, as well as number of DNA/RNA chains (protein-only entries will have 0 here), deposition date and group ID (which will be explained later).

```
def gather_data():
    "read mmCIF files and write down a few numbers (one file -> one line)"
    writer = csv.writer(sys.stdout, dialect='excel-tab')
    writer.writerow(['code', 'na_chains', 'vs', 'vm', 'd_min', 'date', 'group'])
    for path in util.get_file_paths_from_args():
        block = cif.read(path).sole_block()
        code = cif.as_string(block.find_value('_entry.id'))
        na = sum('nucleotide' in t[0]
                 for t in block.find_values('_entity_poly.type'))
        vs = block.find_value('_exptl_crystal.density_percent_sol')
        vm = block.find_value('_exptl_crystal.density_Matthews')
        d_min = block.find_value('_refine.ls_d_res_high')
        dep_date_tag = '_pdbx_database_status.recvd_initial_deposition_date'
        dep_date = parse_date(block.find_values(dep_date_tag).str(0))
        group = block.find_value('_pdbx_deposit_group.group_id')
        writer.writerow([code, na, vs, vm, d_min, dep_date, group])
```

After a few checks (see `examples/matthews.py`) we may decide to use only those PDB entries in which V_M and V_S are consistent. To make things more interesting we plot separately depositions from before and after 1/1/2015.

Ripples in the right subplot show that in many entries V_S is reported as integer, so we should calculate it ourselves (like Weichenberger & Rupp) for better precision. Ripples in the top plot show that we should use a less arbitrary metric of resolution than $d_{\min}$ (but it's not so easy). Or we could just smooth them out by changing parameters of this plot.

On the left side of the yellow egg you can see dark stripes caused by *group depositions*, which were introduced by PDB in 2016. They came from two European high-throughput beamlines and serve as an illustration of how automated software can analyze hundreds of similar samples (fragment screening) and submit them quickly to the PDB.

We can easily filter out group depositions – either using the group IDs extracted from mmCIF files (we do this for pdb-stats) or, like W&B, by excluding redundant entries based on the unit cell and V_M.

Not all the dark spots are group depositions. For example, the one at V_S66.5%, $d_{\min}$ 2.5-3A is proteasome 20S studied over years by Huber *et al*, with dozens PDB submissions.

Weights

Let say we would like to verify consistency of molecular weights. First, let us look at chem_comp tables:

```
loop_
_chem_comp.id
_chem_comp.type
```

(continues on next page)

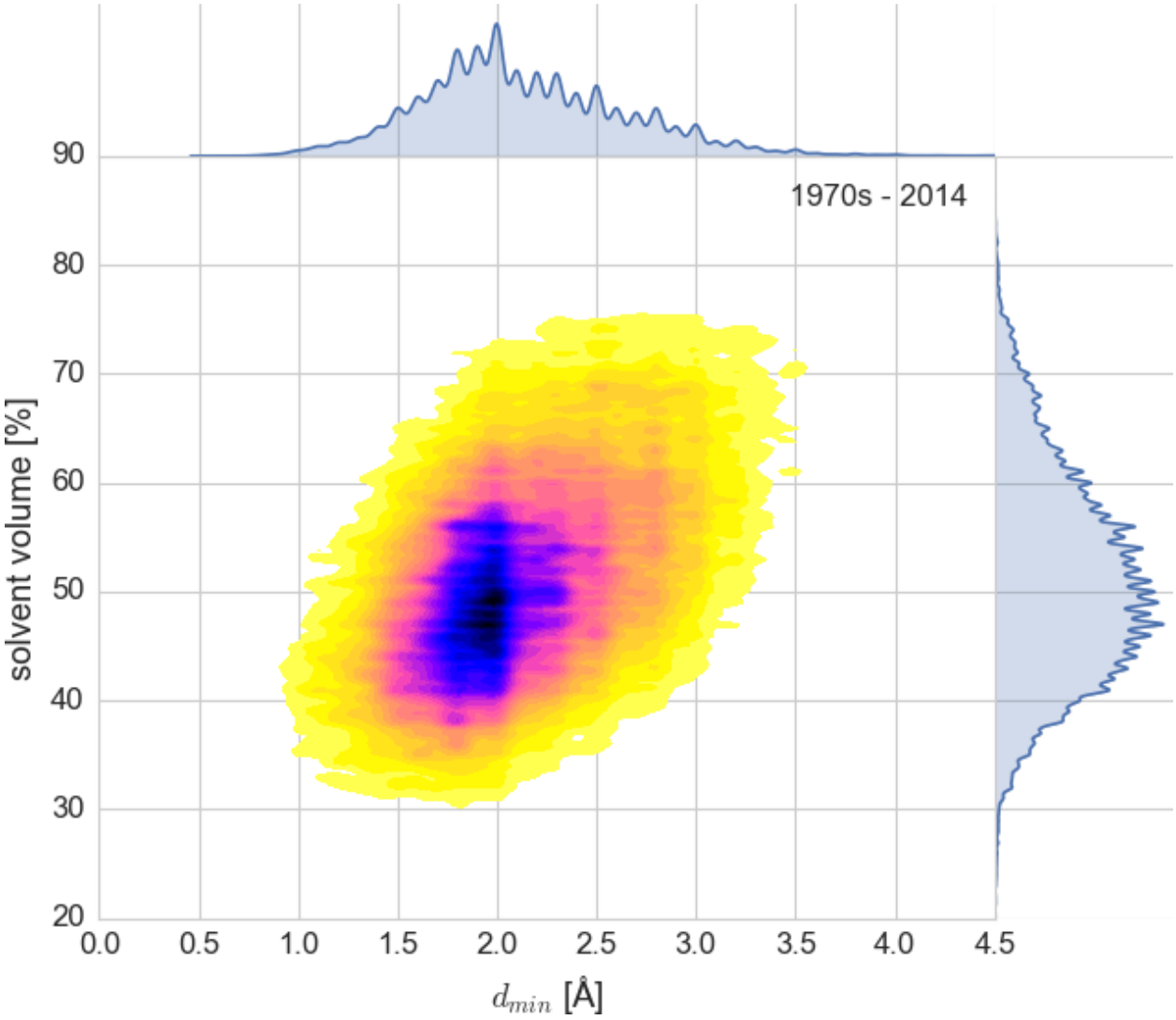


Fig. 1: About 90,000 PDB entries (up to 2014) plotted as intentionally undersmoothed kernel density estimate. The code used to produce this plot is in `examples/matthews.py`.

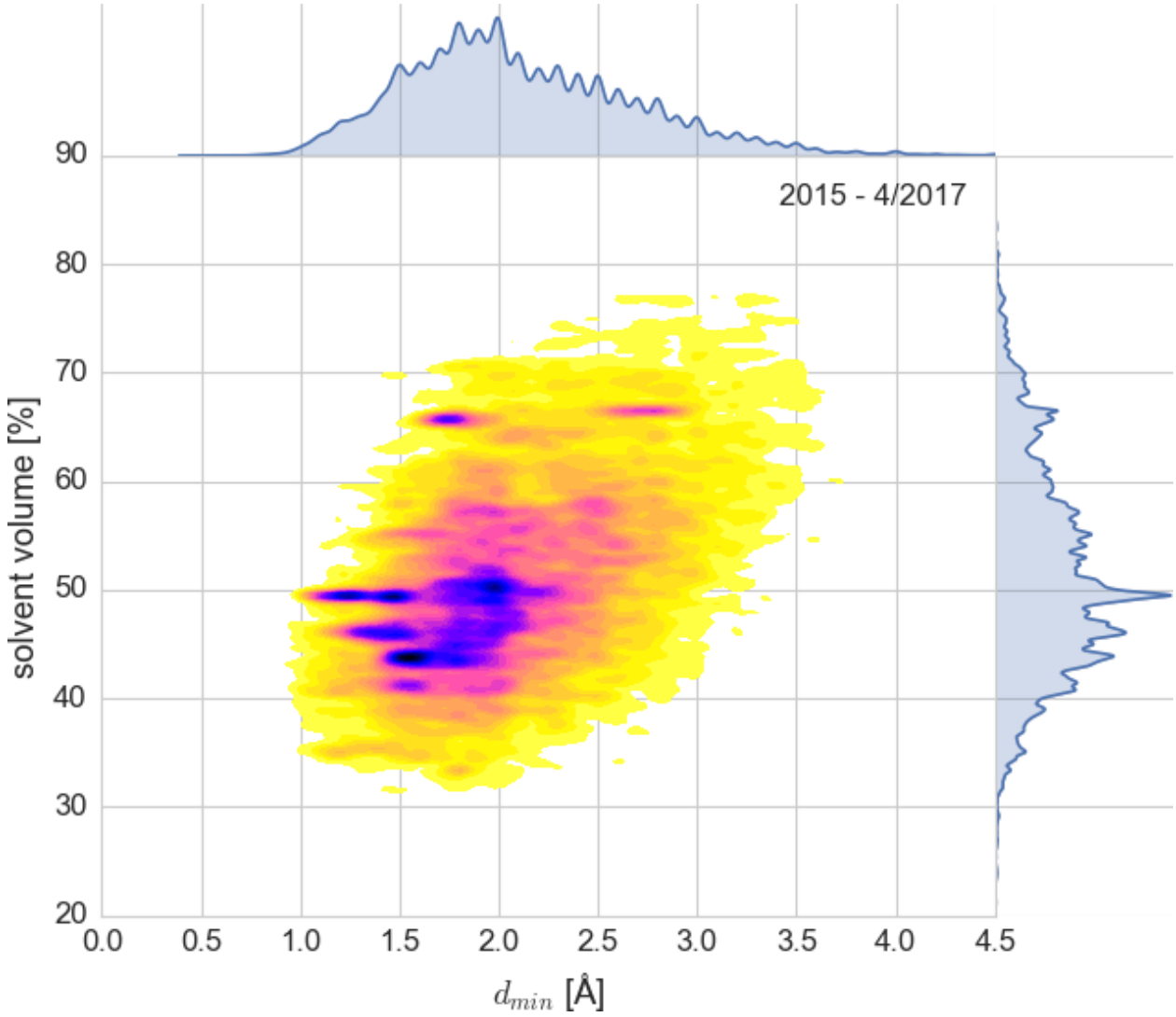


Fig. 2: About 17,000 PDB entries (2015 - April 2017) plotted as intentionally undersmoothed kernel density estimate. The code used to produce this plot is in `examples/matthews.py`.

(continued from previous page)

```
_chem_comp.mon_nstd_flag
_chem_comp.name
_chem_comp.pdbx_synonyms
_chem_comp.formula
_chem_comp.formula_weight
ALA 'L-peptide linking' y ALANINE         ?                 'C3 H7 N O2'     89.093
ARG 'L-peptide linking' y ARGININE        ?                 'C6 H15 N4 O2 1' 175.209
ASN 'L-peptide linking' y ASPARAGINE      ?                 'C4 H8 N2 O3'    132.118
ASP 'L-peptide linking' y 'ASPARTIC ACID' ?                 'C4 H7 N O4'     133.103
CYS 'L-peptide linking' y CYSTEINE        ?                 'C3 H7 N O2 S'   121.158
EDO non-polymer         . 1,2-ETHANEDIOL  'ETHYLENE GLYCOL' 'C2 H6 O2'       62.068
...
```

We expect that by using molecular weights of elements and a simple arithmetic we can recalculate `_chem_comp.formula_weight` from `_chem_comp.formula`. The full code is in `examples/weights.py`. It includes a function that converts `'C2 H6 O2'` to `{C:2, H:6, O:2}`. Here we only show the few lines of code that sum the element weights and compare the result:

```
def check_chem_comp_formula_weight(block):
    for cc in block.find('_chem_comp.', ['id', 'formula', 'formula_weight']):
        if cc.str(0) in ('UNX', 'UNL'):  # unknown residue or ligand
            continue
        fdict = util.formula_to_dict(cc.str(1).title())
        calc_weight = sum(n * Element(e).weight for (e, n) in fdict.items())
        diff = calc_weight - cif.as_number(cc[2])
        if not (abs(diff) < 0.1):  # also true if diff is NaN
            print('%s %s  %-16s % 9.3f - %9s = %+.3f' %
                  (block.name, cc[0], cc.str(1), calc_weight, cc[2], diff))
```

This script prints differences above 0.1 u:

```
3A1R DOD  D2 O               20.028 -    18.015 = +2.013
5AI2 D8U  D 1                 2.014 -         ? = +nan
5AI2 DOD  D2 O               20.028 -    18.015 = +2.013
4AR3 D3O  O 1                15.999 -    22.042 = -6.043
4AR4 D3O  O 1                15.999 -    22.042 = -6.043
4AR5 DOD  D2 O               20.028 -    18.015 = +2.013
4AR6 DOD  D2 O               20.028 -    18.015 = +2.013
4B1A K3G  MO12 O40 P 1     1822.350 -  1822.230 = +0.120
4BVO E43  H2 O40 W12 6     2848.072 -  2848.192 = -0.120
...
5FHW HFW  Hf O61 P2 W17    4341.681 -  4343.697 = -2.016
...
```

The differences are few and minor. We see a few PDB entries with the weight of D_2O set to the weight H_2O. The second line shows missing weight. The differences +0.12 and -0.12 next to Mo12 and W12 probably come from the 0.01u difference in the input masses of the elements. In two entries D3O is missing D in the formula, and -2.016 in HFW suggests two missing hydrogens.

Now let us try to re-calculate `_entity.formula_weight` from the chem_comp weights and the sequence. The PDB software calculates it as a sum of components in the chain, minus the weight of N-1 waters. In case of nucleic acids also PO_2 is subtracted (why not PO_3? – to be checked). And in case of microheterogeneity only the main conformer is taken into account. As the PDB software uses single precision for these computations, we ignore differences below 0.003%, which we checked to be enough to account for numerical errors.

```
def check_entity_formula_weight(block):
    # read chem_comp weights from the file, e.g. THR: 119.120
    cc_weights = {cc.str(0): cif.as_number(cc[1]) for cc in
                  block.find('_chem_comp.', ['id', 'formula_weight'])}

    # read polymer types from _entity_poly.type
    poly_types = {ep.str(0): ep.str(1) for ep in
                  block.find('_entity_poly.', ['entity_id', 'type'])}

    # read and sort sequences from _entity_poly_seq
    entity_seq = collections.defaultdict(list)
    for m in block.find('_entity_poly_seq.', ['entity_id', 'num', 'mon_id']):
        entity_seq[m.str(0)].append((cif.as_int(m[1]), cif.as_string(m[2])))
    for seq in entity_seq.values():
        seq.sort(key=lambda p: p[0])

    # read _entity.formula_weight values
    f_weights = block.find('_entity.', ['id', 'formula_weight'])
    entity_weights = {ent.str(0): cif.as_number(ent[1]) for ent in f_weights}

    # calculate weight from sequences and compare with _entity.formula_weight
    for ent, seq in entity_seq.items():
        # in case of microheterogeneity take the first conformer
        main_seq = [next(g) for _, g in itertools.groupby(seq, lambda p: p[0])]
        weight = sum(cc_weights[mon_id] for (num, mon_id) in main_seq)
        weight -= (len(main_seq) - 1) * H2O_MASS
        if 'ribonucleotide' in poly_types[ent]:  # DNA or RNA
            weight -= PO2_MASS
        diff = weight - entity_weights[ent]
        if abs(diff) > max(0.1, 3e-5 * weight):
            print('%4s entity_id: %2s %10.2f - %10.2f = %+8.3f' %
                  (block.name, ent, weight, entity_weights[ent], diff))
```

Running this script on a local copy of the PDB database prints 26 lines, and the difference is always (except for 4PMN) the mass of PO_2 showing that we have not fully reproduced the rule when to subtract this group.

```
...
4D67 entity_id: 44 1625855.77 - 1625917.62 =  -61.855
1HZS entity_id:  1    1733.88 -    1796.85 =  -62.973
2K4G entity_id:  1    2158.21 -    2221.18 =  -62.974
3MBS entity_id:  1    2261.35 -    2324.33 =  -62.974
1NR8 entity_id:  2    2883.07 -    2946.05 =  -62.974
3OK2 entity_id:  1    3417.14 -    3354.17 =  +62.968
...
```

Disulfide bonds

If we were curious what residues take part in disulfide bonds we could write a little script that inspects annotation in the _struct_conn category. But to show something else, here we will use `gemmi grep`, a little utility that is documented in a *separate section*.

First we try how to extract interesting data from a single entry:

```
$ gemmi grep --delimiter=' ' _struct_conn.conn_type_id \
> -a _struct_conn.ptnr1_label_comp_id -a _struct_conn.ptnr1_label_atom_id \
> -a _struct_conn.ptnr2_label_comp_id -a _struct_conn.ptnr2_label_atom_id \
```

(continues on next page)

(continued from previous page)

```
> 5CBL
5CBL disulf CYS SG BME S2
5CBL covale ILE CD1 BME C2
```

Then we pipe the output through Unix shell utilities. `| grep disulf` limits the output to the disulfide bonds. `| awk '{ print $3, $4 "\n" $5, $6 }'` changes each line into two; the first output line above becomes:

```
CYS SG
BME S2
```

Then we run it on the whole PDB archive, sort, count and print the statistics. The complete command is:

```
$ gemmi grep --delimiter=' ' _struct_conn.conn_type_id \
> -a _struct_conn.ptnr1_label_comp_id -a _struct_conn.ptnr1_label_atom_id \
> -a _struct_conn.ptnr2_label_comp_id -a _struct_conn.ptnr2_label_atom_id \
> /hdd/mmCIF \
> | grep disulf | awk '{ print $3, $4 "\n" $5, $6 }' \
> | sort | uniq -c | sort -nr
   367980 CYS SG
    274 DCY SG
     28 BME S2
     23 SO4 S
     19 CY3 SG
     14 MET SD
     13 MTN S1
     12 V1A S3
     10 MPT SG
      8 NCY SG
      7 LE1 SG
      7 CSX SG
      6 CSO SG
      5 GSH SG2
      5 DTT S1
      4 SC2 SG
      4 MRG S24
      3 H2S S
      3 DTT S4
      3 1WD S7
      2 P8S S1
      2 MTE S2'
      2 MTE S1'
      2 EPE S
      2 DHL SG
      2 DCD S1
      2 CSD SG
      2 COM S1
      2 6LN S2
      2 6LN S1
      2 4K3 S4
      2 3C7 S01
      2 2ON S2
      1 SCN S
      1 RXR SD
      1 RXR S10
      1 MEE S
      1 G47 SG
```

(continues on next page)

(continued from previous page)

```
      1 COA S1P
      1 6ML S1
      1 5O8 SBH
```

And what other bond types are annotated in `_struct_conn`?

```
$ gemmi grep -O -b _struct_conn.conn_type_id /hdd/mmCIF/ | sort | uniq -c
 501851 covale
 184231 disulf
4035115 hydrog
1394732 metalc
```

mmJSON-like data

Gemmi has a built-in support for mmJSON and comes with a *converter* utility, but just as an excercise let us convert mmJSON to mmCIF in Python:

```
import json
import sys
from gemmi import cif

file_in, file_out = sys.argv[1:]

with open(file_in) as f:
    json_data = json.load(f)
assert len(json_data) == 1  # data_1ABC
(block_name, block_data), = json_data.items()
assert block_name.startswith('data_')
# Now block_data is a dictionary that maps category names to dictionaries
# that in turn map column names to lists with values.
doc = cif.Document()
block = doc.add_new_block(block_name[5:])
for cat, data in block_data.items():
    block.set_mmcif_category('_'+cat, data)
doc.write_file(file_out)
```

Chemical Component Dictionary

For something a bit different, let us look at the data from the `components.cif` from CCD. This file describes all the monomers (residues, ligands, solvent molecules) from the PDB entries.

As an exercise, let us check heavy atoms in selenomethionine:

```
>>> from gemmi import cif
>>> cif.read('components.cif')
<gemmi.cif.Document with ... blocks (000, 001, 002...)>
>>> _.find_block('MSE')
<gemmi.cif.Block MSE>
>>> _.find('_chem_comp_atom.', ['atom_id', 'type_symbol'])
<gemmi.cif.Table 20 x 2>
>>> [':'.join(a) for a in _ if a[1] != 'H']
['N:N', 'CA:C', 'C:C', 'O:O', 'OXT:O', 'CB:C', 'CG:C', 'SE:SE', 'CE:C']
```

One may wonder what is the heaviest CCD component:

```
>>> ccd = cif.read('components.cif')
>>> max(ccd, key=lambda b: float(b.find_value('_chem_comp.formula_weight')))
<gemmi.cif.Block WO2>
>>> _.find_value('_chem_comp.formula')
'"O62 P2 W18"'
```

Or which one has the largest number of heavy atoms:

```
>>> el_tag = '_chem_comp_atom.type_symbol'
>>> max(ccd, key=lambda b: sum(el != 'H' for el in b.find_values(el_tag)))
<gemmi.cif.Block X12>
>>> _.find_value('_chem_comp.formula')
'"C96 H153 N31 O25"'
```

The `components.cif` file is big, so we may want to split it into multiple file. As an example, here we create a new document with only one block, and we write it to a file:

```
>>> block = ccd['X12']
>>> d = cif.Document()
>>> d.add_copied_block(ccd['X12'])
<gemmi.cif.Block X12>
>>> d.write_file('X12.cif')
```

In the next example we delete things we do not need. Let us write only components on letter A to a new file. Additionally, we remove the descriptor category:

```
>>> ccd = cif.read('components.cif')
>>> len(ccd)
25219
>>> to_be_deleted = [n for n, block in enumerate(ccd) if block.name[0] != 'A']
>>> for index in reversed(to_be_deleted):
...   del ccd[index]
...
>>> len(ccd)
991
>>> for block in ccd:
...     block.find_mmcif_category('_pdbx_chem_comp_descriptor.').erase()
...
>>> ccd.write_file('A.cif')
```

The examples here present low-level, generic handling of CCD as a CIF file. If we would need to look into coordinates, it would be better to use higher level representation of the chemical component, which is provided by *gemmi::ChemComp*.

1.3 Symmetry

The Gemmi symmetry module provides space group related functionality needed in other parts of the library – when working with coordinate files, electron density maps and reflections.

Although the Gemmi project is developed for macromolecular crystallography, for which only 65 space groups are relevant, we cover all the 230 crystallographic space groups for the sake of completeness.

For C++: this part of Gemmi has no dependencies, all is in a single header `symmetry.hpp`.

1.3.1 Space group table

Gemmi tabulates 550+ settings of the 230 crystallographic space groups. Each entry includes:

- `number` – space group number (1-230),
- `ccp4` – ccp4 number (assigned to particular settings; modulo 1000 they give space group number: 3 and 1003 correspond to `P 1 2 1` and `P 1 1 2`; 0 means none),
- `hm` – Herman-Mauguin (H-M) symbol a.k.a. the international notation (`I a -3 d`, `C 1 2 1`),
- `ext` – extension to the H-M notations (none, `1`, `2`, `H` or `R`) that make extended H-M symbol (`R 3:H`, `P 4/n:1`),
- `hall` – the Hall symbol (`-I 4bd 2c 3`, `C 2y`, `P 32 2c (0 0 -1)`) used to generate the space group operations,
- and `basisop` – change of basis operator from the reference setting.

This data is derived primarily from the CCP4 `syminfo.lib` file, which in turn is based on the data from sgtbx that was augmented with the old data from a CCP4 file named `symop.lib`.

The data from sgtbx is also available in the older SgInfo library, as well as in the International Tables for Crystallography Vol. B ch. 1.4 (in 2010 ed.). It has 530 entries including 3 duplicates (different names for the same settings) in the space group 68.

Gemmi includes also settings from OpenBabel that are absent in `syminfo.lib`. If needed we will add more entries in the future. For example, we do not include all the C- and F-centred tetragonal space groups featured in Crystallographic Space Group Diagrams and Tables by Jeremy Karl Cockcroft (they are also listed on this page written by R.W. Grosse-Kunstleve, and mentioned in the 2015 edition of ITfC Vol.A, Table 1.5.4.4).

We also tabulated alternative names. For now this only includes new standard names introduced in 1990's by the IUCr committee. For example, the space group no. 39 is now officially, in the Volume A of the International Tables, named Aem2 not Abm2. Most of the crystallographic software (as well as the Volume B of the Tables) still use the old names. (spglib uses new ones, sgtbx reads new names with the option `ad_hoc_1992`).

The usual way to access a space group from the table is to search it by name. In C++:

```
#include <gemmi/symmetry.hpp>
// ...
const SpaceGroup* sg = find_spacegroup_by_name(name);
```

and in Python:

```
>>> import gemmi
>>> gemmi.find_spacegroup_by_name('I2')
<gemmi.SpaceGroup("I 1 2 1")>
```

Note: The rest of this section has only Python examples mixed with the text. One longer C++ example is at the end.

The name above is expected to be either a full international Herman-Mauguin symbol or a short symbol (*I2* instead of *I 1 2 1*). This functions also searches the tabulated alternative names:

```
>>> gemmi.find_spacegroup_by_name('C m m e') # new names have 'e' and 'g'
<gemmi.SpaceGroup("C m m a")>
```

Sometimes in the PDB, the setting of the hexagonal crystal system is not clear from the symmetry symbol alone. For instance, the H-M symbol "R 3" can mean either hexagonal or rhombohedral setting. This ambiguity can be resolved by comparing angles of the unit cell. The ratio of gamma to alpha angles is 120:90 in the hexagonal system and 1:1 in

rhombohedral. Therefore, `find_spacegroup_by_name()` accepts also alpha and gamma angles. If the angles are not passed, the hexagonal system is returned:

```
>>> gemmi.find_spacegroup_by_name('R 3 2')
<gemmi.SpaceGroup("R 3 2:H")>
>>> gemmi.find_spacegroup_by_name('R 3 2', alpha=92.02, gamma=92.02)
<gemmi.SpaceGroup("R 3 2:R")>
>>> gemmi.find_spacegroup_by_name('R 3 2', alpha=90, gamma=120)
<gemmi.SpaceGroup("R 3 2:H")>
>>> # of course, you do not need angles if you use extended H-M symbol
>>> gemmi.find_spacegroup_by_name('R 3 2:R')
<gemmi.SpaceGroup("R 3 2:R")>
```

You can also get space group by number:

```
>>> gemmi.find_spacegroup_by_number(5)
<gemmi.SpaceGroup("C 1 2 1")>
```

The number is the ccp4 number mentioned above, so some 4-digit numbers are also recognized:

```
>>> gemmi.find_spacegroup_by_number(4005)
<gemmi.SpaceGroup("I 1 2 1")>
```

For values 1-230 the number corresponds to the first setting of this space group in the table in ITfC vol. B. Usually, it is the reference (standard) setting of the space group, but unfortunately not always. So we have another function that returns always the reference setting:

```
>>> gemmi.get_spacegroup_reference_setting(48)
<gemmi.SpaceGroup("P n n n:2")>
```

The next function for searching the space group table checks symmetry operations:

```
>>> gemmi.symops_from_hall('C 2y (x,y,-x+z)')
<gemmi.GroupOps object at 0x...>
>>> gemmi.find_spacegroup_by_ops(_)
<gemmi.SpaceGroup("I 1 2 1")>
>>> _.hall
'I 2y'
```

This example shows also how to find space group corresponding to a Hall symbol. The Hall notation encodes all the group operations. Unfortunately, in a non-unique way. Different Hall symbols can be used to encode the same symmetry operations. In the example above "C 2y (x,y,-x+z)" is equivalent to "I 2y". That's why we compare operations not symbols.

The last function for searching space group is also comparing operations. It takes two arguments: a space group and a change-of-basis operator, and searches for space group settings that match the transformed operations of the original space group:

```
>>> # I2 -> C2
>>> gemmi.find_spacegroup_by_change_of_basis(gemmi.SpaceGroup('I2'), gemmi.Op('x,y,x+z
↪'))
<gemmi.SpaceGroup("C 1 2 1")>
>>> # enantiomorphic pair
>>> gemmi.find_spacegroup_by_change_of_basis(gemmi.SpaceGroup('P 41'), gemmi.Op('-x,-
↪y,-z'))
<gemmi.SpaceGroup("P 43")>
```

Finally, we may iterate over the space group table:

```
>>> for sg in gemmi.spacegroup_table():
...     if sg.ext == 'H':
...         print(sg)
<gemmi.SpaceGroup("R 3:H")>
<gemmi.SpaceGroup("R -3:H")>
<gemmi.SpaceGroup("R 3 2:H")>
<gemmi.SpaceGroup("R 3 m:H")>
<gemmi.SpaceGroup("R 3 c:H")>
<gemmi.SpaceGroup("R -3 m:H")>
<gemmi.SpaceGroup("R -3 c:H")>
```

`gemmi.SpaceGroup` represents an entry in the space group table. It has the properties listed at the beginning of this section (`number`, `ccp4`, `hm`, `ext`, `hall`) and a few methods:

```
>>> sg = gemmi.SpaceGroup('R3')  # equivalent to find_spacegroup_by_name()
>>> sg.xhm()                     # extended Hermann-Mauguin name
'R 3:H'
>>> sg.short_name()              # short name
'H3'
>>> sg.is_enantiomorphic()       # is it one of 22 chiral space groups?
False
>>> sg.is_sohncke()              # is it one of 65 Sohncke space groups?
True
>>> sg.point_group_hm()          # H-M name of the point group
'3'
>>> sg.laue_str()                # name of the Laue class
'-3'
>>> sg.crystal_system_str()      # name of the crystal system
'trigonal'
>>> sg.is_reference_setting()
True
>>> # and the most important...
>>> sg.operations()
<gemmi.GroupOps object at 0x...>
```

Categories related to chirality can be confusing. Here, we follow the IUCr dictionary:

- Sohncke groups (a.k.a. non-enantiogenic space groups) are "the three-dimensional space groups containing only operations of the first kind (rotations, rototranslations, translations)"; 65 groups in which chiral structures crystallize.
- Enantiomorphic space groups (a.k.a. chiral groups) are those whose *group* structure is chiral; 22 groups forming 11 enantiomorphic pairs. (So chiral structures can crystallize not only in the chiral space groups but also in 43 of the achiral ones.)

If you would like to ignore entries that are absent in SgInfo, sgtbx, spglib and in the International Tables vol. B, use only the first 530 entries of the gemmi table. In Python, we have a helper function for this:

```
>>> for sg in gemmi.spacegroup_table_itb():
...     pass
>>> sg.ccp4  # sg here is the last space group that was iterated
230
```

Implementation notes

The choice between having an explicit list of all the operations and generating them from a smaller set of symbols is a trade-off between simplicity of the code and the amount of the tabulated data.

The number of symmetry operations per space group is between 1 and 192, but they can be split into symmetry operations (max. 48 for point group m-3m) and so-called *centring vectors* (max. 4 for the face-centered lattice).

- The simplest way of storing the operations is to list them all (i.e. 192 triplets for no. 228) in a text file. This approach is used by OpenBabel (in `space-groups.txt`).
- It makes sense to keep the centring vectors separately (192 becomes 48 + 4). This is done in the CCP4 `syminfo.lib` file (539 entries), which is used by csymlib (part of libccp4) and a few other projects.
- The operations can be as well tabulated in the code. This approach is used (with one or two layers of indirection to reduce the data size) by spglib (C library) and NGL.
- The inversion center (if applicable) can be kept separately, to reduce the maximum number of stored operations (48 -> 24+1). This is how the symmetry data is encoded in Fityk.
- Actually all the operations can be generated from only a few generators, at the expense of more complex code. Example: Mantid (`SpaceGroupFactory.cpp`).
- Finally, one can use one of the two computer-adapted descriptions from ITfC. The so-called explicit notation (`ICC$I3Q000$P4C393$P2D933`) is the longer of the two, but easier to parse by the computer. It is used in the SPGGEN program.
- The Hall notation (`-I 4bd 2c 3`), first proposed by Sydney R. Hall in 1981, is shorter and more popular. It can be interpreted by a few libraries:
  - SgInfo and SgLite (old C libraries from Ralf W. Grosse-Kunstleve recently re-licensed to BSD),
  - sgtbx (successor of SgInfo written in C++/Python, part of cctbx),
  - CCP4 Clipper,

  and by many programs. On the bad side, the conciseness is achieved by complex rules of interpreting the symbols; the choice of a Hall symbol for given settings is not unambiguous and the symbols differ between editions of ITfC, and between sgtbx and `syminfo.lib`.

After contemplating all the possibilities we ended up implementing the most complex solution: Hall symbols. The relative complexity does not mean it is slow: translating a Hall notation to generators takes less than a microsecond on a typical desktop machine. Closing a group is also below a microsecond for most of the groups, and up to a few microseconds for the highest symmetry group Fm3m (Gemmi uses Dimino's algorithm for this).

1.3.2 Operations

Crystallographic symmetry operations have a few notations. Gemmi undertands only coordinate triplets (sometimes called the Jones' faithful notation) such as `x,x-y,z+1/2`. The symmetry operation is represented by class Op. It can be created in C++ as:

```
#include <gemmi/symmetry.hpp>
...
gemmi::Op op = gemmi::parse_triplet("-y,x-y,z");
```

and in Python as:

```
>>> import gemmi
>>> op = gemmi.Op('-y,x-y,z+1/3')
```

We can also do the opposite:

```
>>> op.triplet()
'-y,x-y,z+1/3'
```

The operation consists of a 3x3 rotation matrix and a translation vector, both stored internally as integers that need to be divided by `DEN` == 24 to get the actual values.

```
>>> op.rot
[[0, -24, 0], [24, -24, 0], [0, 0, 24]]
>>> op.tran
[0, 0, 8]
```

Alternatively, the operation can be expressed as a single 4x4 transformation matrix (which in crystallography is called *Seitz matrix*).

```
>>> op.seitz()
[[0, -1, 0, 0], [1, -1, 0, 0], [0, 0, 1, Fraction(1, 3)], [0, 0, 0, 1]]
>>> op.float_seitz()
[[0.0, -1.0, 0.0, 0.0],
 [1.0, -1.0, 0.0, 0.0],
 [0.0, 0.0, 1.0, 0.3333333333333333],
 [0.0, 0.0, 0.0, 1.0]]
```

Operations can be combined, inverted and wrapped:

```
>>> gemmi.Op('x-y,x,z+1/6') * '-x,-y,z+1/2'
<gemmi.Op("-x+y,-x,z+2/3")>
>>> _.inverse()
<gemmi.Op("-y,x-y,z-2/3")>
>>> _.wrap()
<gemmi.Op("-y,x-y,z+1/3")>
```

Wrapping applies *modulo* 1 to the translational part. Which is usually desirable for crystallographic symmetry. But the triplets and matrices may represent also, for example, generation of a biological assembly. For this reason, `x,y+1,z` is not automatically reduced to the identity. But when operations are combined, they are assumed to be symmetry operations and the result is wrapped to [0,1):

```
>>> op
<gemmi.Op("-y,x-y,z+1/3")>
>>> op * op
<gemmi.Op("-x+y,-x,z+2/3")>
>>> op * op * op  # without wrapping we'd have z+1
<gemmi.Op("x,y,z")>
```

The `Op.rot` matrix is called "rotation matrix", because that's the primary purpose, but it can also represent different linear transformations:

```
>>> enlarging_op = gemmi.Op("-y+z,x+z,-x+y+z")
>>> enlarging_op.inverse()
<gemmi.Op("-1/3*x+2/3*y-1/3*z,-2/3*x+1/3*y+1/3*z,1/3*x+1/3*y+1/3*z")>
>>> _ * enlarging_op
<gemmi.Op("x,y,z")>
```

In the real space, a crystal symmetry operation can be applied to the fractional atom position to get an equivalent position:

```
>>> op.apply_to_xyz([0.25, 0.21875, 0.3])
[-0.21875, 0.03125, 0.6333333333333333]
```

In the reciprocal space, the same operation relates equivalent reflections. The rotational part determines Miller indices, the translational part – phase shift.

```
>>> hkl = [3, 0, 1]
>>> op.apply_to_hkl(hkl)
[0, -3, 1]
>>> op.phase_shift(hkl)  # -120 degrees in radians
-2.0943951023931953
```

1.3.3 Groups of Operations

Each space group setting corresponds to a unique set of operations. This set is represented by class `GroupOps`.

Symmetry operations (rotation + translation) and centring vectors (translation only) are stored separately:

```
>>> ops = gemmi.find_spacegroup_by_name('I2').operations()
>>> ops
<gemmi.GroupOps object at 0x...>
>>> list(ops.sym_ops)
[<gemmi.Op("x,y,z")>, <gemmi.Op("-x,y,-z")>]
>>> list(ops.cen_ops)
[[0, 0, 0], [12, 12, 12]]
```

but they are be combined on the fly:

```
>>> len(ops)
4
>>> for op in ops:
...   print(op.triplet())
...
x,y,z
-x,y,-z
x+1/2,y+1/2,z+1/2
-x+1/2,y+1/2,-z+1/2
```

We can apply a change-of-basis operator to GroupOps:

```
>>> ops.change_basis(gemmi.Op('x,y,x+z'))  # I2 -> C2
>>> gemmi.find_spacegroup_by_ops(ops)
<gemmi.SpaceGroup("C 1 2 1")>
```

We can create GroupOps from a list of operations:

```
>>> op_list = ['x,y,z', 'x,-y,z+1/2', 'x+1/2,y+1/2,z', 'x+1/2,-y+1/2,z+1/2']
>>> new_ops = gemmi.GroupOps([gemmi.Op(o) for o in op_list])
```

or from a Hall symbol:

```
>>> gemmi.symops_from_hall('P 4w 2c')
<gemmi.GroupOps object at 0x...>
>>> len(_)
8
```

The Hall symbols encode *generators* which are then used to obtain all the operations. If you'd wonder what generators are encoded, use:

```
>>> list(gemmi.generators_from_hall('P 4w 2c'))  # no.93
[<gemmi.Op("x,y,z")>, <gemmi.Op("-y,x,z+1/4")>, <gemmi.Op("x,-y,-z+1/2")>]
```

Combining these 3 generators reconstructs all the 8 symmetry operations.

The GroupOps object has a couple of functions:

```
>>> new_ops.is_centric()
False
>>> new_ops.find_centering()
'C'
```

and, again, it can be used to search in the space group table:

```
>>> gemmi.find_spacegroup_by_ops(new_ops)
<gemmi.SpaceGroup("C 1 c 1")>
```

1.3.4 C++ Example

Since the code snippets above were only in Python, here we compensate it with one longer example in C++.

```
#include <assert.h>
#include <iostream>

#include <gemmi/symmetry.hpp>

int main() {
  // operators
  gemmi::Op op = gemmi::parse_triplet("-y,x-y,z"); // one operation from P 3
  assert(op * op == gemmi::parse_triplet("-x+y,-x,z")); // and the other one
  assert(op * op == op.inverse());

  // iteration over all tabulated settings
  for (const gemmi::SpaceGroup& sg : gemmi::spacegroup_tables::main)
    std::cout << sg.number << ' ' << sg.xhm() << "   " << sg.hall << '\n';

  // selecting a space group
  const gemmi::SpaceGroup* c2 = gemmi::find_spacegroup_by_number(5);
  assert(c2->xhm() == "C 1 2 1");
  assert(c2->ccp4 == 5);

  const gemmi::SpaceGroup* i2 = gemmi::find_spacegroup_by_name("I2");
  assert(i2->number == 5);
  assert(i2->ccp4 == 4005);
  assert(i2->xhm() == "I 1 2 1");

  gemmi::GroupOps ops = i2->operations();
  for (const gemmi::Op& operation : ops)
    std::cout << "   " << operation.triplet();
  // output:   x,y,z   -x,y,-z   x+1/2,y+1/2,z+1/2   -x+1/2,y+1/2,-z+1/2
  ops.change_basis(gemmi::parse_triplet("x,y,x+z"));
  assert(gemmi::find_spacegroup_by_ops(ops) == c2);

  assert(gemmi::find_spacegroup_by_name("C m m e") ==  // new name with 'e'
         gemmi::find_spacegroup_by_name("C m m a"));
}
```

1.4 Molecular models

In this section we show how to handle structural models of biomolecules (to some degree, it also applies to small molecules and inorganic structures).

Models from a single file (PDB, mmCIF, etc.) are stored in the `Structure` class, with the usual Model-Chain-Residue-Atom hierarchy. Gemmi provides basic functions to access and manipulate the structure, and on top of it more complex functions, such as neighbor search, calculation of dihedral angles, removal of ligands from a model, etc.

Comparing with tools rooted in bioinformatics:

- Gemmi focuses more on working with incomplete models (on all stages before they are published and submitted to the PDB),
- and Gemmi is aware of the neighbouring molecules that are implied by the crystallographic and non-crystallographic symmetry.

1.4.1 Elements

When working with molecular structures it is good to have basic data from the periodic table at hand.

C++

```
#include <gemmi/elem.hpp>

gemmi::Element el("Mg");
int its_number = el.atomic_number();
double its_weight = el.weight();
const char* its_name = el.name();
```

Python

```
>>> import gemmi
>>> gemmi.Element('Mg').weight
24.305
>>> gemmi.Element(118).name
'Og'
>>> gemmi.Element('Mo').atomic_number
42
```

We also included covalent radii of elements from a Wikipedia page, which has data from Cordero *et al* (2008), *Covalent radii revisited*, Dalton Trans. 21, 2832.

```
>>> gemmi.Element('Zr').covalent_r
1.75
```

and a flag for metals (the classification is somewhat arbitrary):

```
>>> gemmi.Element('Mg').is_metal
True
>>> gemmi.Element('C').is_metal
False
```

1.4.2 Small Molecules

CIF files that describe small-molecule and inorganic structures can be read into an SmallStructure object. Unlike macromolecular Structure, SmallStructure has no hierarchy. It is just a flat list of atomic sites (`SmallStructure::Site`) together with the unit cell and symmetry.

```
#include <cassert>
#include <gemmi/cif.hpp>
#include <gemmi/smcif.hpp>

int main() {
  auto block = gemmi::cif::read_file("1011031.cif").sole_block();
  gemmi::SmallStructure SiC = gemmi::make_small_structure_from_block(block);
  assert(SiC.cell.a == 4.358);
  assert(SiC.spacegroup_hm == "F -4 3 m");
  assert(SiC.sites.size() == 2);
  assert(SiC.get_all_unit_cell_sites().size() == 8);
}
```

```
>>> import gemmi
>>> SiC = gemmi.read_small_structure('../tests/1011031.cif')
>>> SiC.cell.a
4.358
>>> SiC.spacegroup_hm
'F -4 3 m'
>>> SiC.sites
[<gemmi.SmallStructure.Site Si1>, <gemmi.SmallStructure.Site C1>]
>>> len(SiC.get_all_unit_cell_sites())
8
>>> site = SiC.sites[0]
>>> site.label
'Si1'
>>> site.type_symbol
'Si4+'
>>> site.fract
<gemmi.Fractional(0, 0, 0)>
>>> site.occ
1.0
>>> site.u_iso  # not specified here
0.0
>>> site.element  # obtained from type_symbol 'Si4+'
<gemmi.Element: Si>
>>> site.charge   # obtained from type_symbol 'Si4+'
4
```

We will need another cif file to show anisotropic ADPs and disorder_group:

```
>>> perovskite = gemmi.read_small_structure('../tests/4003024.cif')
>>> for site in perovskite.sites:
...   print(site.label, site.aniso.nonzero(), site.disorder_group or 'n/a')
Cs1 True n/a
Sn2 False 1
Cl1 True n/a
In False 2
>>> perovskite.sites[2].aniso.u11
0.10300000000000001
>>> perovskite.sites[2].aniso.u22
```

(continues on next page)

(continued from previous page)

```
0.15600000000000003
>>> perovskite.sites[2].aniso.u33
0.15600000000000003
>>> perovskite.sites[2].aniso.u12
0.0
>>> perovskite.sites[2].aniso.u13
0.0
>>> perovskite.sites[2].aniso.u23
0.0
```

1.4.3 Chemical Components

Residues (monomers) and small molecule components of macromolecular models are called *chemical components*. Gemmi can use three sources of knowledge about chemical components:

- built-in basic data about 350+ popular components,
- the Chemical Component Dictionary (CCD) maintained by the PDB (25,000+ components),
- so-called CIF files compatible with the format of the Refmac/CCP4 monomer library.

Built-in data

The built-in data is accessed through the function `find_tabulated_residue`. It contains only minimal information about each residue: assigned category, the "standard" flag (non-standard residues are marked as HETATM in the PDB, even in polymer), one-letter code, the number of hydrogens and molecular weight:

```
#include <gemmi/resinfo.hpp>

gemmi::ResidueInfo info = gemmi::find_tabulated_residue("ALA");
bool is_it_aminoacid = info.is_amino_acid();
int approximate_number_of_h_atoms = info.hydrogen_count;
```

```
>>> gln = gemmi.find_tabulated_residue('GLN')
>>> gln.is_amino_acid()
True
>>> gln.one_letter_code
'Q'
>>> round(gln.weight, 3)
146.144
>>> gln.hydrogen_count
10
>>> gemmi.find_tabulated_residue('DOD').is_water()
True
>>> # PDB marks "non-standard" residues as HETATM.
>>> # Pyrrolysine is standard - some microbes have it.
>>> gemmi.find_tabulated_residue('PYL').is_standard()
True
>>> gemmi.find_tabulated_residue('MSE').is_standard()
False
```

CCD and monomer libraries

To get more complete information, including atoms and bonds in the monomer, we need to first read either the CCD or a monomer library.

The CCD `components.cif` file describes all the monomers (residues, ligands, solvent molecules) from the PDB entries. Importantly, it contains information about bonds.

Note: The absence of bond information in mmCIF files from wwPDB is a well-known problem, mitigated somewhat by PDBe which in parallel to the wwPDB archive has also mmCIF files with connectivity and bond-order information; and by RCSB which has this information in the MMTF format.

Macromolecular refinement programs need to know more about monomers than the CCD can tell: they need to know how to restrain the structure. Therefore, they have own dictionaries of monomers (a.k.a monomer libraries), such as the Refmac dictionary, where each monomer is described by one cif file. These libraries are often complemented by user's own cif files.

Gemmi has class `ChemComp` that corresponds to the data about a monomer from either the CCD or a cif file.

```
#include <gemmi/cif.hpp>             // for cif::read_file
#include <gemmi/chemcomp.hpp>        // for ChemComp, make_chemcomp_from_block

gemmi::ChemComp make_chemcomp(const char* path) {
  gemmi::cif::Document doc = gemmi::cif::read_file(path);
  // assuming the component description is in the last block of the file
  return gemmi::make_chemcomp_from_block(doc.blocks.back());
}
```

```
>>> # SO3.cif -> gemmi.ChemComp
>>> block = gemmi.cif.read('../tests/SO3.cif')[-1]
>>> so3 = gemmi.make_chemcomp_from_block(block)
```

It also has class `MonLib` that corresponds to a monomer library. In addition to storing a mapping between residue names and `ChemComp`s, it also stores information that in the CCP4 monomer library is kept in `mon_lib_list.cif`: description of chemical links and modifications, and classification of the residues.

These classes are not documented yet. The examples in *Graph analysis* show how to access the lists of atoms and bonds from `ChemComp`.

1.4.4 Coordinates and matrices

Coordinates are represented by two classes:

- `Position` for coordinates in Angstroms (orthogonal coordinates),
- `Fractional` for coordinates relative to the unit cell (fractional coordinates).

Both `Position` and `Fractional` are derived from `Vec3`, which has three numeric properties: `x`, `y` and `z`.

```
>>> v = gemmi.Vec3(1.2, 3.4, 5.6)
>>> v.y = -v.y
>>> # it can also be indexed
>>> v[1]
-3.4
```

The only reason to have separate types is to prevent functions that expect fractional coordinates from accepting orthogonal ones, and vice versa. In C++ these types are defined in `gemmi/math.hpp`.

If you have points in space you may want to calculate distances, angles and dihedral angles:

```
>>> from math import degrees
>>> p1 = gemmi.Position(0, 0, 0)
>>> p2 = gemmi.Position(0, 0, 1)
>>> p3 = gemmi.Position(0, 1, 0)
>>> p4 = gemmi.Position(-1, 1, 0)
>>> p1.dist(p2)
1.0
>>> degrees(gemmi.calculate_angle(p1, p2, p3))
45.00000000000001
>>> degrees(gemmi.calculate_dihedral(p1, p2, p3, p4))
90.0
```

Additionally, in C++ you have other functions. See headers `gemmi/math.hpp` and `gemmi/calculate.hpp`.

Working with macromolecular coordinates involves 3D transformations, such as crystallographic and non-crystallographic symmetry operations, and fractionalization and orthogonalization of coordinates. This requires a tiny bit of linear algebra.

3D transformations tend to be represented either by a 4x4 matrix, or by a 3x3 matrix and a translation vector. Gemmi uses the latter. Transformations are represented by the `Transform` class that has two member variables: `mat` (of type `Mat33`) and `vec` (of type `Vec3`, which was introduced above).

```
>>> tr = gemmi.Transform()  # identity
>>> tr.mat
<gemmi.Mat33 [1, 0, 0]
             [0, 1, 0]
             [0, 0, 1]>
>>> tr.vec
<gemmi.Vec3(0, 0, 0)>
```

Both `Vec3` and `Mat33` can be converted to and from Python's list: In case of `Mat33` it is a nested list:

```
>>> tr.vec.fromlist([3.0, 4.5, 5])
>>> tr.vec.tolist()
[3.0, 4.5, 5.0]

>>> # nested listed for Mat33
>>> m = tr.mat.tolist()
>>> m
[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
>>> m[1][2] = -5
>>> tr.mat.fromlist(m)
>>> tr.mat
<gemmi.Mat33 [1, 0, 0]
             [0, 1, -5]
             [0, 0, 1]>
```

Here is an example that shows a few other properties:

```
>>> # get NCS transformation from an example pdb file
>>> ncs_op = gemmi.read_structure('../tests/1lzh.pdb.gz').ncs[0].tr
>>> type(ncs_op)
```

(continues on next page)

(continued from previous page)

```
<class 'gemmi.Transform'>
>>> ncs_op.mat
<gemmi.Mat33 [0.97571, -0.2076, 0.06998]
             [0.2156, 0.96659, -0.13867]
             [-0.03885, 0.15039, 0.98786]>
>>> _.determinant()
1.000003887799667
>>> ncs_op.vec
<gemmi.Vec3(-14.1959, 0.72997, -30.5229)>

>>> # is the 3x3 matrix above orthogonal?
>>> mat = ncs_op.mat
>>> identity = gemmi.Mat33()
>>> mat.multiply(mat.transpose()).approx(identity, epsilon=1e-5)
True

>>> ncs_op.apply(gemmi.Vec3(20, 30, 40))
<gemmi.Vec3(1.8895, 28.4929, 12.7262)>
>>> ncs_op.inverse().apply(_)
<gemmi.Vec3(20, 30, 40)>
```

To avoid mixing of orthogonal and fractional coordinates Gemmi also has `FTransform`, which is like `Transform`, but can be applied only to `Fractional` coordinates.

We have special classes for symmetric 3x3 matrices: SMat33f and SMat33d (for 32- and 64-bit floating point numbers, respectively). These classes are used primarily for anisotropic ADP tensors; their member variables are named `u11`, `u22`, `u33`, `u12`, `u13` and `u23`. SMat33 classes provide a few methods, including calculations of eigenvalues and eigenvectors.

```
>>> aniso = perovskite.sites[2].aniso
>>> aniso.u11
0.10300000000000001
>>> aniso.trace()
0.41500000000000004
>>> aniso.determinant()
0.002506608000000001
>>> aniso.calculate_eigenvalues()
[0.10300000000000001, 0.15600000000000003, 0.15600000000000003]
```

In C++ all these types are defined in `gemmi/math.hpp`.

1.4.5 Unit Cell

When working with a structural model in a crystal we need to know the unit cell. In particular, we need to be able to switch between orthogonal and fractional coordinates. Here are the most important properties and methods of the `UnitCell` class:

C++

```
#include <gemmi/unitcell.hpp>

// UnitCell has:
// * directly set properties a, b, c, alpha, beta, gamma,
// * calculated properties such as ar (a*), br (b*), ..., volume,
```

(continues on next page)

(continued from previous page)

```
// * fractionalization and orthogonalization matrices,
// * a list of images (symmetry or NCS mates) that is set externally
//   when reading a file.
// * and a few functions such as orthogonalize(), fractionalize(),
//   is_special_position(), find_nearest_image().

gemmi::UnitCell cell(25.14, 39.50, 45.07, 90, 90, 90);
gemmi::Position p = cell.orthogonalize(gemmi::Fractional(0.5, 0.5, 0.5));
```

Python

```
>>> cell = gemmi.UnitCell(25.14, 39.50, 45.07, 90, 90, 90)
>>> cell
<gemmi.UnitCell(25.14, 39.5, 45.07, 90, 90, 90)>
>>> cell.a, cell.b, cell.c
(25.14, 39.5, 45.07)
>>> cell.alpha, cell.beta, cell.gamma
(90.0, 90.0, 90.0)
>>> cell.volume
44755.8621
>>> cell.fractionalization_matrix
<gemmi.Mat33 [0.0397772, -0, -0]
             [0, 0.0253165, 0]
             [0, 0, 0.0221877]>
>>> cell.fractionalize(gemmi.Position(10, 10, 10))
<gemmi.Fractional(0.397772, 0.253165, 0.221877)>
>>> cell.orthogonalization_matrix
<gemmi.Mat33 [25.14, 0, 0]
             [0, 39.5, -0]
             [0, 0, 45.07]>
>>> cell.orthogonalize(gemmi.Fractional(0.5, 0.5, 0.5))
<gemmi.Position(12.57, 19.75, 22.535)>
```

The UnitCell object can also store a list of symmetry transformations. This list is populated automatically when reading a coordinate file. It contains crystallographic symmetry operations. In rare cases when the file defines strict NCS operarations that are not "given" (MTRIX record in the PDB format or _struct_ncs_oper in mmCIF) the list contains also the NCS operations. With this list we can use:

- `UnitCell::volume_per_image() -> double` – returns `UnitCell::volume` divided by the number of the molecule images in the unit cell,

```
>>> st = gemmi.read_structure('../tests/1pfe.cif.gz')
>>> st.spacegroup_hm
'P 63 2 2'
>>> st.cell.volume / st.cell.volume_per_image()
12.0
```

- `UnitCell::is_special_position(const Position& pos, double max_dist=0.8) -> int` – returns the number of nearby symmetry mates of an atom. Non-zero only for atoms on special positions. For example, returns 3 for an atom on 4-fold symmetry axis.

```
>>> # chloride ion in 1PFE is significantly off the special position
>>> cl = st[0].sole_residue('A', gemmi.SeqId('20'))[0]
>>> cl
<gemmi.Atom CL at (-0.3, 23.0, -19.6)>
>>> round(1.0 / cl.occ)
```

(continues on next page)

(continued from previous page)

```
6
>>> st.cell.is_special_position(cl.pos, max_dist=0.5)
0
>>> st.cell.is_special_position(cl.pos, max_dist=0.8)
3
>>> st.cell.is_special_position(cl.pos, max_dist=1.2)
5
```

- `UnitCell::find_nearest_image(const Position& ref, const Position& pos, Asu asu) -> SymImage` – with the last argument set to `Asu::Any`, it returns the symmetric image of `pos` that is nearest to `ref`. The last argument can also be set to `Asu::Same` or `Asu::Different`.

In the reciprocal space, the unit cell can be used to determine interplanar spacing d_{hkl} (the resolution corresponding to reflection):

```
>>> cell.calculate_d([0, 1, 0])
39.5
```

Computationally, d is calculated from $1/d^2$, so if you need the latter you can calculate it directly:

```
>>> cell.calculate_1_d2([8, -9, 10])
0.20240687828293985
```

1.4.6 Reading coordinate files

Gemmi support the following coordinate file formats:

- mmCIF (PDBx/mmCIF),
- PDB (with popular extensions),
- mmJSON,
- a binary format (MMTF, binary CIF, or own format) is to be considered.

In this section we show how to read a coordinate file in Gemmi. In the next sections we will go into details of the individual formats. Finally, we will show what can be done with a structural model.

C++

All the macromolecular coordinate files supported by Gemmi can be opened using:

```
Structure read_structure_file(const std::string& path, CoorFormat␣
↪format=CoorFormat::Unknown)

// where CoorFormat is defined as
enum class CoorFormat { Unknown, Pdb, Mmcif, Mmjson };
```

For example:

```
#include <gemmi/mmread.hpp>
// ...
gemmi::Structure st = gemmi::read_structure_file(path);
std::cout << "This file has " << st.models.size() << " models.\n";
```

In this example the file format is not specified and is determined from the file extension.

`gemmi::Structure` is defined in `gemmi/model.hpp` and it will be documented *later on*.

Gemmi also has a templated function `read_structure` that you can use to customize how you provide the data (bytes) to the parsers. This function is used to uncompress gzipped files on the fly:

```
#include <iostream>
#include <gemmi/mmread.hpp>
#include <gemmi/gz.hpp>

int main(int argc, char** argv) {
  for (int i = 1; i < argc; ++i)
    try {
      auto st = gemmi::read_structure(gemmi::MaybeGzipped(argv[i]));
      std::cout << "This file has " << st.models.size() << " models.\n";
    } catch (std::runtime_error& e) {
      std::cout << "Oops. " << e.what() << std::endl;
    }
}
```

If you include the `gz.hpp` header (as in the example above) the resulting program must be linked with the zlib library.

```
$ c++ -std=c++11 -Iinclude example_above.cpp -lz
$ ./a.out 2cco.cif.gz
This file has 20 models.
```

The `gemmi/mmread.hpp` header includes many other headers and is relatively slow to compile. For this reason, consider including it in only one compilation unit (that does not change often).

Alternatively, if you want to support gzipped files, use function `gemmi::read_structure_gz()` declared in the header `gemmi/gzread.hpp` and implemented in `gemmi/gzread_impl.hpp`. The latter header must be included in only one compilation unit.

If you know the format of files that you will read, you may also use a function specific to this format. For example, the next section shows how to read just a PDB file (`read_pdb_file(path)`).

Python

Any of the macromolecular coordinate files supported by Gemmi (gzipped or not) can be opened using:

```
>>> gemmi.read_structure(path)
<gemmi.Structure ...>
```

If the file format is not specified (example above) it is determined from the file extension. If the extension is not canonical you can specify the format explicitely:

```
>>> gemmi.read_structure(path, format=gemmi.CoorFormat.Pdb)
<gemmi.Structure ...>
```

The file form `gemmi.Structure` will be documented *later on*.

1.4.7 PDB format

The PDB format evolved between 1970's and 2012. Nowadays the PDB organization uses PDBx/mmCIF as the primary format and the legacy PDB format is frozen.

Note: The PDB format specification aims to describe the format of files generated by the wwPDB. It does not aim to specify a format that can be used for data exchange between third-party programs. Following literally the specification is neither useful nor possible. For example: the REVDAT record is mandatory, but using it makes sense only for the entries released by the PDB. Therefore no software generates files conforming to the specification except from the wwPDB software (and even this one is not strictly conforming: it writes `1555` in the LINK record for the identity operator while the specifications requires leaving these fields blank).

Do not read too much into the specification.

Gemmi aims to support all flavours of PDB files that are in common use in the field of macromolecular crystallography. This includes files from wwPDB as well as files outputted by mainstream software.

In particular, we support the following extensions:

- two-character chain IDs (columns 21 and 22),
- segment ID (columns 73-76) from PDB v2,
- hybrid-36 encoding of sequence IDs for sequences longer than 9999 (although we are yet to find an examples for this),
- hybrid-36 encoding of serial numbers for more than 99,999 atoms.

Gemmi interprets more PDB records than most of programs and libraries, but supporting all the records is not a goal. The records that are interpreted can be converted from/to mmCIF:

- HEADER
- TITLE
- KEYWDS
- EXPDTA
- NUMMDL
- REMARK 2
- REMARK 3 (read-only, i.e. only in PDB -> mmCIF conversion)
- REMARK 200/230/240 (read-only)
- REMARK 290 (partly-read, but not by default)
- REMARK 300 (read-only)
- REMARK 350
- DBREF/DBREF1/DBREF2
- SEQRES
- HELIX
- SHEET
- SSBOND
- LINK
- CISPEP
- CRYST1
- ORIGXn

- SCALEn
- MTRIXn
- MODEL/ENDMDL
- ATOM/HETATM
- ANISOU
- TER
- END

Although the PDB format is widely used, some of its features can be easily overlooked. The rest of this section describes such features. It is for people who are interested in the details of the PDB format. You do not need to read it if you just want to use Gemmi and work with molecular models.

Let us start with the the list of atoms:

```
HETATM    8  CE  MSE A   1       8.081   3.884  27.398  1.00 35.65           C
ATOM      9  N   GLU A   2       2.464   5.718  24.671  1.00 14.40           N
ATOM     10  CA  GLU A   2       1.798   5.810  23.368  1.00 13.26           C
```

Standard residues of protein, DNA or RNA are marked as ATOM. Solvent, ligands, metals, carbohydrates and everything else is marked as HETATM. What about non-standard residues of protein, DNA or RNA? According to the wwPDB they are HETATM, but some programs and crystallographers prefer to mark them as ATOM. It is better to not rely on any of the two conventions. In particular, removing ligands and solvent cannot be done by removing all the HETATM records.

The next field after ATOM/HETATM is the serial number of an atom. The wwPDB spec limits the serial numbers to the range 1–99,999, but the popular extension called hybrid-36 allows to have more atoms in the file by using also letters in this field. If you do not need to interpret the CONECT records the serial number can be simply ignored.

Columns 13-27 describe the atom's place in the hierarchy. In the example above they are:

```
1       2
345678901234567

 CE  MSE A   1
 N   GLU A   2
 CA  GLU A   2
```

Here the CE atom is in chain A, in residue MSE with sequence ID 1.

The atom names (columns 13-16) starts with the element name, and as a rule columns 13-14 contain only the element name. Therefore Cα and calcium ion, both named CA, are aligned differently:

```
1       2
345678901234567
 CA  GLU A   2
CA    CA A 101
```

This rule has an exception: when the atom name has four characters it starts in column 13 even if it has a one-letter element code:

```
HETATM 6495  CAX R58 A 502      17.143 -29.934   7.180  1.00 58.54           C
HETATM 6496 CAX3 R58 A 502      16.438 -31.175   6.663  1.00 57.68           C
```

Columns 23-27 contain a sequence ID. It consists of a number (columns 23-26) and, optionally, also an insertion code (A-Z) in column 27:

```
ATOM  11918  CZ  PHE D 100      -6.852  76.356 -23.289  1.00107.94           C
ATOM  11919  N   ARG D 100A     -9.676  74.726 -19.958  1.00105.71           N
...
ATOM  11970  CE  MET D 100H     -8.264  83.348 -19.494  1.00107.93           C
ATOM  11971  N   ASP D 101     -11.329  81.237 -14.804  1.00107.41           N
```

The insertion codes are the opposite of gaps in the numbering; both are used to make the numbering consistent with a reference sequence (and for the same reason the sequence number can be negative).

Another fields that is blank for most of the atoms is altloc. It is a letter marking an alternative conformation (columns 17, just before the residue name):

```
HETATM  557  O  AHOH A 301      13.464  41.125   8.469  0.50 20.23           O
HETATM  558  O  BHOH A 301      12.554  42.700   8.853  0.50 26.40           O
```

Handling alternative conformations adds a lot of complexity, as it will be described later on in this documentation. These were all tricky things in the atom list.

Now let's go to matrices. In most of the PDB entries the CRYST1 record is all that is needed to construct the crystal structure. But in some PDB files we need to take into account two other records:

- MTRIX – if marked as not-given it defines operations needed to reconstruct the asymmetric unit,
- SCALE – provides fractionalization matrix. The format of this entry is unfortunate: for large unit cells the relative precision of numbers is too small. So if coordinates are given in standard settings it is better to calculate the fractionalization matrix from the unit cell dimensions (i.e. from the CRYST1 record). But the SCALE record needs to be checked to see if the settings *are* the standard ones.

Reading

C++

As described in the previous section, all coordinate files can be read using the same function calls. Additionally, in C++, you may read a selected file format to avoid linking with the code you do not use:

```
#include <gemmi/pdb.hpp>     // to read
#include <gemmi/gz.hpp>      // to uncompress on the fly

gemmi::Structure st1 = gemmi::read_pdb_file(path);
// or
gemmi::Structure st2 = gemmi::read_pdb(gemmi::MaybeGzipped(path));
```

The content of the file can also be read from a string or from memory:

```
Structure read_pdb_string(const std::string& str, const std::string& name);
Structure read_pdb_from_memory(const char* data, size_t size, const std::string&␣
↪name);
```

Python

```
import gemmi

# just use interface common for all file formats
structure = gemmi.read_structure(path)
```

(continues on next page)

(continued from previous page)

```
# or a function that reads only pdb files
structure = gemmi.read_pdb(path)

# if you have the content of the PDB file in a string:
structure = gemmi.read_pdb_string(string)
```

Not all the metadata read from a PDB file is directly accessible from Python. Experimental details, refinement statistics, the secondary structure information, and many other things can be only read indirectly, by first putting it into a cif.Block:

```
>>> st = gemmi.read_structure('../tests/5moo_header.pdb')
>>> block = st.make_mmcif_headers()
>>> block.get_mmcif_category('_diffrn')
{'id': ['1', '2'], 'crystal_id': ['1', '2'], 'ambient_temp': ['295', '295']}
>>> block.get_mmcif_category('_diffrn_radiation')
{'diffrn_id': ['1', '2'], 'pdbx_scattering_type': ['x-ray', 'neutron'], 'pdbx_
↪monochromatic_or_laue_m_l': ['M', None], 'monochromator': [None, None]}
```

PDB files are expected to have 80 columns, although trailing spaces are often not included. Some programs in certain situations produce longer lines, so Gemmi reads lines up to 120 characters. In some old files from the wwPDB snapshots columns 73-80 contain PDB ID and line number (such as "1ABC 205"). It confuses the PDB parser and it is not handled automatically – such files are not in use nowadays. Nevertheless, they can be read by manually limiting the line length:

```
>>> gemmi.read_pdb('../tests/pdb1gdr.ent', max_line_length=72)
<gemmi.Structure pdb1gdr.ent with 1 model(s)>
```

Writing

C++

Function for writing data from Structure to a pdb file are in a header `gemmi/to_pdb.hpp`:

```
void write_pdb(const Structure& st, std::ostream& os,
               PdbWriteOptions opt=PdbWriteOptions());
void write_minimal_pdb(const Structure& st, std::ostream& os);
std::string make_pdb_headers(const Structure& st);
```

Internally, these functions use the stb_sprintf library. And like in stb-style libraries, the implementation of the functions above is guarded by a macro. In exactly one file you need to add:

```
#define GEMMI_WRITE_IMPLEMENTATION
#include <gemmi/to_pdb.hpp>
```

Python

To output a file or string in the PDB format use one of the functions:

```
# To write full PDB use (the options are listed below):
structure.write_pdb(path [, options])

# To write only CRYST1 and coordinates, use:
structure.write_minimal_pdb(path)

# To get the same as a string:
```

(continues on next page)

(continued from previous page)

```
pdb_string = structure.make_minimal_pdb()

# To get PDB headers as a string:
header_string = structure.make_pdb_headers()
```

`write_pdb()` has options to suppress writing of various records, to avoid assigning a serial number to the TER record, and to add use non-standard Refmac LINKR record instead of LINK. Here is the full signature:

```
>>> print(gemmi.Structure.write_pdb.__doc__.replace(',', ',\n          '))
write_pdb(self: gemmi.Structure,
          path: str,
          seqres_records: bool = True,
          ssbond_records: bool = True,
          link_records: bool = True,
          cispep_records: bool = True,
          ter_records: bool = True,
          numbered_ter: bool = True,
          use_linkr: bool = False) -> None
```

1.4.8 PDBx/mmCIF format

The mmCIF format (more formally: PDBx/mmCIF) became the primary format used by the wwPDB. The format uses the CIF 1.1 syntax with semantics described by the PDBx/mmCIF DDL2 dictionary.

While this section may clarify a few things, you do not need to read it to work with mmCIF files.

The main characteristics of the CIF syntax are described in the *CIF introduction*. Here we focus on things specific to mmCIF:

- PDBx/mmCIF dictionary is clearly inspired by relational databases. Categories correspond to tables. Data items correspond to columns. Key data items correspond to primary (or composite) keys in RDBMS.

  While a single block in a single file always describes a single PDB entry, some relations between tables seem to be designed for any number of entries in one block. For example, although a file has only one `_entry.id` and `_struct.title`, the dictionary uses an extra item called `_struct.entry_id` to match the title with id. Is it a good practice to check `_struct.entry_id` before reading `_struct.title`? Probably not, as I have seen files with missing `_struct.entry_id` but never (yet) with multiple `_struct.title`.

- Any category (RDBMS table) can be written as a CIF loop (table). If such a table would have a single row it can be (and always is in wwPDB) written as key-value pairs. So when accessing a value it is safer to use abstraction that hides the difference between a loop and a key-value pair (`cif::Table` in Gemmi).

- Arguably, the mmCIF format is harder to parse than the old PDB format. Using `grep` and `awk` to extract atoms will work only with files written in a specific layout, usually by a particular software. It is unfortunate that the wwPDB FAQ encourages it, so one may expect portability problems when using mmCIF.

- The atoms (`_atom_site`) table has four "author defined alternatives" (`.auth_*`) that have similar meaning to the "primary" identifiers (`.label_*`). Two of them, atom name (`atom_id`) and residue name (`comp_id`) *almost never* differ (update: these few differences were removed from the PDB in 2018). The other two, chain name (`asym_id`) and sequence number (`seq_id`) may differ in a confusing way (A,B,C <-> C,A,B). Which one is presented to the user depends on a program (usually the author's version). This may lead to funny situations.

- There is a formal distinction between mmCIF and PDBx/mmCIF dictionaries (they are controlled by separate committees). The latter is built upon the former. So we have the `pdbx_` prefix in otherwise random places, to mark tags that are not in the vanilla mmCIF.

Here are example lines from a PDB file (3B9F) with the fields numbered at the bottom:

```
ATOM   1033  OE2 GLU H  77      -9.804  19.834 -55.805  1.00 25.54           O
ATOM   1034  N  AARG H  77A     -4.657  24.646 -55.236  0.11 20.46           N
ATOM   1035  N  BARG H  77A     -4.641  24.646 -55.195  0.82 22.07           N
 |       |   |  | |  |  | |       |       |       |      |     |             | |
 1       2   3  4 5  6  7 8       9       10      11     12    13           14 15
```

and the corresponding lines from PDBx/mmCIF v5 (as served by the PDB in 2018):

```
ATOM   1032 O OE2 . GLU B 2  72  ? -9.804  19.834  -55.805 1.00 25.54 ? 77   GLU H␣
↪OE2 1
ATOM   1033 N N   A ARG B 2  73  A -4.657  24.646  -55.236 0.11 20.46 ? 77   ARG H N ␣
↪ 1
ATOM   1034 N N   B ARG B 2  73  A -4.641  24.646  -55.195 0.82 22.07 ? 77   ARG H N ␣
↪ 1
 |       |  | |   |  |  | |   |  |    |       |       |     |    |    |  |    |  | | ␣
↪ |
 1       2 14 x   4  x  x x   x  8    9       10      11    12   13   15 7    5  6 3 ␣
↪ x
 |       |  | |   |  label_comp_id    Cartn_x |       |     |    B_iso_or_equiv  |␣
↪auth_atom_id
 |       id | |   label_alt_id|  pdbx_PDB_ins_code    |     occupancy |  |    |  auth_
↪asym_id
 group_PDB  | label_atom_id   label_seq_id    |       Cartn_z         |  |    auth_
↪comp_id
            type_symbol | label_entity_id     Cartn_y                 |  auth_seq_id ␣
↪ pdbx_PDB_model_num
                        label_asym_id                                 pdbx_formal_
↪charge
```

`x` marks columns not present in the PDB file. The numbers in column 2 differ because in the PDB file the TER record (that marks the end of a polymer) is also assigned a number.

`auth_seq_id` used to be the full author's sequence ID, but currently in the wwPDB entries it is only the sequence number; the insertion code is stored in a separate column (`pdbx_PDB_ins_code`). Confusingly, `pdbx_PDB_ins_code` is placed next to `label_seq_id` not `auth_seq_id` (`label_seq_id` is always a positive number and has nothing to do with the insertion code).

As mentioned above, the mmCIF format has two sets of names/numbers: *label* and *auth* (for "author"). Both atom names (`label_atom_id` and `auth_atom_id`) are normally the same. Both residue names (`label_comp_id` and `auth_comp_id`) are also normally the same. So Gemmi reads and stores only one name: *label* if it is present, otherwise *auth*.

On the other hand, chain names (`asym_id`) and sequence numbers often differ and in the user interface it is better to use the author-defined names, for consistency with the PDB format and with the literature.

While this is not guaranteed by the specification, in all PDB entries each `auth_asym_id` "chain" is split into one or more `label_asym_id` "chains"; let us call them *subchains*. The polymer (residues before the TER record in the PDB format) goes into one subchain; all the other (non-polymer) residues are put into single-residue subchains; except the waters, which are all put into one subchain. Currently, wwPDB treats non-linear polymers (such as sugars) as non-polymers.

Note: Having two sets of identifiers in parallel is not a good idea. Making them look the same so they can be confused is a bad design.

Additionally, the label_* identifiers are not unique: waters have null `label_seq_id` and therefore all waters in one chain have the same identifier. If a water atom is referenced in another table (_struct_conn or _struct_site_gen) the

label_* identifier is ambiguous, so it is necessary to use the auth_* identifier anyway.

This all is quite confusing and lacks a proper documentation. So once again, now in a color-coded version:

and a couple lines from another file (6any):

Each atom site has three independent identifiers:

1. The number in bold is a short and simple one (it does not need to be a number according to the mmCIF spec).
2. The hierarchical identifier from the PDB format (blue background) is what people usually use. Unfortunately, the arbitrary ordering of columns makes it harder to interpret.
3. The new mmCIF identifier (orange) is confusingly similar to 2, but it cannot uniquely identify water atoms, so it cannot be used in every context.

How other tables in the mmCIF file refer to atom sites? Some use both 2 and 3 (e.g. _struct_conn), some use only 2 (e.g. _struct_site), and _atom_site_anisotrop uses all 1, 2 and 3.

Reading

As a reminder, you may use the functions common for all file formats (such as `read_structure_gz()`) to read a structure.

But you may also use two functions that give you more control. These functions correspond to two stages of reading mmCIF files in Gemmi: file → `cif::Document` → `Structure`.

C++

```
#include <gemmi/cif.hpp>       // file -> cif::Document
#include <gemmi/gz.hpp>        // uncompressing on the fly
#include <gemmi/mmcif.hpp>     // cif::Document -> Structure

namespace cif = gemmi::cif;

cif::Document doc = cif::read(gemmi::MaybeGzipped(mmcif_file));
gemmi::Structure structure = gemmi::make_structure(doc);
```

`cif::Document` can be additionally used to access meta-data, such as the details of the experiment or software used for data processing. The examples are provided in the *CIF parser* section.

Python

```
>>> cif_block = gemmi.cif.read(mmcif_path)[0]
>>> structure = gemmi.make_structure_from_block(cif_block)
```

`cif_block` can be additionally used to access meta-data.

Writing

Writing is also in two stages: first a `cif::Document` is created and then it is written to disk.

C++

```
#include <gemmi/to_mmcif.hpp>  // Structure -> cif::Document
#include <gemmi/to_cif.hpp>    // cif::Document -> file

std::ofstream os("new.cif");
gemmi::write_cif_to_file(os, gemmi::make_mmcif_document(structure));
```

Python

```
>>> structure.make_mmcif_document().write_file('new.cif')
```

1.4.9 mmJSON format

The mmJSON format is a JSON representation of the mmCIF data. This format can be easily parsed with any JSON parser (Gemmi uses sajson). It is a good alternative to PDBML – easier to parse and smaller.

Files in this format are available from PDBj using REST API:

```
curl -o 5MOO.json.gz 'https://pdbj.org/rest/downloadPDBfile?id=5MOO&format=mmjson-all'
```

as well as ftp/rsync.

Gemmi reads mmJSON files into `cif::Document`, as it does with mmCIF files.

Reading

C++

```
#include <gemmi/json.hpp>     // JSON -> cif::Document
#include <gemmi/mmcif.hpp>    // cif::Document -> Structure
#include <gemmi/gz.hpp>       // to uncompress on the fly

namespace cif = gemmi::cif;

cif::Document doc = cif::read_mmjson_file(path);
// or, to handle gzipped files:
cif::Document doc = cif::read_mmjson(gemmi::MaybeGzipped(path));
// and then:
gemmi::Structure structure =  gemmi::make_structure(doc);
```

Python

```
>>> # just use interface common for all file formats
>>> structure = gemmi.read_structure(mmjson_path)
>>>
>>> # but you can do it in two steps if you wish
>>> cif_block = gemmi.cif.read_mmjson(mmjson_path)[0]
>>> structure = gemmi.make_structure_from_block(cif_block)
```

Writing

C++

```
#include <gemmi/to_json.hpp>  // for write_mmjson_to_stream

// cif::Document doc = gemmi::make_mmcif_document(structure);
gemmi::write_mmjson_to_stream(ostream, doc);
```

Python

```
>>> # Structure -> cif.Document -> mmJSON
>>> json_str = structure.make_mmcif_document().as_json(mmjson=True)
```

1.4.10 Hierarchy

The most useful representation for working with macromolecular models is a hierarchy of objects. To a first approximation all macromolecular libraries present the same hierarchy: model - chain - residue - atom.

Naming

While *chain* and *residue* are not good names when referring to ligands and waters, we use this nomenclature as it is the most popular one. Some libraries (clipper) call it polymer - monomer - atom. PDBx/mmCIF uses more general (but not so obvious) terms: *entity* and *struct_asym* (structural component in asymetric unit) instead of chain, and *chem_comp* (chemical component) for residue/monomer.

Alternative conformations

Apart from the naming, the biggest difference between libraries is how the disorder is presented. The main options are:

- group together atoms from the same conformer
- group together alternative locations of the same atom (cctbx.iotbx has residue-groups and atom-groups)
- leave it to the user (e.g. mmdb and clipper).

Handling alternative conformations adds significant complexity. Reportedly, "about 90% of the development time invested into iotbx.pdb was in some form related to alternative conformations".

Gemmi exposes the *altloc* field to the user (like mmdb). On top of it it offers utilities that make working with conformers easier:

- functions that ignore all but the main conformation (inspired by BioPython),
- and lightweight proxy objects ResidueGroup and AtomGroup that group alternative conformers (inspired by iotbx).

Discontinuous chains

The usual order of atoms in a file is

- either by chain (A-polymer, A-ligands, A-waters, B-polymer, B-ligands, B-waters)
- or by chain parts (A-polymer, B-polymer, A-ligands, B-ligands, A-waters, B-waters).

In the latter case (example: 100D), chain parts with the same name are either merged automatically (MMDB, BioPython) or left as separate chains (iotbx).

In gemmi we support both ways. Since merging is easier than splitting, the chains are first read separately and after reading the file the user can call `Structure::merge_chain_parts()`.

In the Python interface merging is also controlled by second argument to the `gemmi.read_structure()` function:

```
read_structure(path: str, merge_chain_parts: bool = True) -> gemmi.Structure
```

Example

Next sections document each level of the hierarchy. But first a simple example. The code below iterates over all the hierarchy levels and mutates methionine residues (MET) to selenomethionine (MSE).

C++

```
#include <gemmi/model.hpp>

void met_to_mse(gemmi::Structure& st) {
  for (gemmi::Model& model : st.models)
    for (gemmi::Chain& chain : model.chains)
      for (gemmi::Residue& res : chain.residues)
        if (res.name == "MET") {
          res.name = "MSE";
          for (gemmi::Atom& atom : res.atoms)
            if (atom.name == "SD") {
              atom.name = "SE";
              atom.element = gemmi::El::Se;
            }
        }
}
```

Python

```
import gemmi

def met_to_mse(st: gemmi.Structure) -> None:
    for model in st:
        for chain in model:
            for residue in chain:
                if residue.name == 'MET':
                    residue.name = 'MSE'
                    for atom in residue:
                        if atom.name == 'SD':
                            atom.name = 'SE'
                            atom.element = gemmi.Element('Se')
```

1.4.11 Structure

The object of type Structure that we get from reading a PDB or mmCIF file contains one or more models. This is the top level in the hierarchy: structure - model - chain - residue - atom.

Apart from storing models (usually just a single model) the `Structure` has the following properties:

- `name` (string) – usually the file basename or PDB code,
- `cell` – *unit cell*,
- `spacegroup_hm` (string) – full space group name in Hermann–Mauguin notation (usually taken from the coordinate file),
- `ncs` (C++ type: `vector<NcsOp>`) – list of NCS operations, usually taken from the MTRIX record or from the _struct_ncs_oper category,
- `resolution` (C++ type: `double`) – resolution value from REMARK 2 or 3,
- `entities` (C++ type: `vector<Entity>`) – additional information about *subchains*, such as entity type and polymer's sequence,

- `connections` (C++ type: `vector<Connection>`) – list of connections corresponding to the _struct_conn category in mmCIF, or to the pdb records LINK and SSBOND,
- `assemblies` (C++ type: `vector<Assembly>`) – list of biological assemblies defined in the REMARK 350 in pdb, or in corresponding mmCIF categories (_pdbx_struct_assembly, _pdbx_struct_assembly_gen, _pdbx_struct_assembly_prop and _pdbx_struct_oper_list)
- `info` (C++ type: `map<string, string>`) – minimal metadata with keys being mmcif tags (_entry.id, _exptl.method, ...),
- `raw_remarks` (C++ type: `vector<string>`) – REMARK records from a PDB file, empty if the input file has different format.

In Python, the `info` member variable is a dictionary-like object:

```
>>> for key, value in st.info.items(): print(key, value)
_cell.Z_PDB 4
_entry.id 1ORC
_exptl.method X-RAY DIFFRACTION
_pdbx_database_status.recvd_initial_deposition_date 1995-10-30
_struct.title CRO REPRESSOR INSERTION MUTANT K56-[DGEVK]
_struct_keywords.pdbx_keywords GENE REGULATING PROTEIN
_struct_keywords.text GENE REGULATING PROTEIN
```

Gemmi parses many more records from the PDB format, including REMARK 3 and 200/230. This information is stored in the `Metadata` structure defined in `gemmi/metadata.hpp`. Currently, it's not exposed to Python.

`Structure` has also a number of methods. To access or delete a model with known name use:

```
Model* Structure::find_model(const std::string& model_name)
void Structure::remove_model(const std::string& model_name)
```

In Python these functions are wrapped as `__getitem__` and `__delitem__`:

```
>>> structure[0]         # by 0-based index
<gemmi.Model 1 with 6 chain(s)>
>>> structure['1']       # by name, which is usually a 1-based index as string
<gemmi.Model 1 with 6 chain(s)>
>>> del structure[1:]    # delete all models but the first one
>>> del structure['1']   # delete model "1" (normally, the first one)
```

To add a model to the structure, in C++ use directly methods of:

```
std::vector<Model> Structure::models
```

and in Python use:

```
Structure.add_model(model, pos=-1)
```

for example,

```
structure.add_model(gemmi.Model('7'))   # add a new model
structure.add_model(structure[0])       # add a copy of model #0
```

Warning: Adding and removing models may invalidate references to other models from the same Structure. This is expected when working with a C++ vector, but when using Gemmi from Python it is a flaw. More precisely:

- `add_model` may cause memory re-allocation invalidating references to all other models,

- `remove_model` and `__delitem__` invalidate references only to models that are after the removed one.

This means that you need to update a reference before using it:

```
model_reference = st[0]
st.add_model(...)          # model_reference gets invalidated
model_reference = st[0]    # model_reference is valid again
```

The same rules apply to functions that add and remove chains, residues and atoms (`add_chain`, `add_residue`, `add_atom`, `__delitem__`).

After adding or removing models you may call:

```
>>> structure.renumber_models()
```

which will set model *names* to sequential numbers (next section explains why models have names).

Entity

Entity is a new concept introduced in the mmCIF format. If the structure is read from a PDB file, we can assign entities by calling method `setup_entities`. This method uses a simple heuristic to group residues into *subchains* which are mapped to entities (this is primarily about finding where the polymer ends; works best if the TER record is used). All polymers with identical sequence in the SEQRES record are mapped to the same entity.

Calling `setup_entities` is useful when converting from PDB to mmCIF (but to just convert files use *gemmi-convert*):

```
>>> st = gemmi.read_structure('../tests/1orc.pdb')
>>> st.setup_entities()
>>> st.make_mmcif_document().write_file('out.cif')
```

The Entity object may change in the future. Here we only show its properties in an example:

```
>>> for entity in st.entities: print(entity)
<gemmi.Entity 'A' polymer polypeptide(L) object at 0x...>
<gemmi.Entity 'water' water object at 0x...>
>>> ent = st.entities[0]
>>> ent.name
'A'
>>> ent.subchains
['Apoly']
>>> ent.entity_type
EntityType.Polymer
>>> ent.polymer_type
PolymerType.PeptideL
>>> ent.full_sequence[:5]
['MET', 'GLU', 'GLN', 'ARG', 'ILE']
```

The last property is sequence from the PDB SEQRES record (or mmCIF equivalent). More details in the *section about sequence*.

Connection

The list of connections contains bonds explicitely annotated in the file:

```
>>> st = gemmi.read_structure('../tests/4oz7.pdb')
>>> st.connections[0]
<gemmi.Connection disulf1  A/CYS 4/SG - A/CYS 10/SG>
>>> st.connections[2]
<gemmi.Connection covale1  A/22Q 1/C - A/ALA 2/N>
>>> st.connections[-1]
<gemmi.Connection metalc8  B/22Q 1/S - A/CU1 101/CU>
```

You can find connection between two atoms, or check if it exists, by specifying two *atom addresses*:

```
>>> addr1 = gemmi.AtomAddress(chain='B', seqid=gemmi.SeqId('4'), resname='CYS', atom=
↪'SG')
>>> addr2 = gemmi.AtomAddress('B', gemmi.SeqId('10'), 'CYS', atom='SG')
>>> st.find_connection(addr1, addr2)
<gemmi.Connection disulf2  B/CYS 4/SG - B/CYS 10/SG>
```

Each connection stores:

- type – corresponding to _struct_conn.type in the mmCIF format; one of enumeration values: Covale, Disulf, Hydrog, MetalC, None; when reading PDB format the SSBOND record corresponds to Disulf, LINK records – to Covale or MetalC,

  ```
  >>> st.connections[0].type
  ConnectionType.Disulf
  ```

- name – a unique name corresponding to _struct_conn.id in the mmCIF format; it is auto-generated the connections are read from the PDB format,

  ```
  >>> st.connections[0].name
  'disulf1'
  ```

- optionally, ID of the link used to restrain this bond during refinement (_chem_link.id from the CCP4 monomer library), written as _struct_conn.ccp4_link_id in mmCIF,

  ```
  >>> st.connections[0].link_id  # no link ID -> empty string
  ''
  ```

- addresses of two atoms (`partner1` and `partner2`),

  ```
  >>> st.connections[2].partner2
  <gemmi.AtomAddress A/ALA 2/N>
  ```

- a flag that for connections between different symmetry images,

  ```
  >>> st.connections[2].asu
  Asu.Same
  >>> st.connections[-1].asu
  Asu.Different
  ```

- and a distance read from the file.

  ```
  >>> st.connections[-1].reported_distance
  2.22
  ```

When the connection is written to a file, the symmetry image and the distance are recalculated like this:

```
>>> con = st.connections[-1]
>>> pos1 = st[0].find_cra(con.partner1).atom.pos
>>> pos2 = st[0].find_cra(con.partner2).atom.pos
>>> st.cell.find_nearest_image(pos1, pos2, con.asu)
<gemmi.SymImage box:[2, 1, 1] sym:5>
>>> _.dist()
2.22115330402924
```

The vast majority of connections is intramolecular, so usually you get:

```
>>> st.cell.find_nearest_image(pos1, pos2, con.asu)
<gemmi.SymImage box:[0, 0, 0] sym:0>
```

The section about *AtomAddress* has an example that shows how to create a new connection.

Assembly

Biological assemblies are nicely introduced in PDB-101. Description of a biological assembly read from a coordinate file is represented in Gemmi by the `Assembly` class. It contains a recipe how to construct the assembly from a model. In the PDB format, REMARK 350 says what operations should be applied to what chains. In the PDBx/mmCIF format it is similar, but the *subchains* are used instead of chains.

```
>>> for assembly in st.assemblies:
...   print(assembly.name, assembly.oligomeric_details)
1 MONOMERIC
2 MONOMERIC
```

As always, naming things is hard. Biological unit may contain a number of copies of one chain. Each copy needs to be named. Gemmi provides three options:

- HowToNameCopiedChains.Dup (in C++: HowToNameCopiedChains::Dup) – simply leaves the original chain name in all copies,
- HowToNameCopiedChains.AddNumber – copies of chain A are named A1, A2, ..., copies of chain B – B1, B2, ..., etc,
- HowToNameCopiedChains.Short – unique one-character chain names are used until exhausted (after 26*2+10=62 chains), then two-character names are used. This option is appropriate when the output is to be stored in the PDB format.

Function `make_assembly` takes Model and one of the naming options above, and returns a new Model that represents the assembly. In C++ this function is in `<gemmi/assembly.hpp>`.

```
>>> st.assemblies[0].make_assembly(st[0], gemmi.HowToNameCopiedChains.AddNumber)
<gemmi.Model 1 with 1 chain(s)>
>>> list(_)
[<gemmi.Chain A1 with 21 res>]
>>> st.assemblies[1].make_assembly(st[0], gemmi.HowToNameCopiedChains.AddNumber)
<gemmi.Model 1 with 1 chain(s)>
>>> list(_)
[<gemmi.Chain B1 with 26 res>]
```

See also the `--assembly` option in command-line program *gemmi-convert*.

Common operations

In Python, `Structure` has also methods for more specialized, but often needed operations:

```
>>> st.remove_alternative_conformations()
>>> st.remove_hydrogens()
>>> st.remove_waters()
>>> st.remove_ligands_and_waters()
>>> st.remove_empty_chains()
```

In C++ the functions above are provided in `gemmi/polyheur.hpp`. They are implemented as templated free functions that can be applied not only to `Structure`, but also to `Model` and `Chain`.

Occasionally, you may come across an mmCIF file with chain names longer than necessary. To store such structure in a PDB format you need to shorten the chain names first:

```
>>> st.shorten_chain_names()
```

In C++ this functions is in `gemmi/assembly.hpp`.

1.4.12 Sequence

In the previous section we introduced sequence with the following example:

```
>>> ent.full_sequence[:5]
['MET', 'GLU', 'GLN', 'ARG', 'ILE']
```

`Entity.full_sequence` is a list (in C++: `std::vector`) of residue names. It stores sequence from the SEQRES record (pdb) or from the _entity_poly_seq category (mmCIF). The latter can contain microheterogeneity (point mutation). In such case, the residue names at the same point in sequence are separated by commas:

```
>>> st = gemmi.read_structure('../tests/1pfe.cif.gz')
>>> seq = st.get_entity('2').full_sequence
>>> seq
['DSN', 'ALA', 'N2C,NCY', 'MVA', 'DSN', 'ALA', 'NCY,N2C', 'MVA']
>>> #           ^^^^^^^  microheterogeneity     ^^^^^^^
```

To ignore point mutations we can use a helper function `Entity::first_mon`:

```
>>> [gemmi.Entity.first_mon(item) for item in seq]
['DSN', 'ALA', 'N2C', 'MVA', 'DSN', 'ALA', 'NCY', 'MVA']
```

An example in the section about Chain shows how to *extract corresponding sequence from the model*. In general, the sequence in SEQRES and the sequence in model differ, but in this file they are the same.

To get a sequence as one-letter codes you can use the *built-in table* of popular residues:

```
>>> [gemmi.find_tabulated_residue(resname).one_letter_code for resname in _]
['s', 'A', ' ', 'v', 's', 'A', ' ', 'v']
```

`one_letter_code` is lowercase for non-standard residues where it denotes the parent component. If the code is blank, either the parent component is not known, or the component is not tabulated in Gemmi (i.e. it's not in the top 300+ most popular components in the PDB). To get a FASTA-like string, you could continue the previous line with:

```
>>> ''.join((code if code.isupper() else 'X') for code in _)
'XAXXXAXX'
```

or use:

```
>>> gemmi.one_letter_code(seq)
'XAXXXAXX'
```

To go in the opposite direction, use:

```
>>> [gemmi.expand_protein_one_letter(letter) for letter in _]
['UNK', 'ALA', 'UNK', 'UNK', 'UNK', 'ALA', 'UNK', 'UNK']
```

or

```
>>> gemmi.expand_protein_one_letter_string('XAXXXAXX')
['UNK', 'ALA', 'UNK', 'UNK', 'UNK', 'ALA', 'UNK', 'UNK']
```

Molecular weight

Gemmi provides a simple function to calculate molecular weight from the sequence. It uses the same built-in table of popular residues. Since in this example we have two rare components that are not tabulated, we must specify the avarage weight of unknown residue:

```
>>> gemmi.calculate_sequence_weight(seq, unknown=130.0)
910.7184143066406
```

In such case the result is not accurate, but this is not a typical case.

Now we will take a PDB file with standard residues and calculate the Matthews coefficient:

```
>>> st = gemmi.read_structure('../tests/5cvz_final.pdb')
>>> list(st[0])
[<gemmi.Chain A with 141 res>]
>>> # we have just a single chain, which makes this example simpler
>>> chain = st[0]['A']
>>> chain.get_polymer()
<gemmi.ResidueSpan of 0: []>
>>> # Not good. The chain parts where not assigned automatically,
>>> # because of the missing TER record in this file. We need to call:
>>> st.setup_entities()  # it should sort out chain parts
>>> chain.get_polymer()
<gemmi.ResidueSpan of 141: [17(ALA) 18(ALA) 19(ALA) ... 157(SER)]>
>>> st.get_entity_of(_)
<gemmi.Entity 'A' polymer polypeptide(L) object at 0x...>
>>> weight = gemmi.calculate_sequence_weight(_.full_sequence)
>>> # Now we can calculate Matthews coefficient
>>> st.cell.volume_per_image() / weight
2.7407915442022364
```

We could continue and calculate the solvent content, assuming the protein density of 1.35 g/cm^3 (the other constants below are the Avogadro number and Å^3/cm^3 = 10^{-24}):

```
>>> protein_fraction = 1. / (6.02214e23 * 1e-24 * 1.35 * _)
>>> print('Solvent content: {:.1f}%'.format(100 * (1 - protein_fraction)))
Solvent content: 55.1%
```

Gemmi also includes a program that calculates the solvent content: *gemmi-contents*.

Sequence alignment

Gemmi includes a sequence alignment algorithm based on the simplest function (ksw_gg) from the ksw2 project of Heng Li.

It is a pairwise, global alignment with substitution matrix (or just match/mismatch values) and affine gap penalty. Additionally, in Gemmi the gap openings at selected positions can be made free.

Let say that we want to align residues in the model to the full sequence. Sometimes, the alignment is ambiguous. If we'd align texts ABBC and ABC, both A-BC and AB-C would have the same score. In a 3D structure, the position of gap can be informed by inter-atomic distances. This information is used automatically in the `align_sequence_to_polymer` function. Gap positions, determined by a simple heuristic, are passed to the alignment algorithm as places where the gap opening penalty is not to be imposed.

```
>>> st = gemmi.read_pdb('../tests/pdb1gdr.ent', max_line_length=72)
>>> result = gemmi.align_sequence_to_polymer(st.entities[0].full_sequence,
...                                          st[0][0].get_polymer(),
...                                          gemmi.PolymerType.PeptideL)
```

The arguments of this functions are: sequence (a list of residue names), *ResidueSpan* (a span of residues in a chain), and the type of chain, which is used to infer gaps. (The type can be taken from Entity.polymer_type, but in this example we wanted to keep things simple).

The result provides statistics and methods of summarizing the alignment:

```
>>> result
<gemmi.AlignmentResult object at 0x...>

>>> # score calculated according AlignmentScoring explained below
>>> result.score
69

>>> # number of matching (identical) residues
>>> result.match_count
105
>>> # identity = match count / length of the shorter sequence
>>> result.calculate_identity()
100.0
>>> # identity wrt. the 1st sequence ( = match count / 1st sequence length)
>>> result.calculate_identity(1)
75.0
>>> # identity wrt. the 2nd sequence
>>> result.calculate_identity(2)
100.0

>>> # CIGAR = Concise Idiosyncratic Gapped Alignment Report
>>> result.cigar_str()
'11M3I23M7I71M25I'
```

To print out the alignment, we can combine function `add_gaps` and property `match_string`:

```
>>> result.add_gaps(gemmi.one_letter_code(st.entities[0].full_sequence), 1)[:70]
'MRLFGYARVSTSQQSLDIQVRALKDAGVKANRIFTDKASGSSSDRKGLDLLRMKVEEGDVILVKKLDRLG'
>>> result.match_string[:70]
'||||||||||   ||||||||||||||||||||||       ||||||||||||||||||||||||'
>>> result.add_gaps(gemmi.one_letter_code(st[0][0].get_polymer()), 2)[:70]
'MRLFGYARVST---SLDIQVRALKDAGVKANRIFTDK-------RKGLDLLRMKVEEGDVILVKKLDRLG'
```

or we can use function `AlignmentResult.formatted()`.

We also have a function that aligns two sequences. We can exercise it by comparing two strings:

```
>>> result = gemmi.align_string_sequences(list('kitten'), list('sitting'), [])
```

The third argument above is a list of free gap openings. Now we can visualize the match:

```
>>> print(result.formatted('kitten', 'sitting'), end='')
kitten-
.|||.|
sitting
>>> result.score
0
```

The alignment and the score is calculate according to AlignmentScoring, which can be passed as the last argument to both `align_string_sequences` and `align_sequence_to_polymer` functions. The default scoring is +1 for match, -1 for mismatch, -1 for gap opening, and -1 for each residue in the gap. If we would like to calculate the Levenshtein distance, we would use the following scoring:

```
>>> scoring = gemmi.AlignmentScoring()
>>> scoring.match = 0
>>> scoring.mismatch = -1
>>> scoring.gapo = 0
>>> scoring.gape = -1
>>> gemmi.align_string_sequences(list('kitten'), list('sitting'), [], scoring)
<gemmi.AlignmentResult object at 0x...>
>>> _.score
-3
```

So the distance is 3, as expected.

In addition to the scoring parameters above, we can define a substitution matrix. Gemmi includes ready-to-use BLOSUM62 matrix with the gap cost 10/1, like in BLAST.

```
>>> blosum62 = gemmi.prepare_blosum62_scoring()
>>> blosum62.gapo, blosum62.gape
(-10, -1)
```

Now we can test it on one of examples from the BioPython tutorial. First, we try global alignment:

```
>>> result = gemmi.align_string_sequences(
...         gemmi.expand_protein_one_letter_string('LSPADKTNVKAA'),
...         gemmi.expand_protein_one_letter_string('PEEKSAV'),
...         [], blosum62)
>>> print(result.formatted('LSPADKTNVKAA', 'PEEKSAV'), end='')
LSPADKTNVKAA
  |..|.   |.
--PEEKS---AV
>>> result.score
-7
```

We have only global aligment available, but we can use free-gaps to approximate a semi-global alignment (infix method) where gaps at the start and at the end of the second sequence are not penalized. Approximate – because only gap openings are not penalized, residues in the gap still decrease the score:

```
>>> result = gemmi.align_string_sequences(
...         gemmi.expand_protein_one_letter_string('LSPADKTNVKAA'),
...         gemmi.expand_protein_one_letter_string('PEEKSAV'),
...         # free gaps at 0 (start) and 7 (end):   01234567
```

(continues on next page)

(continued from previous page)

```
...          [i in (0, 7) for i in range(8)],
...          blosum62)
>>> print(result.formatted('LSPADKTNVKAA', 'PEEKSAV'), end='')
LSPADKTNVKAA
  |..|..|
--PEEKSAV---
>>> result.score
11
```

The real infix method (or local alignment) would yield the score 16 (11+5), because we have 5 missing residues at the ends.

See also the *gemmi-align* program.

1.4.13 Model

Model contains chains (class `Chain`) that can be accessed by index or by name:

```
// to access or delete a chain by index use directly the chains vector:
std::vector<Chain> Model::chains
// to access or delete a chain by name use functions:
Chain* Model::find_chain(const std::string& chain_name)
void Model::remove_chain(const std::string& chain_name)
```

```
>>> model = gemmi.read_structure('../tests/1orc.pdb')[0]
>>> model
<gemmi.Model 1 with 1 chain(s)>
>>> model[0]
<gemmi.Chain A with 121 res>
>>> model['A']
<gemmi.Chain A with 121 res>
>>> del model['A']  # deletes chain A
```

As it was shown in the *MET to MSE example*, you can iterate over chains in the model. You can also use function `all()` to iterate over all atoms in the model, getting objects of the *CRA* class which holds three pointers – chain, residue and atom. The function mutating MET to MSE could be alternatively implemented as:

```
def met_to_mse2(st: gemmi.Structure) -> None:
    for model in st:
        for cra in model.all():
            if cra.residue.name == 'MET' and cra.atom.name == 'SD':
                cra.residue.name = 'MSE'
                cra.atom.name = 'SE'
                cra.atom.element = gemmi.Element('Se')
```

To add a chain to the model, in C++ use directly methods of `Model::chains` and in Python use:

```
Model.add_chain(chain, pos=-1)
```

for example,

```
model.add_chain(gemmi.Chain('E'))  # add a new (empty) chain
model.add_chain(model[0])          # add a copy of chain #0
```

Each `Model` in a Structure must have a unique name (`string name`). Normally, models are numbered and the name is a number. But according to the mmCIF spec the name does not need to be a number, so just in case we store it as a string.

```
>>> model.name
'1'
```

As was discussed before, the PDBx/mmCIF format has also a set of parallel identifiers. In particular, it has `label_asym_id` in parallel to `auth_asym_id`. In Gemmi the residues with the same `label_asym_id` are called *subchain*. Subchain is represented by class `ResidueSpan`. If you want to access a subchain with the specified `label_asym_id`, use:

```
Model::get_subchain(const std::string& sub_name) -> ResidueSpan
```

```
>>> model = gemmi.read_structure('../tests/1pfe.cif.gz')[0]
>>> model.get_subchain('A')
<gemmi.ResidueSpan of 8: [1(DG) 2(DC) 3(DG) ... 8(DC)]>
```

To get the list of all subchains in the model, use:

```
Model::subchains() -> std::vector<ResidueSpan>
```

```
>>> [subchain.subchain_id() for subchain in model.subchains()]
['A', 'C', 'F', 'B', 'D', 'E', 'G']
```

The subchains got re-ordered when the chain parts were merged. Alternatively, we could do:

```
>>> model = gemmi.read_structure('../tests/1pfe.cif.gz', merge_chain_parts=False)[0]
>>> [subchain.subchain_id() for subchain in model.subchains()]
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

The `ResidueSpan` is described in the next section.

In Python, `Model` has also methods for often needed calculations:

```
>>> model.count_atom_sites()
342
>>> model.count_occupancies()
302.9999997317791
>>> model.calculate_mass()
4395.826034891504
>>> model.calculate_center_of_mass()
<gemmi.Position(-5.7572, 16.4099, 2.88299)>
```

In C++, the same functionality is provided by templated functions from `gemmi/calculate.hpp`. These functions (in C++) can be applied not only to `Model`, but also to `Structure`, `Chain` and `Residue`.

1.4.14 Chain

Chain corresponds to the chain in the PDB format and to `_atom_site.auth_asym_id` in the mmCIF format. It has a name and a list of residues (class `Residue`).

To get the name or access a residue by index, in C++ you may access these properties directly:

```
std::string name;
std::vector<Residue> residues;
```

In Python, we also have the `name` property:

```
>>> model = gemmi.read_structure('../tests/1pfe.cif.gz')[0]
>>> chain_a = model['A']
>>> chain_a.name
'A'
```

but the residues are accessed by iterating or indexing directly the chain object:

```
>>> chain_a[0]   # first residue
<gemmi.Residue 1(DG) with 23 atoms>
>>> chain_a[-1]  # last residue
<gemmi.Residue 2070(HOH) with 1 atoms>
>>> len(chain_a)
79
>>> sum(res.is_water() for res in chain_a)
70
```

To add a residue to the chain, in C++ use directly methods of `Chain::residues` and in Python use:

```
Chain.add_residue(residue, pos=-1)
```

for example,

```
>>> # add a copy of the first residue at the end
>>> chain_a.add_residue(chain_a[0])
<gemmi.Residue 1(DG) with 23 atoms>
>>> # and then delete it
>>> del chain_a[-1]
```

In the literature, residues are referred to by sequence ID (number and, optionally, insertion code) and residue name. To get residues with with the specified sequence ID use indexing with a string as an argument:

```
>>> chain_a['1']
<gemmi.ResidueGroup [1(DG)]>
```

The returned object is a ResidueGroup with a single residue, unless we have a point mutation. The ResidueGroup is documented later on. For now let's only show how to extract the residue we want:

```
>>> chain_a['1']['DG']   # gets residue DG
<gemmi.Residue 1(DG) with 23 atoms>
>>> chain_a['1'][0]      # gets first residue in the group
<gemmi.Residue 1(DG) with 23 atoms>
```

Often, we need to refer to a part of the chain. A span of consecutive residues can be represented by `ResidueSpan`. For example, if we want to process separately the polymer, ligand and water parts of the chain, we can use the following functions that return `ResidueSpan`:

```
ResidueSpan Chain::get_polymer()
ResidueSpan Chain::get_ligands()
ResidueSpan Chain::get_waters()
```

```
>>> chain_a.get_polymer()
<gemmi.ResidueSpan of 8: [1(DG) 2(DC) 3(DG) ... 8(DC)]>
>>> chain_a.get_ligands()
<gemmi.ResidueSpan of 1: [20(CL)]>
>>> chain_a.get_waters()
<gemmi.ResidueSpan of 70: [2001(HOH) 2002(HOH) 2003(HOH) ... 2070(HOH)]>
```

Note: This is possible because, conventionally, polymer is at the beginning of the chain, waters are at the end, and ligands are in the middle. It won't work if for some reasons the residues of different categories are intermixed.

We also have a function that returns the whole chain as a residue span:

```
ResidueSpan Chain::whole()
```

```
>>> chain_a.whole()
<gemmi.ResidueSpan of 79: [1(DG) 2(DC) 3(DG) ... 2070(HOH)]>
```

`Chain` has also functions `get_subchain()` and `subchains()` that do the same as the functions of `Model` with the same names:

```
>>> [subchain.subchain_id() for subchain in model['A'].subchains()]
['A', 'C', 'F']
>>> [subchain.subchain_id() for subchain in model['B'].subchains()]
['B', 'D', 'E', 'G']
```

Now let us consider microheterogeneities (point mutations). They are less frequent than alternative conformations of atoms in a residue, but we still need to handle them. So we have two approaches, as mentioned before in the section about *alternative conformations*.

For quick and approximate analysis of the structure, one may get by with ignoring all but the first (main) conformer. Both `Chain` and `ResidueSpan` have function `first_conformer()` which returns iterator over residues of the main conformer.

```
>>> polymer_b = model['B'].get_polymer()
>>> # iteration goes through all residues and atom sites
>>> [res.name for res in polymer_b]
['DSN', 'ALA', 'N2C', 'NCY', 'MVA', 'DSN', 'ALA', 'NCY', 'N2C', 'MVA']
>>> # The two pairs N2C/NCY above are alternative conformations.
>>> # Sometimes we want to ignore alternative conformations:
>>> [res.name for res in polymer_b.first_conformer()]
['DSN', 'ALA', 'N2C', 'MVA', 'DSN', 'ALA', 'NCY', 'MVA']
```

A more complex approach is to group together the alternatives. Such a group is represented by `ResidueGroup`, which is derived from `ResidueSpan`.

```
>>> for group in polymer_b.residue_groups():
...     print(','.join(residue.name for residue in group), end='  ')
DSN  ALA  N2C,NCY  MVA  DSN  ALA  NCY,N2C  MVA
```

In Python, Chain has a few specialized, but commonly used functions. Three that are present also in the Model class:

```
>>> chain_a.count_atom_sites()
242
>>> chain_a.count_occupancies()
216.9999997317791
>>> chain_a.calculate_mass()
3211.093834891507
```

and a function that changes a polypeptide chain into polyalanine:

```
>>> chain_a.trim_to_alanine()
```

In C++ `trim_to_alanine()` is defined in `gemmi/polyheur.hpp`.

1.4.15 ResidueSpan, ResidueGroup

ResidueSpan and ResidueGroup are lightweight structures that point to a consecutive span of residues in a chain. But as was was shown in the previous section, they are used for different things.

Both allow addressing residue by (0-based) index:

```
>>> # in the following examples we use polymer_b from the previous section
>>> polymer_b
<gemmi.ResidueSpan of 10: [1(DSN) 2(ALA) 3(N2C) ... 8(MVA)]>
>>> polymer_b[1]   # gets residue by index
<gemmi.Residue 2(ALA) with 5 atoms>
```

You can iterate over residues, although for ResidueSpan it may be better to iterate only over one conformer:

```
>>> # iterating over all residues
>>> for res in polymer_b: print(res.name, end=' ')
DSN ALA N2C NCY MVA DSN ALA NCY N2C MVA
>>> # iterating over primary (first) conformer
>>> for res in polymer_b.first_conformer(): print(res.name, end=' ')
DSN ALA N2C MVA DSN ALA NCY MVA
```

Related to this, the length can be calculating in two ways:

```
>>> len(polymer_b)       # number of residues
10
>>> polymer_b.length()   # length of the chain (which has 2 point mutations)
8
```

The functions for adding and removing residues are the same as in Chain:

```
>>> # add a new (empty) residue at the beginning
>>> polymer_b.add_residue(gemmi.Residue(), 0)
<gemmi.Residue ?() with 0 atoms>
>>> # and delete it
>>> del polymer_b[0]
```

If ResidueSpan represents a subchain we can get its ID (`label_asym_id`):

```
>>> polymer_b.subchain_id()
'B'
```

If it's a polymer, we can ask for polymer type and sequence:

```
>>> polymer_b.check_polymer_type()
PolymerType.PeptideL
>>> polymer_b.make_one_letter_sequence()
'sAXvsAXv'
```

(In C++ these two functions are available in `gemmi/polyheur.hpp`.)

In addition to the numeric indexing, `ResidueSpan.__getitem__` (like `Chain.__getitem__`) can take sequence ID as a string, returning ResidueGroup. In ResidueGroup we can uniquely address a residue by name, therefore `ResidueGroup.__getitem__` (and `__delitem__`) takes residue name.

```
>>> polymer_b['2']  # ResidueSpan[sequence ID] -> ResidueGroup
<gemmi.ResidueGroup [2(ALA)]>
>>> _['ALA']        # ResidueGroup[residue name] -> Residue
<gemmi.Residue 2(ALA) with 5 atoms>
```

1.4.16 Residue

Residue contains atoms (class `Atom`).

From C++ you may access directly the list of atoms:

```
std::vector<Atom> Residue::atoms
```

Or you may use helper functions that take: atom name, alternative location (`'*'` = take the first matching atom regardless of altloc, `'\0'` = no altloc) and, optionally, also the expected element if you want to verify it:

```
Atom* Residue::find_atom(const std::string& atom_name, char altloc, El el=El::X)
std::vector<Atom>::iterator Residue::find_atom_iter(const std::string& atom_name,␣
↪char altloc, El el=El::X)
```

If atom is not found, the first function return `nullptr`, the second one throws exception.

To get all atoms with given name as `AtomGroup` (most often it will be just a single atom) use `Residue::get(const std::string& name)`.

In Python it is similar (but `__getitem__` is used instead of `get()`):

```
>>> residue = polymer_b['2']['ALA']
>>> residue
<gemmi.Residue 2(ALA) with 5 atoms>
>>> residue[0]
<gemmi.Atom N at (-9.9, 10.9, 13.5)>
>>> residue[-1]
<gemmi.Atom CB at (-10.6, 9.7, 11.5)>
>>> residue.find_atom('CA', '*')
<gemmi.Atom CA at (-9.5, 10.0, 12.5)>
>>> residue['CA']
<gemmi.AtomGroup CA, sites: 1>

>>> # Residue also has ``__contains__`` and ``__iter__``
>>> 'CB' in residue
True
>>> ' '.join(a.name for a in residue)
'N CA C O CB'
```

Atoms can be added, modified and removed:

```
>>> new_atom = gemmi.Atom()
>>> new_atom.name = 'HA'
>>> residue.add_atom(new_atom, 2)  # added at (0-based) position 2
<gemmi.Atom HA at (0.0, 0.0, 0.0)>
>>> del residue[2]
```

Residue contains also a number of properties:

- `name` – residue name, such as ALA,
- `seqid` – sequence ID, class SeqId with two properties:
  - `num` – sequence number,
  - `icode` – insertion code (a single character, `' '` = none),
- `segment` – segment from the PDB format v2,
- `subchain` – label_asym_id from mmCIF file, or ID generated by `Structure.assign_subchains()`,
- `label_seq` – numeric value from the label_seq_id field.
- `entity_type` – one of EntityType.Unknown, Polymer, NonPolymer, Water,
- `het_flag` – a single character based on the PDB record or on the _atom_site.group_PDB field: `A`=ATOM, `H`=HETATM, `\0`=unspecified,

```
>>> residue.seqid.num, residue.seqid.icode
(2, ' ')
>>> residue.subchain
'B'
>>> residue.label_seq
2
>>> residue.entity_type
EntityType.Polymer
>>> residue.het_flag
'A'
```

To check if a residue is water (normal or heavy) you may use a helper function:

```
>>> residue.is_water()
False
```

Classes Chain and ResidueSpan have function `first_conformer()` for iterating over residues of one conformer. Similarly, `Residue::first_conformer()` iterates over atoms of a single conformer:

```
>>> residue = chain_a[0]
>>> for atom in residue: print(atom.name, end=' ')
O5' C5' C4' C4' O4' C3' C3' O3' O3' C2' C2' C1' N9 C8 N7 C5 C6 O6 N1 C2 N2 N3 C4
>>> for atom in residue.first_conformer(): print(atom.name, end=' ')
O5' C5' C4' O4' C3' O3' C2' C1' N9 C8 N7 C5 C6 O6 N1 C2 N2 N3 C4
```

1.4.17 AtomGroup

AtomGroup represents alternative locations of the same atom. It is implemented as a lightweight object that points to a consecutive atoms (atom sites) inside the same Residue. It has minimal functionality:

```
>>> residue["O5'"]
<gemmi.AtomGroup O5', sites: 1>
>>> _.name()
"O5'"
>>> len(residue["O5'"])
1

>>> residue["O3'"]
<gemmi.AtomGroup O3', sites: 2>
>>> residue["O3'"][0]    # get atom site by index
<gemmi.Atom O3'.A at (-8.3, 20.3, 17.9)>
>>> residue["O3'"]['A']  # get atom site by altloc
<gemmi.Atom O3'.A at (-8.3, 20.3, 17.9)>
>>> for a in residue["O3'"]: print(a.altloc, end=' ')
A B
```

1.4.18 Atom

Atom (more accurately: atom site) has the following properties:

- `name` – atom name, such as `CA` or `CB`,
- `altloc` – alternative location indicator (one character),
- `charge` – integer number (partial charges are not supported),
- `element` – *element* from a periodic table,
- `pos` – coordinates in Angstroms (instance of `Position`),
- `occ` – occupancy,
- `b_iso` – isotropic temperature factor or, more accurately, atomic displacement parameter (ADP),
- `aniso` – anisotropic atomic displacement parameters (U not B).
- `serial` – atom serial number (integer).
- `flag` – custom flag, a single character that can be used for anything by the user.

These properties can be read and written from both C++ and Python, as was shown in *the example* where sulfur was mutated to selenium.

```
>>> atom = polymer_b['2']['ALA']['CA'][0]
>>> atom.name
'CA'
>>> atom.element
<gemmi.Element: C>
>>> atom.pos
<gemmi.Position(-9.498, 10.028, 12.461)>
>>> atom.occ
1.0
>>> atom.b_iso
9.4399995803833
>>> atom.charge
0
>>> atom.serial
179
>>> atom.flag
'\x00'
```

`altloc` is stored as a single character. Majority of atoms has a single conformations and the altloc character set to NUL (`'\0'`). If you want to check if an atom has non-NUL altloc, you may also use method `has_altloc()`:

```
>>> atom.altloc
'\x00'
>>> atom.has_altloc()
False
```

`element` can be compared (==, !=) with other instances of gemmi.Element. For checking if it is a hydrogen we have a dedicated function `is_hydrogen()` which returns true for both H and D:

```
>>> atom.element == gemmi.Element('C')
True
>>> atom.is_hydrogen()
False
```

B-factors – atomic displacement parameters.

The PDB format stores isotropic ADP as B and anisotropic as U ($B = 8\pi^2 U$). So is Gemmi:

```
>>> atom.b_iso
9.4399995803833
>>> atom.aniso.nonzero()  # has non-zero anisotropic ADP
True
>>> '%g %g %g' % (atom.aniso.u11, atom.aniso.u22, atom.aniso.u33)
'0.1386 0.1295 0.0907'
>>> '%g %g %g' % (atom.aniso.u12, atom.aniso.u23, atom.aniso.u23)
'-0.0026 0.0068 0.0068'
>>> U_eq = atom.aniso.trace() / 3
>>> from math import pi
>>> '%g ~= %g' % (atom.b_iso, 8 * pi**2 * U_eq)
'9.44 ~= 9.44324'
```

Anisotropic models also contain B_{iso}, which should be a full isotropic B-factor. But, as discussed in the BDB paper, some PDB entries contain "residual" B-factors instead. Moreover, "full isotropic ADP" can mean different things. Usually, B_{eq} is used ($B_{eq} \sim \mathrm{tr}(U_{ij})$). But because B_{eq} tends to give values larger than the B-factors that would be obtained in isotropic refinement, Ethan Merrit proposed a metric named B_{est}, more similar to the would-be isotropic Bs. Gemmi can calculate both:

```
>>> atom.b_eq()                    # B_eq
9.443238117199861
>>> gemmi.calculate_b_est(atom)    # B_est
9.15448356208817
```

1.4.19 AtomAddress and CRA

Atoms are often referred to by specifying their chain, residue, atom name and, optionally, altloc. In gemmi, a structure to store such a specification is called AtomAddress. For instance, the following line from a PDB file:

```
LINK         C   22Q A   1                N   ALA A   2     1555   1555  1.34
```

corresponds to Connection that contains two addresses:

```
>>> st = gemmi.read_structure('../tests/4oz7.pdb')
>>> st.connections[2]
<gemmi.Connection covale1  A/22Q 1/C - A/ALA 2/N>
```

Let us check the properties of the second address:

```
>>> addr = _.partner2
>>> addr
<gemmi.AtomAddress A/ALA 2/N>
>>> addr.chain_name
'A'
>>> addr.res_id
<gemmi.ResidueId 2(ALA)>
>>> addr.res_id.seqid
<gemmi.SeqId 2>
>>> addr.res_id.name
'ALA'
>>> addr.atom_name
'N'
>>> addr.altloc
'\x00'
```

A valid AtomAddress points to a chain, residue and atom in a model. Pointers to the Chain, Residue and Atom can be kept together in another small structure, called CRA:

```
>>> cra = st[0].find_cra(addr)
>>> cra
<gemmi.CRA A/ALA 2/N>
>>> cra.chain
<gemmi.Chain A with 21 res>
>>> cra.residue
<gemmi.Residue 2(ALA) with 5 atoms>
>>> cra.atom
<gemmi.Atom N at (-24.5, -13.9, 14.8)>
```

Now, as an exercise, we will delete and re-create a disulfide bond:

```
>>> # remove
>>> st.connections.pop(0)
<gemmi.Connection disulf1  A/CYS 4/SG - A/CYS 10/SG>
>>> # create
>>> con = gemmi.Connection()
>>> con
<gemmi.Connection   / ?/ - / ?/>
>>> con.name = 'new_disulf'
>>> con.type = gemmi.ConnectionType.Disulf
>>> con.asu = gemmi.Asu.Same
>>> chain_a = st[0]['A']
>>> res4 = chain_a['4']['CYS']
>>> res10 = chain_a['10']['CYS']
>>> con.partner1 = gemmi.AtomAddress(chain_a, res4, res4.sole_atom('SG'))
>>> con.partner2 = gemmi.AtomAddress(chain_a, res10, res10.sole_atom('SG'))
>>> st.connections.append(con)
>>> st.connections[-1]
<gemmi.Connection new_disulf  A/CYS 4/SG - A/CYS 10/SG>
```

1.4.20 Examples

Chain longer than cell

Is it possible for a single chain to exceed the size of the unit cell in one of the directions? How much longer can it be than the cell?

```
# This script looks for chains that exceed the size of the unit cell (by >20%)
# in one of the a, b, c directions.

import sys
import gemmi

def run(path):
    counter = 0
    st = gemmi.read_structure(path)
    if st.cell.is_crystal():
        st.add_entity_types()
        for chain in st[0]:
            polymer = chain.get_polymer()
            if polymer:
                low_bounds = [float('+inf')] * 3
                high_bounds = [float('-inf')] * 3
                for residue in polymer:
                    for atom in residue:
                        pos = st.cell.fractionalize(atom.pos)
                        for i in range(3):
                            if pos[i] < low_bounds[i]:
                                low_bounds[i] = pos[i]
                            if pos[i] > high_bounds[i]:
                                high_bounds[i] = pos[i]
                for i in range(3):
                    delta = high_bounds[i] - low_bounds[i]
                    if delta > 1.2:  # 120% of the unit cell size
                        counter += 1
                        code = st.info['_entry.id']
                        print('%s   chain:%s   delta%c = %.3f' %
                              (code, chain.name, ord('X') + i, delta))
    return counter

def main():
    for arg in sys.argv[1:]:
        for path in gemmi.CoorFileWalk(arg):
            run(path)

if __name__ == '__main__':
    main()
```

When run on the PDB database (on a local copy of either pdb or mmCIF files) this script prints too many lines to show here.

```
$ ./examples/long_geom.py $PDB_DIR/structures/divided/pdb/
105M   chain:A   deltaY = 1.225
208L   chain:A   deltaZ = 1.203
11BA   chain:A   deltaX = 1.227
11BA   chain:B   deltaX = 1.202
...
3NWH   chain:A   deltaX = 3.893
3NWH   chain:B   deltaX = 3.955
3NWH   chain:C   deltaX = 4.093
```

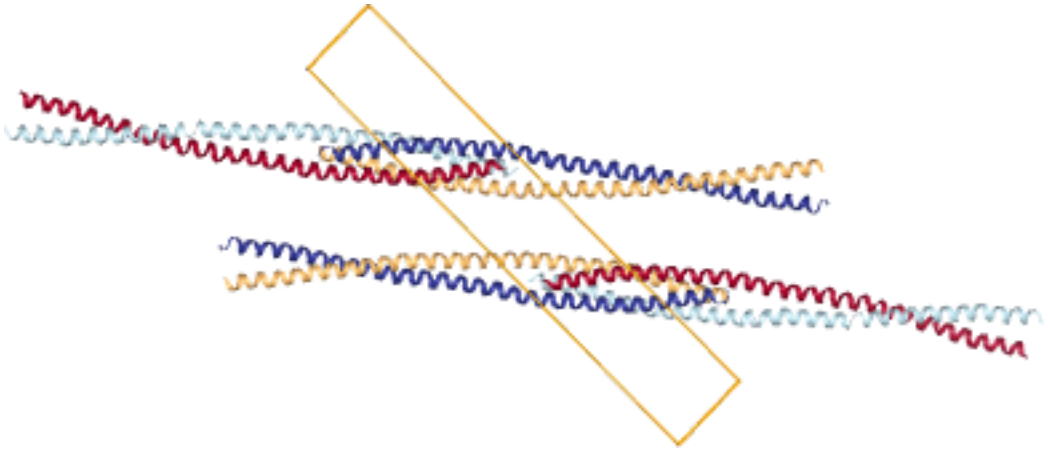
(continues on next page)

(continued from previous page)

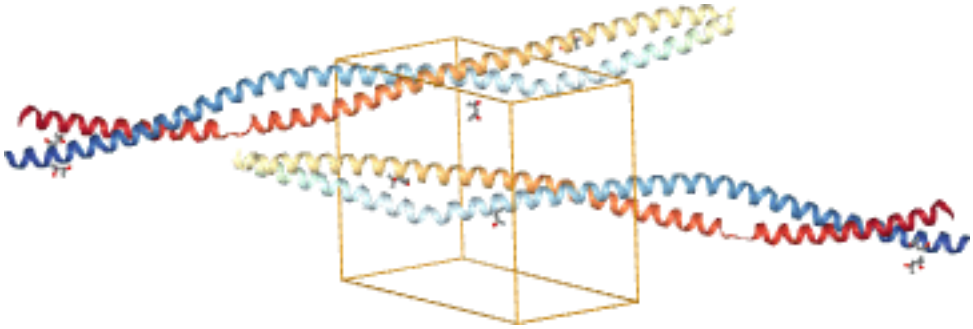
```
3NWH   chain:D   deltaX = 3.472
...
5XG2   chain:A   deltaX = 4.267
5XG2   chain:A   deltaZ = 1.467
...
```

As we see, a single chain may be even longer than four unit cells in one of the directions. How such chains look like?

For example, here is 3NWH – a homo-4-mer in P2 (4 x 2 chains per unit cell) – colored by chain id:



And here is 5XG2 – a monomer in P21 – with two copies of the rainbow-colored chain:



CCD subset

Since the whole Chemical Component Dictionary is large we may want to extract a subset of it that covers only residues in given structures.

```
# Make a list of residue names that we need.
mon_names = set()
for coordinate_file in COORDINATE_FILES:
    st = gemmi.read_structure(coordinate_file)
    mon_names.update(st[0].get_all_residue_names())

# Copy blocks corresponding to these residues to a new file.
sub = gemmi.cif.Document()
for block in gemmi.cif.read(CCD_PATH):
    if block.name in mon_names:
        sub.add_copied_block(block)
sub.write_file(OUTPUT_PATH)
```

For complete and ready-to-use script see `examples/sub_ccd.py`.

1.5 Structure analysis

1.5.1 Neighbor search

Fixed-radius near neighbor search is usually implemented using the cell lists method, also known as binning, bucketing or cell technique (or cubing – as it was called in an article from 1966). The method is simple. The unit cell (or the area where the molecules are located) is divided into small cells. The size of these cells depends on the search radius. Each cell stores the list of atoms in its area; these lists are used for fast lookup of atoms.

In Gemmi the cell technique is implemented in a class named `NeighborSearch`. The implementation works with both crystal and non-crystal system and:

- handles crystallographic symmetry (including non-standard settings with origin shift that are present in a couple hundreds of PDB entries),
- handles strict NCS (MTRIX record in the PDB format that is not "given"; in mmCIF it is the _struct_ncs_oper category),
- handles alternative locations (atoms from different conformers are not neighbors),
- can find neighbors any number of unit cells apart; surprisingly, molecules from different and not neighboring unit cells can be in contact, either because of the molecule shape (a single chain can be *longer then four unit cells*) or because of the non-optimal choice of symmetric images in the model (some PDB entries have even links between chains more than 10 unit cells away which cannot be expressed in the 1555 type of notation).

Note that while an atom can be bonded with its own symmetric image, it sometimes happens that an atom meant to be on a special position is slightly off, and its symmetric images represent the same atom (so we may have four nearby images each with occupancy 0.25). Such images will be returned by the NeighborSearch class as neighbors and need to be filtered out by the users.

The NeighborSearch constructor divides the unit cell into bins. For this it needs to know the the maximum radius that will be used in searches, as well as the unit cell. Since the system may be non-periodic, the constructor also takes the model as an argument – it is used to calculate the bounding box for the model if there is no unit cell. It is also stored and used if `populate()` is called. The C++ signature (in `gemmi/neighbor.hpp`) is:

```
NeighborSearch::NeighborSearch(Model& model, const UnitCell& cell, double max_radius)
```

Then the cell lists need to be populated with items either by calling:

```
void NeighborSearch::populate(bool include_h=true)
```

or by adding individual atoms:

```
void NeighborSearch::add_atom(const Atom& atom, int n_ch, int n_res, int n_atom)
```

where `n_ch` is the index of the chain in the model, `n_res` is the index of the residue in the chain, and `n_atom` is the index of the atom in the residue.

An example in Python:

```
>>> import gemmi
>>> st = gemmi.read_structure('../tests/1pfe.cif.gz')
>>> ns = gemmi.NeighborSearch(st[0], st.cell, 3).populate()
```

If we'd like to choose which atoms to add, for example to ignore hydrogens, we could use `add_atom()` instead of `populate()`:

```
>>> ns = gemmi.NeighborSearch(st[0], st.cell, 3)
>>> for n_ch, chain in enumerate(st[0]):
...     for n_res, res in enumerate(chain):
...         for n_atom, atom in enumerate(res):
...             if not atom.is_hydrogen():
...                 ns.add_atom(atom, n_ch, n_res, n_atom)
...
```

NeighborSearch has a couple of functions for searching. The first one takes atom as an argument:

```
std::vector<Mark*> NeighborSearch::find_neighbors(const Atom& atom, float min_dist,␣
↪float max_dist)
```

```
>>> ref_atom = st[0].sole_residue('A', gemmi.SeqId('3')).sole_atom('P')
>>> marks = ns.find_neighbors(ref_atom, min_dist=0.1, max_dist=3)
>>> len(marks)
6
```

`find_neighbors()` checks altloc of the atom and considers as potential neighbors only atoms from the same conformation. In particular, if altloc is empty all atoms are considered. Non-negative `min_dist` in the `find_neighbors()` call prevents the atom whose neighbors we search from being included in the results (the distance of the atom to itself is zero).

The second one takes position and altloc as explicit arguments:

```
std::vector<Mark*> NeighborSearch::find_atoms(const Position& pos, char altloc, float␣
↪radius)
```

```
>>> point = gemmi.Position(20, 20, 20)
>>> marks = ns.find_atoms(point, '\0', radius=3)
>>> len(marks)
7
```

Additionally, in C++ you may use a function that takes a callback as the last argument (usage examples are in the source code):

```
template<typename T>
void NeighborSearch::for_each(const Position& pos, char altloc, float radius, const T&
↪ func)
```

Cell-lists store `Marks`. When searching for neighbors you get references (in C++ – pointers) to these marks. `Mark` has a number of properties: `x`, `y`, `z`, `altloc`, `element`, `image_idx` (index of the symmetry operation that was used to generate this mark, 0 for identity), `chain_idx`, `residue_idx` and `atom_idx`.

```
>>> mark = marks[0]
>>> mark
<gemmi.NeighborSearch.Mark O of atom 0/7/3>
>>> mark.x, mark.y, mark.z
(19.659000396728516, 20.248884201049805, 17.645000457763672)
>>> mark.altloc
'\x00'
>>> mark.element
<gemmi.Element: O>
>>> mark.image_idx
```

(continues on next page)

(continued from previous page)

```
11
>>> mark.chain_idx, mark.residue_idx, mark.atom_idx
(0, 7, 3)
```

The references to the original model and to atoms are not stored. `Mark` has a method `to_cra()` that needs to be called with `Model` as an argument to get a triple of Chain, Residue and Atom:

```
CRA NeighborSearch::Mark::to_cra(Model& model) const
```

```
>>> cra = mark.to_cra(st[0])
>>> cra.chain
<gemmi.Chain A with 79 res>
>>> cra.residue
<gemmi.Residue 8(DC) with 19 atoms>
>>> cra.atom
<gemmi.Atom O5' at (-0.0, 13.9, -17.6)>
```

`Mark` also has a helper method `pos()` that returns `Position(x, y, z)`:

```
Position NeighborSearch::Mark::pos() const
```

```
>>> mark.pos()
<gemmi.Position(19.659, 20.2489, 17.645)>
```

Note that it can be the position of a symmetric image of the atom. In this example the "original" atom is in a different location:

```
>>> cra.atom.pos
<gemmi.Position(-0.028, 13.85, -17.645)>
```

1.5.2 Contact search

Contacts in a molecule or in a crystal can be found using the neighbor search described in the previous section. But to make it easier we have a dedicated class ContactSearch. It uses the neighbor search to find pairs of atoms close to each other and applies the filters described below.

When constructing ContactSearch we set the overall maximum search distance. This distance is stored as the `search_radius` property:

```
>>> cs = gemmi.ContactSearch(4.0)
>>> cs.search_radius
4.0
```

Additionally, we can set up per-element radii. This excludes pairs of atoms in a distance larger than the sum of their per-element radii. The radii are initialized as a linear function of the *covalent radius*: $r = a \times r_{\text{cov}} + b/2$.

```
>>> cs.setup_atomic_radii(1.0, 1.5)
```

Then each radius can be accessed and modified individually:

```
>>> cs.get_radius(gemmi.Element('Zr'))
2.5
>>> cs.set_radius(gemmi.Element('Hg'), 1.5)
```

Next, we have the `ignore` property that can take one of the following values:

- ContactSearch.Ignore.Nothing – no filtering here,
- ContactSearch.Ignore.SameResidue – ignore atom pairs from the same residue,
- ContactSearch.Ignore.AdjacentResidues – ignore atom pairs from the same or adjacent residues,
- ContactSearch.Ignore.SameChain – show only inter-chain contacts (including contacts between different symmetry images of one chain),
- ContactSearch.Ignore.SameAsu – show only contacts between different asymmetric units.

```
>>> cs.ignore = gemmi.ContactSearch.Ignore.AdjacentResidues
```

You can also ignore atoms that have occupancy below the specified threshold:

```
>>> cs.min_occupancy = 0.01
```

Sometimes, it is handy to get each atom pair twice (as A-B and B-A). In such case make the `twice` property true. By default, it is false:

```
>>> cs.twice
False
```

Next property deals with atoms at special positions (such as rotation axis). Such atoms can be slightly off the special position (because macromolecular refinement programs usually don't constrain coordinates), so we must ensure that an atom that should sit on the special position and its apparent symmetry image are not regarded a contact. We assume that if the distance between an atom and its image is small, it is not a real thing. For larger distances we assume it is a real contact with atom's symmetry mate. To tell apart the two cases we use a cut-off distance that can be modified:

```
>>> cs.special_pos_cutoff_sq = 0.5 ** 2  # setting cut-off to 0.5A
```

The contact search uses an instance of NeighborSearch.

```
>>> st = gemmi.read_structure('../tests/5cvz_final.pdb')
>>> st.setup_entities()
>>> ns = gemmi.NeighborSearch(st[0], st.cell, 5).populate()
```

If you'd like to ignore hydrogens from the model, call `ns.populate(include_h=False)`.

If you'd like to ignore waters, either remove waters from the Model (function `remove_waters()`) or ignore results that contain waters.

The actual contact search is done by:

```
>>> results = cs.find_contacts(ns)

>>> len(results)
49
>>> results[0]
<gemmi.ContactSearch.Result object at 0x...>
```

The ContactSearch.Result class has four properties:

```
>>> results[0].partner1
<gemmi.CRA A/SER 21/OG>
>>> results[0].partner2
<gemmi.CRA A/TYR 24/N>
>>> results[0].image_idx
```

(continues on next page)

(continued from previous page)

```
52
>>> results[0].dist
2.8613362312316895
```

The first two properties are *CRA*s for the involved atoms. The `image_idx` is an index of the symmetry image (both crystallographic symmetry and strict NCS count). Value 0 would mean that both atoms (`partner1` and `partner2`) are in the same unit. In this example the value can be high because it is a structure of icosahedral viral capsid with 240 identical units in the unit cell. The last property is the distance between atoms.

See also command-line program *gemmi-contact*.

Gemmi provides also an undocumented class LinkHunt which matches contacts to links definitions from *monomer library* and to connections (LINK, SSBOND) from the structure. If you would find it useful, contact the author.

1.5.3 Selections

For now, Gemmi supports only the selection syntax from MMDB, called CID (Coordinate ID). The syntax is described at the bottom of the pdbcur documentation.

The selections in Gemmi are not widely used yet and the API may evolve. The examples below demonstrates currently provided functions.

Example 1

Working with CID selections.

```
>>> st = gemmi.read_structure('../tests/1pfe.cif.gz')

>>> # select all Cl atoms
>>> sel = gemmi.parse_cid('[CL]')
>>> # get the first result as pointer to model and CRA (chain, residue, atom)
>>> sel.first(st)
(<gemmi.Model 1 with 2 chain(s)>, <gemmi.CRA A/CL 20/CL>)

>>> sel = gemmi.parse_cid('A/1-4/N9')
>>> sel.to_cid()
'//A/1.-4./N9'
>>> # iterate over hierarchy filtered by the selection
>>> for model in sel.models(st):
...     for chain in sel.chains(model):
...         print('-', chain.name)
...         for residue in sel.residues(chain):
...             print('   -', str(residue))
...             for atom in sel.atoms(residue):
...                 print('          -', atom.name)
...
- A
   - 1(DG)
          - N9
   - 2(DC)
   - 3(DG)
          - N9
   - 4(DT)
```

Example 2

Copy alpha-carbon atoms to a new structure (or a model).

```
>>> st = gemmi.read_structure('../tests/1orc.pdb')
>>> st[0].count_atom_sites()
559
>>> selection = gemmi.parse_cid('CA[C]')

>>> # create a new structure
>>> ca_st = selection.copy_structure_selection(st)
>>> ca_st[0].count_atom_sites()
64

>>> # create a new model
>>> ca_model = selection.copy_model_selection(st[0])
>>> ca_model.count_atom_sites()
64
```

Example 3

Select residues in the radius of 8Å from a selected point.

```
>>> selected_point = gemmi.Position(20, 40, 30)
>>> ns = gemmi.NeighborSearch(st[0], st.cell, 8.0).populate()
>>> for mark in ns.find_atoms(selected_point):
...     mark.to_cra(st[0]).residue.flag = 's'
>>> selection = gemmi.Selection().set_residue_flags('s')
>>> selection.copy_model_selection(st[0]).count_atom_sites()
121
```

Note: NeighborSearch searches for atoms in all symmetry images. This is why it takes UnitCell as a parameter. To search only in atoms directly listed in the file pass empty cell (`gemmi.UnitCell()`).

Example 3a

Select atoms in the radius of 8Å from a selected point.

```
>>> # selected_point and ns are reused from the previous example
>>> for mark in ns.find_atoms(selected_point):
...     mark.to_cra(st[0]).atom.flag = 's'
>>> selection = gemmi.Selection().set_atom_flags('s')
>>> selection.copy_model_selection(st[0]).count_atom_sites()
59
```

1.5.4 Graph analysis

The graph algorithms in Gemmi are limited to finding the shortest path between atoms (bonds = graph edges). This part of the library is not documented yet.

The rest of this section shows how to use Gemmi together with external graph analysis libraries to analyse the similarity of chemical molecules. To do this, first we set up a graph corresponding to the molecule.

Here we show how it can be done in the Boost Graph Library.

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <gemmi/cif.hpp>             // for cif::read_file
#include <gemmi/chemcomp.hpp>        // for ChemComp, make_chemcomp_from_block

struct AtomVertex {
```

(continues on next page)

(continued from previous page)

```
  gemmi::Element el = gemmi::El::X;
  std::string name;
};

struct BondEdge {
  gemmi::BondType type;
};

using Graph = boost::adjacency_list<boost::vecS, boost::vecS,
                                    boost::undirectedS,
                                    AtomVertex, BondEdge>;

Graph make_graph(const gemmi::ChemComp& cc) {
  Graph g(cc.atoms.size());
  for (size_t i = 0; i != cc.atoms.size(); ++i) {
    g[i].el = cc.atoms[i].el;
    g[i].name = cc.atoms[i].id;
  }
  for (const gemmi::Restraints::Bond& bond : cc.rt.bonds) {
    int n1 = cc.get_atom_index(bond.id1.atom);
    int n2 = cc.get_atom_index(bond.id2.atom);
    boost::add_edge(n1, n2, BondEdge{bond.type}, g);
  }
  return g;
}
```

And here we use NetworkX in Python:

```
>>> import networkx

>>> G = networkx.Graph()
>>> block = gemmi.cif.read('../tests/SO3.cif')[-1]
>>> so3 = gemmi.make_chemcomp_from_block(block)
>>> for atom in so3.atoms:
...     G.add_node(atom.id, Z=atom.el.atomic_number)
...
>>> for bond in so3.rt.bonds:
...     G.add_edge(bond.id1.atom, bond.id2.atom)  # ignoring bond type
...
```

To show a quick example, let us count automorphisms of SO3:

```
>>> import networkx.algorithms.isomorphism as iso
>>> GM = iso.GraphMatcher(G, G, node_match=iso.categorical_node_match('Z', 0))
>>> # expecting 3! automorphisms (permutations of the three oxygens)
>>> sum(1 for _ in GM.isomorphisms_iter())
6
```

With a bit more of code we could perform a real cheminformatics task.

Graph isomorphism

In this example we use Python NetworkX to compare molecules from the Refmac monomer library with Chemical Component Dictionary (CCD) from PDB. The same could be done with other graph analysis libraries, such as Boost Graph Library, igraph, etc.

The program below takes compares specified monomer cif files with corresponding CCD entries. Hydrogens and bond types are ignored. It takes less than half a minute to go through the 25,000 monomer files distributed with CCP4 (as of Oct 2018), so we do not try to optimize the program.

```
# Compares graphs of molecules from cif files (Refmac dictionary or similar)
# with CCD entries.

import sys
import networkx
from networkx.algorithms import isomorphism
import gemmi

CCD_PATH = 'components.cif.gz'

def graph_from_chemcomp(cc):
    G = networkx.Graph()
    for atom in cc.atoms:
        G.add_node(atom.id, Z=atom.el.atomic_number)
    for bond in cc.rt.bonds:
        G.add_edge(bond.id1.atom, bond.id2.atom)
    return G

def compare(cc1, cc2):
    s1 = {a.id for a in cc1.atoms}
    s2 = {a.id for a in cc2.atoms}
    b1 = {b.lexicographic_str() for b in cc1.rt.bonds}
    b2 = {b.lexicographic_str() for b in cc2.rt.bonds}
    if s1 == s2 and b1 == b2:
        #print(cc1.name, "the same")
        return
    G1 = graph_from_chemcomp(cc1)
    G2 = graph_from_chemcomp(cc2)
    node_match = isomorphism.categorical_node_match('Z', 0)
    GM = isomorphism.GraphMatcher(G1, G2, node_match=node_match)
    if GM.is_isomorphic():
        print(cc1.name, 'is isomorphic')
        # we could use GM.match(), but here we try to find the shortest diff
        short_diff = None
        for n, mapping in enumerate(GM.isomorphisms_iter()):
            diff = {k: v for k, v in mapping.items() if k != v}
            if short_diff is None or len(diff) < len(short_diff):
                short_diff = diff
            if n == 10000:  # don't spend too much here
                print(' (it may not be the simplest isomorphism)')
                break
        for id1, id2 in short_diff.items():
            print('\t', id1, '->', id2)
    else:
        print(cc1.name, 'differs')
        if s2 - s1:
            print('\tmissing:', ' '.join(s2 - s1))
        if s1 - s2:
            print('\textra:  ', ' '.join(s1 - s2))

def main():
    ccd = gemmi.cif.read(CCD_PATH)
    absent = 0
    for f in sys.argv[1:]:
```

(continues on next page)

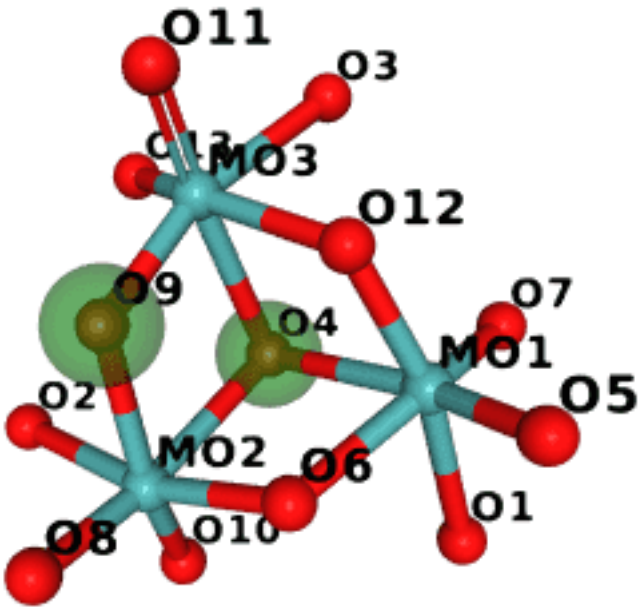
(continued from previous page)

```
        block = gemmi.cif.read(f)[-1]
        cc1 = gemmi.make_chemcomp_from_block(block)
        try:
            block2 = ccd[cc1.name]
        except KeyError:
            absent += 1
            #print(cc1.name, 'not in CCD')
            continue
        cc2 = gemmi.make_chemcomp_from_block(block2)
        cc1.remove_hydrogens()
        cc2.remove_hydrogens()
        compare(cc1, cc2)
    if absent != 0:
        print(absent, 'of', len(sys.argv) - 1, 'monomers not found in CCD')

main()
```

If we run it on monomers that start with M we get:

```
$ examples/ccd_gi.py $CLIBD_MON/m/*.cif
M10 is isomorphic
       O9 -> O4
       O4 -> O9
MK8 is isomorphic
       O2 -> OXT
MMR differs
      missing: O12 O4
2 of 821 monomers not found in CCD
```

So in M10 the two atoms marked green are swapped:



(The image was generated in NGL and compressed with Compress-Or-Die.)

Substructure matching

Now a little script to illustrate subgraph isomorphism. The script takes a (three-letter-)code of a molecule that is to be used as a pattern and finds CCD entries that contain such a a substructure. As in the previous example, hydrogens and bond types are ignored.

```
# List CCD entries that contain the specified entry as a substructure.
# Ignoring hydrogens and bond types.

import sys
import networkx
from networkx.algorithms import isomorphism
import gemmi

CCD_PATH = 'components.cif.gz'

def graph_from_block(block):
    cc = gemmi.make_chemcomp_from_block(block)
    cc.remove_hydrogens()
    G = networkx.Graph()
    for atom in cc.atoms:
        G.add_node(atom.id, Z=atom.el.atomic_number)
    for bond in cc.rt.bonds:
        G.add_edge(bond.id1.atom, bond.id2.atom)
    return G

def main():
    assert len(sys.argv) == 2, "Usage: ccd_subgraph.py three-letter-code"
    ccd = gemmi.cif.read(CCD_PATH)
    entry = sys.argv[1]
    pattern = graph_from_block(ccd[entry])
    pattern_nodes = networkx.number_of_nodes(pattern)
    pattern_edges = networkx.number_of_edges(pattern)
    node_match = isomorphism.categorical_node_match('Z', 0)
    for block in ccd:
        G = graph_from_block(block)
        GM = isomorphism.GraphMatcher(G, pattern, node_match=node_match)
        if GM.subgraph_is_isomorphic():
            print(block.name, '\t +%d nodes, +%d edges' % (
                networkx.number_of_nodes(G) - pattern_nodes,
                networkx.number_of_edges(G) - pattern_edges))

main()
```

Let us check what entries have HEM as a substructure:

```
$ examples/ccd_subgraph.py HEM
1FH     +6 nodes, +7 edges
2FH     +6 nodes, +7 edges
4HE     +7 nodes, +8 edges
522     +2 nodes, +2 edges
6CO     +6 nodes, +7 edges
6CQ     +7 nodes, +8 edges
89R     +3 nodes, +3 edges
CLN     +1 nodes, +2 edges
DDH     +2 nodes, +2 edges
FEC     +6 nodes, +6 edges
HAS     +22 nodes, +22 edges
```

(continues on next page)

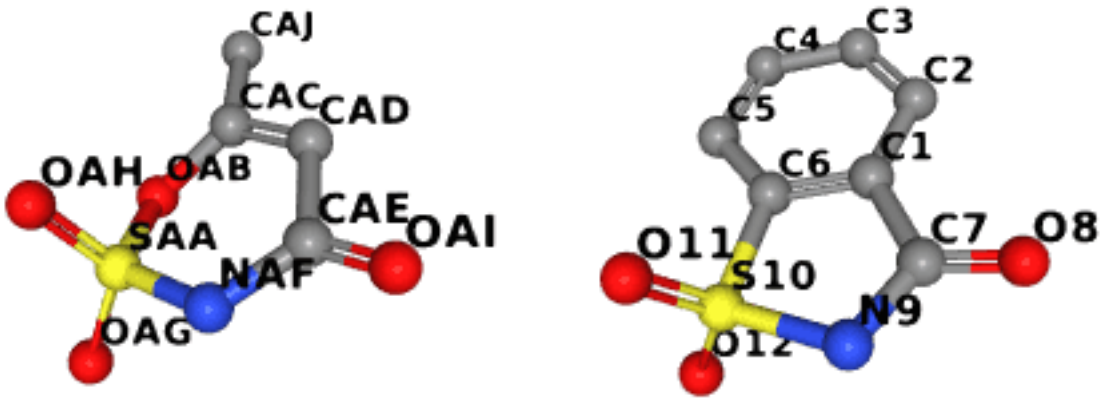
(continued from previous page)

```
HCO    +1 nodes, +1 edges
HDM    +2 nodes, +2 edges
HEA    +17 nodes, +17 edges
HEB    +0 nodes, +0 edges
HEC    +0 nodes, +0 edges
HEM    +0 nodes, +0 edges
HEO    +16 nodes, +16 edges
HEV    +2 nodes, +2 edges
HP5    +2 nodes, +2 edges
ISW    +0 nodes, +0 edges
MH0    +0 nodes, +0 edges
MHM    +0 nodes, +0 edges
N7H    +3 nodes, +3 edges
NTE    +3 nodes, +3 edges
OBV    +14 nodes, +14 edges
SRM    +20 nodes, +20 edges
UFE    +18 nodes, +18 edges
```

Maximum common subgraph

In this example we use McGregor's algorithm implemented in the Boost Graph Library to find maximum common induced subgraph. We call the MCS searching function with option `only_connected_subgraphs=true`, which has obvious meaning and can be changed if needed.

To illustrate this example, we compare ligands AUD and LSA:



The whole code is in `examples/with_bgl.cpp`. The same file has also examples of using the BGL implementation of VF2 to check graph and subgraph isomorphisms.

```
// We ignore hydrogens here.
// Example output:
//   $ time with_bgl -c monomers/a/AUD.cif monomers/l/LSA.cif
//   Searching largest subgraphs of AUD and LSA (10 and 12 vertices)...
//   Maximum connected common subgraph has 7 vertices:
//     SAA -> S10
//     OAG -> O12
//     OAH -> O11
//     NAF -> N9
//     CAE -> C7
//     OAI -> O8
```

(continues on next page)

(continued from previous page)

```
//     CAD -> C1
//   Maximum connected common subgraph has 7 vertices:
//     SAA -> S10
//     OAG -> O11
//     OAH -> O12
//     NAF -> N9
//     CAE -> C7
//     OAI -> O8
//     CAD -> C1
//
//   real        0m0.012s
//   user        0m0.008s
//   sys         0m0.004s

struct PrintCommonSubgraphCallback {
  Graph g1, g2;
  template <typename CorrespondenceMap1To2, typename CorrespondenceMap2To1>
  bool operator()(CorrespondenceMap1To2 map1, CorrespondenceMap2To1,
                  typename GraphTraits::vertices_size_type subgraph_size) {
    std::cout << "Maximum connected common subgraph has " << subgraph_size
              << " vertices:\n";
    for (auto vp = boost::vertices(g1); vp.first != vp.second; ++vp.first) {
      Vertex v1 = *vp.first;
      Vertex v2 = boost::get(map1, v1);
      if (v2 != GraphTraits::null_vertex())
        std::cout << "  " << g1[v1].name << " -> " << g2[v2].name << std::endl;
    }
    return true;
  }
};

void find_common_subgraph(const char* cif1, const char* cif2) {
  gemmi::ChemComp cc1 = make_chemcomp(cif1).remove_hydrogens();
  Graph graph1 = make_graph(cc1);
  gemmi::ChemComp cc2 = make_chemcomp(cif2).remove_hydrogens();
  Graph graph2 = make_graph(cc2);
  std::cout << "Searching largest subgraphs of " << cc1.name << " and "
      << cc2.name << " (" << cc1.atoms.size() << " and " << cc2.atoms.size()
      << " vertices)..." << std::endl;
  mcgregor_common_subgraphs_maximum_unique(
      graph1, graph2,
      get(boost::vertex_index, graph1), get(boost::vertex_index, graph2),
      [&](Edge a, Edge b) { return graph1[a].type == graph2[b].type; },
      [&](Vertex a, Vertex b) { return graph1[a].el == graph2[b].el; },
      /*only_connected_subgraphs=*/ true,
      PrintCommonSubgraphCallback{graph1, graph2});
}
```

1.5.5 Torsion Angles

This section presents functions dedicated to calculation of the dihedral angles ϕ (phi), ψ (psi) and ω (omega) of the protein backbone. These functions are built upon the more general `calculate_dihedral` function, introduced in *the section about coordinates*, which takes four points in the space as arguments.

`calculate_omega()` calculates the ω angle, which is usually around 180°:

```
>>> from math import degrees
>>> chain = gemmi.read_structure('../tests/5cvz_final.pdb')[0]['A']
>>> degrees(gemmi.calculate_omega(chain[0], chain[1]))
159.90922150065668
>>> for res in chain[:5]:
...     next_res = chain.next_residue(res)
...     if next_res:
...         omega = gemmi.calculate_omega(res, next_res)
...         print(res.name, degrees(omega))
...
ALA 159.90922150065668
ALA -165.26874513591105
ALA -165.85686681169656
THR -172.99968385093513
SER 176.74223937657646
```

The ϕ and ψ angles are often used together, so they are calculated in one function `calculate_phi_psi()`:

```
>>> for res in chain[:5]:
...     prev_res = chain.previous_residue(res)
...     next_res = chain.next_residue(res)
...     phi, psi = gemmi.calculate_phi_psi(prev_res, res, next_res)
...     print('%s %8.2f %8.2f' % (res.name, degrees(phi), degrees(psi)))
...
ALA      nan   106.94
ALA  -116.64    84.57
ALA   -45.57   127.40
THR   -62.01   147.45
SER   -92.85   161.53
```

In C++ these functions can be found in `gemmi/calculate.hpp`.

The torsion angles ϕ and ψ can be visualized on the Ramachandran plot. Let us plot angles from all PDB entries with the resolution higher than 1.5A. Usually, glycine, proline and the residue preceding proline (pre-proline) are plotted separately. Here, we will exclude pre-proline and make separate plot for each amino acid. So first, we calculate angles and save ϕ,ψ pairs in a set of files – one file per residue.

```
import sys
from math import degrees
import gemmi

ramas = {aa: [] for aa in [
    'LEU', 'ALA', 'GLY', 'VAL', 'GLU', 'SER', 'LYS', 'ASP', 'THR', 'ILE',
    'ARG', 'PRO', 'ASN', 'PHE', 'GLN', 'TYR', 'HIS', 'MET', 'CYS', 'TRP']}

for path in gemmi.CoorFileWalk(sys.argv[1]):
    st = gemmi.read_structure(path)
    if 0.1 < st.resolution < 1.5:
        model = st[0]
        for chain in model:
            for res in chain.get_polymer():
                # previous_residue() and next_residue() return previous/next
                # residue only if the residues are bonded. Otherwise -- None.
                prev_res = chain.previous_residue(res)
                next_res = chain.next_residue(res)
                if prev_res and next_res and next_res.name != 'PRO':
                    v = gemmi.calculate_phi_psi(prev_res, res, next_res)
```

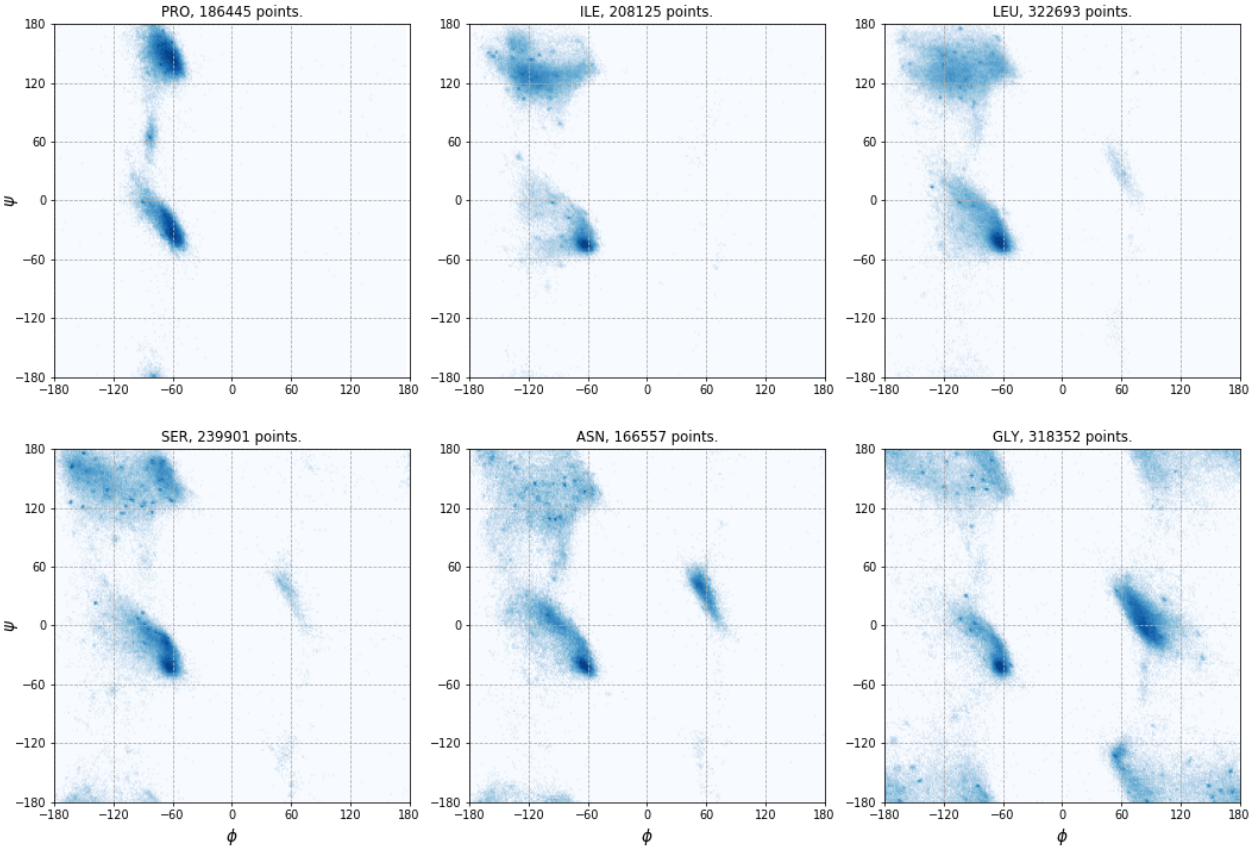
(continues on next page)

(continued from previous page)

```
                    try:
                        ramas[res.name].append(v)
                    except KeyError:
                        pass

# Write data to files
for aa, data in ramas.items():
    with open('ramas/' + aa + '.tsv', 'w') as f:
        for phi, psi in data:
            f.write('%.4f\t%.4f\n' % (degrees(phi), degrees(psi)))
```

The script above works with coordinate files in any of the formats supported by gemmi (PDB, mmCIF, mmJSON). As of 2019, processing a *local copy of the PDB archive* in the PDB format takes about 20 minutes.

In the second step we plot the data points with matplotlib. We use a script that can be found in `examples/rama_gather.py`. Six of the resulting plots are shown here (click to enlarge):



1.5.6 Local copy of the PDB archive

Some examples in this documentation work on a local copy of the Protein Data Bank archive. This subsection is actually a footnote describing our setup.

Like in BioJava, we assume that the `$PDB_DIR` environment variable points to a directory that contains `structures/divided/mmCIF` – the same arrangement as on the PDB's FTP server.

```
$ cd $PDB_DIR
$ du -sh structures/*/*  # as of Jun 2017
34G    structures/divided/mmCIF
25G    structures/divided/pdb
101G   structures/divided/structure_factors
2.6G   structures/obsolete/mmCIF
```

A traditional way to keep an up-to-date local archive is to rsync it once a week:

```
#!/bin/sh -x
set -u  # PDB_DIR must be defined
rsync_subdir() {
  mkdir -p "$PDB_DIR/$1"
  # Using PDBe (UK) here, can be replaced with RCSB (USA) or PDBj (Japan),
  # see https://www.wwpdb.org/download/downloads
  rsync -rlpt -v -z --delete \
      rsync.ebi.ac.uk::pub/databases/pdb/data/$1/ "$PDB_DIR/$1/"
}
rsync_subdir structures/divided/mmCIF
#rsync_subdir structures/obsolete/mmCIF
#rsync_subdir structures/divided/pdb
#rsync_subdir structures/divided/structure_factors
```

1.6 Grids and maps

1.6.1 Volumetric grid

Macromolecular models are often accompanied by 3D data on an evenly spaced, rectangular grid. The data may represent electron density, a mask of the protein area, or any other scalar data.

In Gemmi such a data is stored in a class called Grid. Actually, it is a set of classes for storing different types of data: floating point numbers, integers or boolean masks. These classes also store:

- unit cell dimensions (so the grid nodes can be assigned atomic coordinates),
- and crystallographic symmetry (that determines which points on the grid are equivalent under the symmetry).

If the symmetry is not set (or is set to P1) then we effectively have a box with periodic boundary conditions.

C++

The `gemmi/grid.hpp` header defines:

```
template<typename T=float> struct Grid;
```

which stores dimensions and data:

```
int nu, nv, nw;
std::vector<T> data;
```

The data point can be accessed with:

```
T Grid<T>::get_value(int u, int v, int w) const
void Grid<T>::set_value(int u, int v, int w, T x)
```

The unit cell and symmetry:

```
UnitCell unit_cell;
const SpaceGroup* spacegroup;
```

can be accessed directly, except that `unit_cell` should be set using `Grid<T>::set_unit_cell()`.

Unit cell parameters enable conversion between coordinates and grid points. To get an interpolated value at any position (in either fractional or orthogonal coordinates), use:

```
T Grid<T>::interpolate_value(const Fractional& fctr) const
T Grid<T>::interpolate_value(const Position& ctr) const
```

We also have a few more specialized functions. For example, a member function used primarily for masking area around atoms:

> void Grid<T>::set_points_around(const Position& ctr, double radius, T value)

To make it more efficient, the function above does not consider symmetry. At the end, we should call one of the *symmetrizing* functions. In this case, if two symmetry-related grid point have values 0 and 1 we want to set both to 1. It can be done by calling:

```
void Grid<T>::symmetrize_max()
```

This illustrates how the Grid is meant to be used. For more information consult the source code or contact the author.

Python

Let us create a new grid:

```
>>> import gemmi
>>>
>>> grid = gemmi.FloatGrid(12, 12, 12)
>>> grid.set_value(1, 1, 1, 7.0)
>>> grid.get_value(1, 1, 1)
7.0
>>> # we can test wrapping of indices (a.k.a. periodic boundary conditions)
>>> grid.get_value(-11, 13, 25)
7.0
```

The main advantage of Grid over generic 3D arrays is that it understands crystallographic symmetry.

```
>>> grid.spacegroup = gemmi.find_spacegroup_by_name('P2')
>>> grid.set_value(0, 0, 0, 0.125)  # a special position
>>> grid.sum()  # for now only two points: 7.0 + 0.125
7.125
>>> grid.symmetrize_max()  # applying symmetry
>>> grid.sum()  # one point got duplicated, the other is on rotation axis
14.125
>>> for point in grid:
...   if point.value != 0.: print(point)
<gemmi.FloatGridPoint (0, 0, 0) -> 0.125>
<gemmi.FloatGridPoint (1, 1, 1) -> 7>
<gemmi.FloatGridPoint (11, 1, 11) -> 7>
```

The point that you get when iterating over grid has four properties:

```
>>> grid.get_point(0, 0, 0)
<gemmi.FloatGridPoint (0, 0, 0) -> 0.125>
>>> _.u, _.v, _.w, _.value
(0, 0, 0, 0.125)
```

The point can also be converted to index and to fractional and orthoghonal c oordinates, as will be demonstrated later.

The data can be also acesssed through the buffer protocol. It means that you can use it as a NumPy array (Fortran-style contiguous) without copying the data:

```
>>> import numpy
>>> array = numpy.array(grid, copy=False)
>>> array.dtype
dtype('float32')
>>> array.shape
(12, 12, 12)
>>> numpy.argwhere(array == 7.0)
array([[ 1,  1,  1],
       [11,  1, 11]])
```

(It does not make gemmi dependent on NumPy – gemmi talks with NumPy through the buffer protocol, and it can talk with any other Python library that supports this protocol.)

We may be interested only in selected part of the map. For this, we have MaskedGrid that combines two Grid objects, using one of them as a mask for the other.

When an element of the mask is 0 (false), the corresponding element of the other grid is unmasked and is to be used. The same convention is used in numpy MaskedArray.

The primary use for MaskedGrid is working with asymmetric unit (asu) only:

```
>>> asu = grid.asu()
>>> asu
<gemmi.MaskedFloatGrid object at 0x...>
>>> asu.grid is grid
True
>>> asu.mask
<gemmi.Int8Grid(12, 12, 12)>
>>> sum(point.value for point in asu)
7.125
>>> for point in asu:
...   if point.value != 0: print(point)
<gemmi.FloatGridPoint (0, 0, 0) -> 0.125>
<gemmi.FloatGridPoint (1, 1, 1) -> 7>
```

To convert a point to index or to fractional coordinates use:

```
>>> point = grid.get_point(6, 6, 6)
>>> grid.point_to_index(point)
942
>>> grid.point_to_fractional(point)
<gemmi.Fractional(0.5, 0.5, 0.5)>
```

In addition to the symmetry, Grid may also have associated unit cell.

```
>>> grid.set_unit_cell(gemmi.UnitCell(45, 45, 45, 90, 82.5, 90))
>>> grid.unit_cell
<gemmi.UnitCell(45, 45, 45, 90, 82.5, 90)>
```

This allows us to translate location on the grid to position in Angstroms:

```
>>> grid.point_to_position(point)
<gemmi.Position(25.4368, 22.5, 22.3075)>
```

And the other way around. We can translate position in Angstroms to the location in grid, and get an interpolated value (with trilinear interpolation) at any point:

```
>>> grid.interpolate_value(gemmi.Position(2, 3, 4))
2.0333263874053955
```

This function can also take fractional position:

```
>>> grid.interpolate_value(gemmi.Fractional(1/24, 1/24, 1/24))
0.890625
```

If you need to interpolate values at so many points on a regular 3D grid that calling `interpolate_value()` for each point would be too slow, use `interpolate_values()` (with s at the end) instead. It takes a 3D numpy array and a *Transform* that relates indices of the array to positions in the grid, and fills the array with the interpolated values:

```
>>> # first we create a numpy array of the same type as the grid
>>> arr = numpy.zeros([32, 32, 32], dtype=numpy.float32)
>>> # then we setup a transformation (array indices) -> (position [A]).
>>> tr = gemmi.Transform()
>>> tr.mat.fromlist([[0.1, 0, 0], [0, 0.1, 0], [0, 0, 0.1]])
>>> tr.vec.fromlist([1, 2, 3])
>>> grid.interpolate_values(arr, tr)
>>> arr[10, 10, 10]  # -> corresponds to Position(2, 3, 4)
2.0333264
```

If you would like to set grid points near a specified position use the `set_points_around()` function:

```
>>> grid.set_points_around(gemmi.Position(25, 25, 25), radius=3, value=10)
>>> numpy.argwhere(array == 10)
array([[6, 6, 7],
       [6, 7, 7]])
>>> # now the data does not obey symmetry, we should call symmetrize*()
```

To set all point values, do:

```
>>> grid.fill(1.23)
```

1.6.2 MRC/CCP4 maps

We support one file format for storing the grid data on disk: MRC/CCP4 map. The content of the map file is stored in a class that contains both the Grid class and all the meta-data from the CCP4 file header.

The CCP4 format has a few different modes that correspond to different data types. Gemmi supports:

- mode 0 – which correspond to the C++ type int8_t,
- mode 1 – corresponds to int16_t,
- mode 2 – float,
- and mode 6 – uint16_t.

CCP4 programs use mode 2 (float) for the electron density, and mode 0 (int8_t) for masks. Mask is 0/1 data that marks part of the volume (e.g. the solvent region). Other modes are not used in crystallography, but may be used for CryoEM data.

The CCP4 format is quite flexible. The data is stored as sections, rows and columns that correspond to a permutation of the X, Y and Z axes as defined in the file header. The file can contain only a part of the asymmetric unit, or more than an asymmetric unit (i.e. redundant data). There are two typical approaches to generate a crystallographic map:

- old-school way: a map covering a molecule with some margin around it is produced using CCP4 utilities such as `fft` and `mapmask`,
- or a map is made for the asymmetric unit (asu), and the program that reads the map is supposed to expand the symmetry. This approach is used by the CCP4 clipper library and by programs that use this library, such as `cfft` and Coot.

The latter approach generates map for exactly one asu, if possible. It is not possible if the shape of the asu in fractional coordinates is not rectangular, and in such case we must have some redundancy. On average, the maps generated for asu are significantly smaller, as compared in the UglyMol wiki.

Nowadays, the CCP4 format is rarely used in crystallography. Almost all programs read the reflection data and calculate maps on the fly.

C++

Reading

To read and write CCP4 maps you need:

```
#include <gemmi/ccp4.hpp>
```

We normally use float type when reading a map file:

```
gemmi::Ccp4<float> map;
map.read_ccp4_map("my_map.ccp4");
```

and int8_t when reading a mask (mask typically has only values 0 and 1, but in principle it can have values from -127 to 128):

```
gemmi::Ccp4<int8_t> mask;
mask.read_ccp4_map("my_mask.ccp4");
```

If the grid data type does not match the file data type, the library will attempt to convert the data when reading.

Header

The CCP4 map header is organised as 56 words followed by space for ten 80-character text labels. The member functions that access the data from the map header use the word number (as in the format description) as a location in the header:

```
int32_t header_i32(int w) const;
float header_float(int w) const;
// ccp4 map header has mostly 80-byte strings
std::string header_str(int w, size_t len=80) const;

void set_header_i32(int w, int32_t value);
```

(continues on next page)

(continued from previous page)

```
void set_header_float(int w, float value);
void set_header_str(int w, const std::string& str);
```

For example:

```
int mode = map.header_i32(4);
float x = map.header_float(11);
```

setup()

`read_ccp4_map()` stores the data as it is written in the file. In many situation, it is convenient to have the data expanded to the whole unit cell, with axes in a specific order (X, Y, Z is the most conventional one). For this we have a function:

```
map.setup(GridSetup::Full, NAN);  // unknown values are set to NAN
```

This call is required to make grid functions work correctly with the unit cell parameters.

Writing

To write a map to a file:

```
// the file header needs to be prepared/updated with an explicit call
int mode = 2; // ccp4 file mode: 2 for floating-point data, 0 for masks
bool update_stats = true; // update min/max/mean/rms values in the header
map.update_ccp4_header(mode, update_stats);

map.write_ccp4_map(filename);
```

Python

The Python API is similar.

```
>>> m = gemmi.read_ccp4_map('../tests/5i55_tiny.ccp4')
>>> m
<gemmi.Ccp4Map with grid (8, 6, 10) in SG #4>
>>> m.grid  # tiny grid as it is a toy example
<gemmi.FloatGrid(8, 6, 10)>
>>> m.grid.spacegroup
<gemmi.SpaceGroup("P 1 21 1")>
>>> m.grid.unit_cell
<gemmi.UnitCell(29.45, 10.5, 29.7, 90, 111.975, 90)>
>>> m.setup()
>>> m.grid
<gemmi.FloatGrid(60, 24, 60)>
>>> m.write_ccp4_map('out.ccp4')
```

For the low-level access to header one can use the same getters and setters as in the C++ version.

```
>>> m.header_float(20), m.header_float(21)  # dmin, dmax
(-0.5310382843017578, 2.3988280296325684)
```

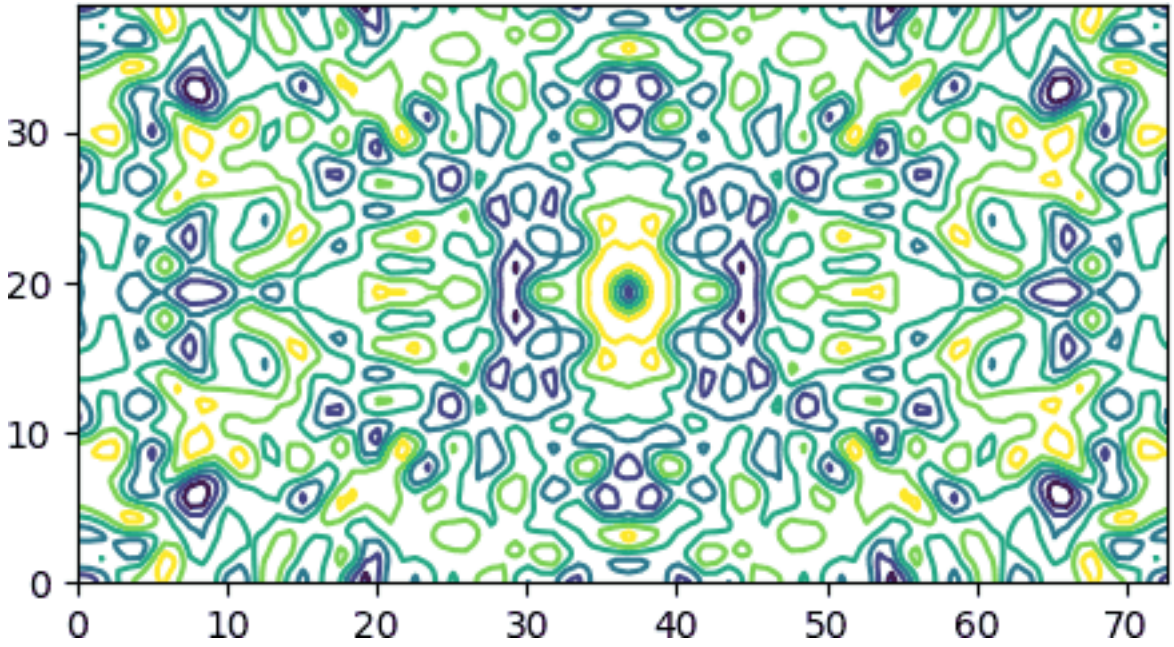
(continues on next page)

(continued from previous page)

```
>>> m.header_i32(28)
0
>>> m.set_header_i32(28, 20140)
>>> m.header_str(57, 80).strip()
'Created by MAPMAN V. 080625/7.8.5 at Wed Jan 3 12:57:38 2018 for A. Nonymous'
```

Let us end with a short code that draws a contour plot similar to mapslicer plots (see Fig. 3 in this CCP4 paper if you wonder what is mapslicer). To keep the example short we assume that the lattice vectors are orthogonal.

```
import numpy
from matplotlib import pyplot
import gemmi

# toxd_aupatt.map is generated by $CCP4/examples/unix/runnable/patterson
ccp4 = gemmi.read_ccp4_map('/tmp/wojdyr/toxd_aupatt.map')
ccp4.setup()
arr = numpy.array(ccp4.grid, copy=False)
x = numpy.linspace(0, ccp4.grid.unit_cell.a, num=arr.shape[0], endpoint=False)
y = numpy.linspace(0, ccp4.grid.unit_cell.b, num=arr.shape[1], endpoint=False)
X, Y = numpy.meshgrid(x, y, indexing='ij')
pyplot.contour(X, Y, arr[:,:,40])
pyplot.gca().set_aspect('equal', adjustable='box')
pyplot.show()
```



1.7 Reciprocal space

This section is primarily for people working with crystallographic data.

Gemmi supports three reflection file formats:

- MTZ files – the most popular format in macromolecular crystallography,
- structure factor mmCIF files – used for data archiving in the Protein Data Bank,

- and small molecule structure factor CIF files (usually with extension hkl).

Reflection files store Miller indices (*hkl*) with various associated numbers. Initially, the numbers represent observations derived from diffraction images (reflection intensities and error estimates). Then other quantities derived from these are added, as well as quantities derived from a macromolecular model that is built to explain the experimental data.

Even electron density maps, which are used in the real space, are nowadays stored mostly as map coefficients in reflection files. Molecular viewers can Fourier-transform them on the fly and show in the real space. Therefore, switching between the real and reciprocal space is also included in Gemmi, as well as calculation of structure factors from the model.

1.7.1 MTZ format

MTZ format has textual headers and a binary data table, where all numbers are stored in a 32-bit floating point format. The headers, as well as data, are stored in class `Mtz`. We have two types of MTZ files:

- merged – single record per (*hkl*) reflection
- unmerged – multi-record

In merged files the columns of data are grouped hierarchically (Project -> Crystal -> Dataset -> Column). Normally, the column name is all that is needed; the hierarchy can be ignored. In Gemmi, the hierarchy is flattened: we have a list of columns and a list of datasets. Each columns is associated with one dataset, and each dataset has properties `dataset_name`, `project_name` and `crystal_name`, which is enough to reconstruct the tree-like hierarchy if needed.

In unmerged files it is records, not columns, that are the associated with datasets. The BATCH column links records to batches (~ diffraction images). The batch header, in turn, stores dataset ID.

Reading

In C++, the MTZ file can be read using either stand-alone functions:

```
Mtz read_mtz_file(const std::string& path)
template<typename Input> Mtz read_mtz(Input&& input, bool with_data)
```

or member functions of the Mtz class, when more control over the reading process is needed.

In Python, we have a single function for reading MTZ files:

```
>>> import gemmi
>>> mtz = gemmi.read_mtz_file('../tests/5e5z.mtz')
```

The Mtz class has a number of properties read from the MTZ header (they are the same in C++ and Python):

```
>>> mtz.cell         # from MTZ record CELL
<gemmi.UnitCell(9.643, 9.609, 19.029, 90, 101.224, 90)>
>>> mtz.spacegroup  # from SYMINF
<gemmi.SpaceGroup("P 1 21 1")>
>>> mtz.title        # from TITLE
''
>>> mtz.history      # from history lines
['From cif2mtz 17/ 5/2019 12:15:14']

>>> # The properties below are also read from the MTZ file,
>>> # although they could be recalculated from the data.
>>> mtz.nreflections   # from MTZ record NCOL
```

(continues on next page)

(continued from previous page)

```
441
>>> mtz.min_1_d2        # from RESO
0.0028703967109323008
>>> mtz.max_1_d2        # from RESO
0.36117017269134527
```

The resolution can also be checked using functions:

```
>>> mtz.resolution_low()    # sqrt(1 / min_1_d2)
18.665044863474492
>>> mtz.resolution_high()   # sqrt(1 / max_1_d2)
1.6639645192598425
```

Importantly, Mtz class has a list of datasets and a list of columns. Datasets are stored in the variable `datasets`:

```
std::vector<Mtz::Dataset> Mtz::datasets
```

```
>>> mtz.datasets
MtzDatasets[<gemmi.Mtz.Dataset 0 HKL_base/HKL_base/HKL_base>, <gemmi.Mtz.Dataset 1␣
↪5e5z/5e5z/1>]
```

In the MTZ file, each dataset is identified internally by an integer "dataset ID". To get dataset with the specified ID use function:

```
Dataset& Mtz::dataset(int id)
```

```
>>> mtz.dataset(0)
<gemmi.Mtz.Dataset 0 HKL_base/HKL_base/HKL_base>
```

Dataset has a few properties that can be accessed directly:

```
struct Dataset {
  int number;
  std::string project_name;
  std::string crystal_name;
  std::string dataset_name;
  UnitCell cell;
  double wavelength = 0.;
};
```

Python bindings provide the same properties:

```
>>> mtz.dataset(0).project_name
'HKL_base'
>>> mtz.dataset(0).crystal_name
'HKL_base'
>>> mtz.dataset(0).dataset_name
'HKL_base'
>>> mtz.dataset(0).cell
<gemmi.UnitCell(9.643, 9.609, 19.029, 90, 101.224, 90)>
>>> mtz.dataset(0).wavelength
0.0
```

Columns are stored in variable `columns`:

```
std::vector<Mtz::Column> Mtz::columns
```

```
>>> mtz.columns[0]
<gemmi.Mtz.Column H type H>
>>> len(mtz.columns)
8
```

To get the first column with the specified label use functions:

```
>>> mtz.column_with_label('FREE')
<gemmi.Mtz.Column FREE type I>
```

If the column names are not unique, you may specify the dataset:

```
>>> mtz.column_with_label('FREE', mtz.dataset(0))
>>> # None
>>> mtz.column_with_label('FREE', mtz.dataset(1))
<gemmi.Mtz.Column FREE type I>
```

To get all columns of the specified type use:

```
>>> mtz.columns_with_type('Q')
MtzColumnRefs[<gemmi.Mtz.Column SIGFP type Q>, <gemmi.Mtz.Column SIGI type Q>]
```

Column has properties read from MTZ headers COLUMN and COLSRC:

```
struct Column {
  int dataset_id;
  char type;
  std::string label;
  float min_value = NAN;
  float max_value = NAN;
  std::string source;  // from COLSRC
  // (and functions and variables that help in accessing data)
};
```

Python bindings provide the same properties:

```
>>> intensity = mtz.column_with_label('I')
>>> intensity.dataset_id
1
>>> intensity.type
'J'
>>> intensity.label
'I'
>>> intensity.min_value, intensity.max_value
(-0.30090001225471497, 216.60499572753906)
>>> intensity.source
'CREATED_17/05/2019_12:15:14'
```

C++ function `Mtz::Column::size()` is wrapped in Python as `__len__`:

```
>>> len(intensity)
441
```

In both C++ and Python `Column` supports the iteration protocol:

```
>>> [x for x in intensity if x > 100]
[124.25700378417969, 100.08699798583984, 216.60499572753906]
>>> numpy.nanmean(intensity)
11.505858
```

Unmerged MTZ

Unmerged (multi-record) MTZ files store a list of batches:

```
std::vector<Mtz::Batch> Mtz::batches
```

```
>>> # here we use mdm2_unmerged.mtz file distributed with CCP4 7.1
>>> mdm2 = gemmi.read_mtz_file(mdm2_unmerged_mtz_path)
>>> len(mdm2.batches)
60
```

Each batch has a number of properties:

```
>>> batch = mdm2.batches[0]
>>> batch.number
1
>>> batch.title
'TITLE Batch 2'
>>> batch.axes
['AXIS']
>>> batch.dataset_id  # link to a dataset
2
>>> mdm2.dataset(2).wavelength
0.8726000000000002
```

Most of the batch header properties are not decoded, but they can be accessed directly if needed:

```
>>> batch.ints
[185, 29, 156, 0, -1, 1, -1, 0, 0, 0, 0, 0, 1, 0, 2, 1, 0, 1, 0, 1, 2, 0, 0, 0, 0, 0,␣
↪0, 0, 0]
>>> batch.floats[36:38]  # start and end of phi
[80.0, 80.5]
```

One pecularity of the MTZ format is that instead of the original Miller indices it stores indices of the equivalent reflection in the ASU + the index of the symmetry operation that transforms the former to the latter. It can be useful to switch between the two:

```
>>> mdm2.switch_to_original_hkl()
True
>>> mdm2.switch_to_asu_hkl()
True
```

The return value indicates that the indices were switched. It would be `False` only if it was a merged MTZ file. Keeping track what the current indices mean is up to the user. They are "asu" after reading a file and they must be "asu" before writing to a file.

Data in NumPy and pandas

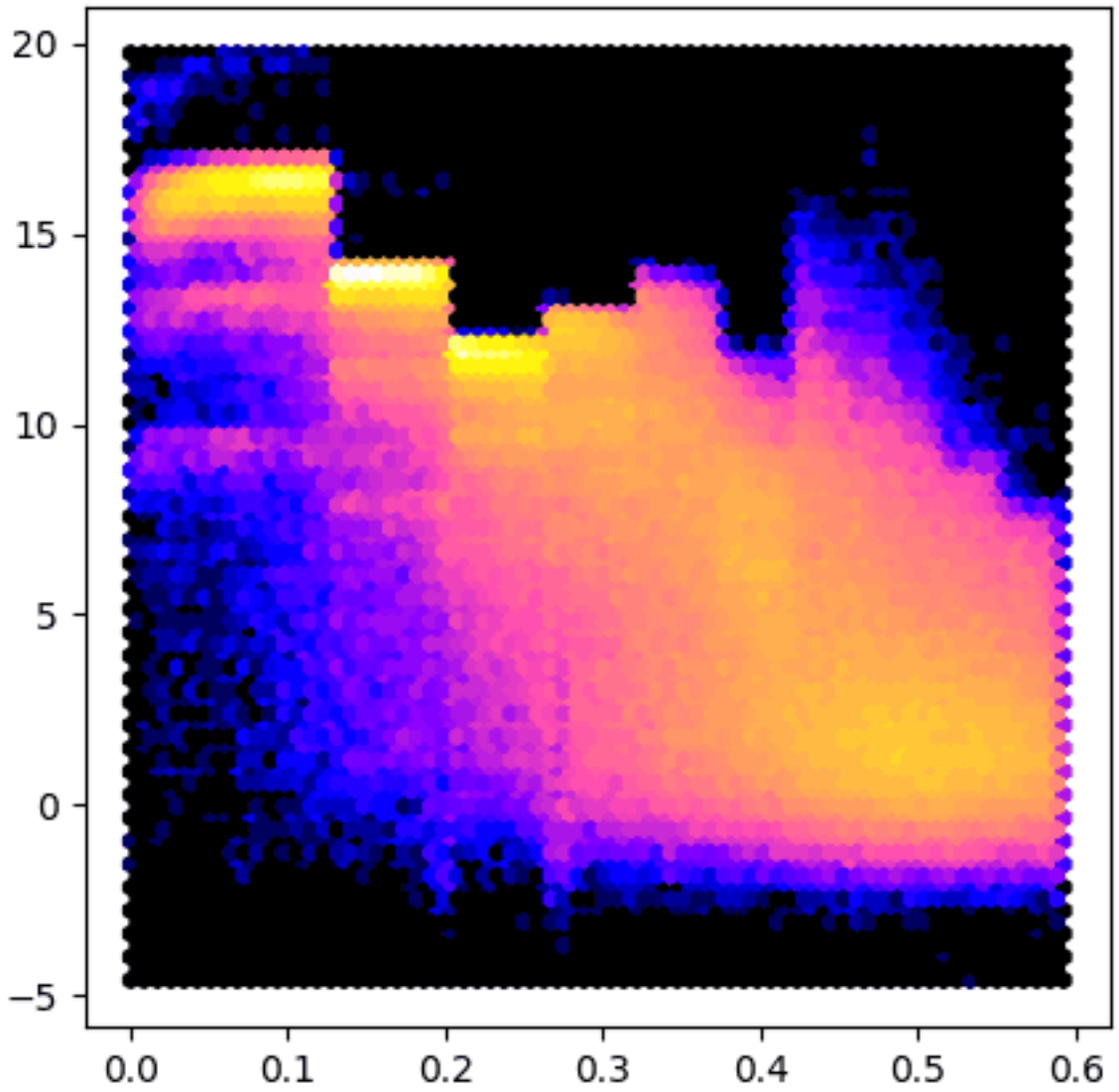
In Python Column has a read-only `array` property that provides a view of the data compatible with NumPy. It does not copy the data, so the data is not contiguous (because it's stored row-wise in MTZ):

```
>>> intensity.array.strides
(32,)
```

Here is an example that uses the array property to make a plot similar to AUSPEX:

```
import sys
from matplotlib import pyplot
import gemmi

for path in sys.argv[1:]:
    mtz = gemmi.read_mtz_file(path)
    intensity = mtz.column_with_label('I')
    sigma = mtz.column_with_label('SIGI')
    if intensity is None or sigma is None:
        sys.exit("Columns I and SIGI not in the file: " + path)
    x = mtz.make_1_d2_array()
    y = intensity.array / sigma.array
    pyplot.figure(figsize=(5,5))
    pyplot.hexbin(x, y, gridsize=180, bins='log', cmap='gnuplot2')
    pyplot.show()
```

To test this program we run it on data from 5DEI and we get result similar to #3:



Another way to access all the MTZ data in Python is through the buffer protocol: For example, to view the data as 2D NumPy array (C-style contiguous) without copying do:

```
>>> import numpy
>>> all_data = numpy.array(mtz, copy=False)
```

It helps to have labels on the columns. A good data structure for this is Pandas DataFrame:

```
>>> import pandas
>>> df = pandas.DataFrame(data=all_data, columns=mtz.column_labels())
>>> # now we can handle columns using their labels:
>>> I_over_sigma = df['I'] / df['SIGI']
```

The data in numpy array is float32. To cast Miller indices in the DataFrame to integer type:

```
>>> df = df.astype({name: 'int32' for name in ['H', 'K', 'L']})
```

You may also have a look at a different project (not associated with Gemmi) for working with reflection data in Python: ReciprocalSpaceship.

Modifying

To change the data in Mtz object we can use function set_data(). In Python, we pass to this function a 2D NumPy array of floating point numbers.

In the previous section, we got such an array (`all_data`) with the original data. Now, as an example, let us remove reflections with the resolution above 2Å (i.e. $d < 2$Å) and copy the result back into `mtz`. To show that it really has an effect we print the appropriate *grid size* before and after:

```
>>> mtz.get_size_for_hkl()
[12, 12, 24]
>>> mtz.set_data(all_data[mtz.make_d_array() >= 2.0])
>>> mtz.get_size_for_hkl()
[10, 10, 20]
```

The metadata in an `Mtz` object can also be modified. To illustrate it, we will create a complete `Mtz` object from scratch. The code below is in Python, but all the functions and properties have equivalents with the same names in C++.

We start from creating an object and setting its space group and unit cell:

```
>>> mtz = gemmi.Mtz()
>>> mtz.spacegroup = gemmi.find_spacegroup_by_name('P 21 21 2')
>>> mtz.cell.set(77.7, 149.5, 62.4, 90, 90, 90)
```

Now we need to add datasets and columns.

```
>>> mtz.add_dataset('HKL_base')
<gemmi.Mtz.Dataset 0 HKL_base/HKL_base/HKL_base>
```

The name passed to `add_dataset()` is used to set `dataset_name`, `crystal_name` and `project_name`. These three names are often kept the same, but if needed, they can be changed afterwards.

The MTZ format stores general, per file cell dimensions (keyword CELL), as well per dataset ones (keyword DCELL). Function `add_dataset` initializes the latter from the former. This also can be changed afterwards.

The first three columns must always be Miller indices:

```
>>> for label in ['H', 'K', 'L']: mtz.add_column(label, 'H')
<gemmi.Mtz.Column H type H>
<gemmi.Mtz.Column K type H>
<gemmi.Mtz.Column L type H>
```

Let's add two more columns in a separate dataset:

```
>>> mtz.add_dataset('synthetic')
<gemmi.Mtz.Dataset 1 synthetic/synthetic/synthetic>
>>> mtz.add_column('F', 'F')
<gemmi.Mtz.Column F type F>
>>> mtz.add_column('SIGF', 'Q')
<gemmi.Mtz.Column SIGF type Q>
```

By default, the new column is assigned to the last dataset and is placed at the end of the column list. But we can choose any dataset and position:

```
>>> mtz.add_column('FREE', 'I', dataset_id=0, pos=3)
<gemmi.Mtz.Column FREE type I>
>>> mtz.column_labels()
['H', 'K', 'L', 'FREE', 'F', 'SIGF']
```

Now it is time to add data. We will use the `set_data()` function that takes 2D NumPy array of floating point numbers (even indices are converted to floats, but they need to be converted at some point anyway – the MTZ format stores all numbers as 32-bit floats).

```
>>> data = numpy.array([[4, 13, 8, 1, 453.9, 19.12],
...                     [4, 13, 9, 0, 102.0, 27.31]], numpy.float)
>>> mtz.set_data(data)
>>> mtz
<gemmi.Mtz with 6 columns, 2 reflections>
```

In C++ the `set_data` function takes a pointer to row-wise ordered data and its size (columns x rows):

```
void Mtz::set_data(const float* new_data, size_t n)
```

To update properties `min_1_d2` and `max_1_d2` call `update_reso()`:

```
>>> mtz.update_reso()
>>> mtz.min_1_d2, mtz.max_1_d2
(0.02664818727144997, 0.03101414716625602)
```

You do not need to call `update_reso()` before writing an MTZ file – the values for the RESO record are re-calculated automatically when the file is written.

Writing

In C++, the MTZ file can be written to a file using one of the functions:

```
void Mtz::write_to_stream(std::FILE* stream) const
void Mtz::write_to_file(const std::string& path) const
```

and in Python using:

```
>>> mtz.write_to_file('output.mtz')
```

1.7.2 SF mmCIF

The mmCIF format is also used to store reflection data from crystallographic experiments. Reflections and coordinates are normally stored in separate mmCIF files. The files with reflections are called structure factor mmCIF or shortly SF mmCIF. Such file for the 1ABC PDB entry is usually named either `1ABC-sf.cif` (if downloaded through RCSB website) or `r1abcsf.ent` (if downloaded through PDBe website or through FTP).

SF mmCIF files usually contain one block, but may have multiple blocks, for example merged and unmerged data in separate blocks. Merged and unmerged data is expected in mmCIF categories `_refln` and `_diffrn_refln`, respectively. Usually, also the unit cell, space group, and sometimes the radiation wavelength is recorded.

The support for SF mmCIF files in Gemmi is built on top of the generic support for the CIF format. We have class `ReflnBlock` that wraps `cif::Block` and a function `as_refln_blocks`:

```
// in C++ it can be called:
// auto rblocks = gemmi::as_refln_blocks(gemmi::read_cif_gz(path).blocks);
std::vector<ReflnBlock> as_refln_blocks(std::vector<cif::Block>&& blocks)
```

In Python this function takes `cif.Document` as an argument:

```
>>> doc = gemmi.cif.read('../tests/r5wkdsf.ent')
>>> doc
<gemmi.cif.Document with 1 blocks (r5wkdsf)>
>>> rblocks = gemmi.as_refln_blocks(doc)
```

Blocks are moved from the Document to the new list:

```
>>> doc
<gemmi.cif.Document with 0 blocks ()>
>>> rblocks
ReflnBlocks[<gemmi.ReflnBlock r5wkdsf with 17 x 406 loop>]
```

When ReflnBlock is created some of the mmCIF tags are interpreted to initialize the following properties:

```
>>> rblock = rblocks[0]
>>> rblock.entry_id
'5wkd'
>>> rblock.cell
<gemmi.UnitCell(50.347, 4.777, 14.746, 90, 101.733, 90)>
>>> rblock.spacegroup
<gemmi.SpaceGroup("C 1 2 1")>
>>> rblock.wavelength
0.9791
```

To check if block has either merged or unmerged data, in C++ use function `ok()`, and in Python:

```
>>> bool(rblock)
True
```

Normally, one block has only one type of data, merged and unmerged. Which one is used can be checked with the function:

```
>>> rblock.is_unmerged()
False
```

But it is syntactically correct to have both types of data in two tables in one block. In such case you can switch which table is used:

```
>>> rblock.use_unmerged(True)
>>> rblock.use_unmerged(False)
```

Finally, ReflnBlock has functions for working with the data table:

```
std::vector<std::string> ReflnBlock::column_labels() const

size_t ReflnBlock::get_column_index(const std::string& tag) const

template<typename T>
std::vector<T> ReflnBlock::make_vector(const std::string& tag, T null) const

std::vector<std::array<int,3>> ReflnBlock::make_index_vector() const

std::vector<double> ReflnBlock::make_1_d2_vector() const

std::vector<double> ReflnBlock::make_d_vector() const
```

We will describe these functions while going through its Python equivalents. `column_labels()` returns list of tags associated with the columns, excluding the category part. Unlike in the MTZ format, here the tags must be unique.

```
>>> rblock.column_labels()
['crystal_id', 'wavelength_id', 'scale_group_code', 'index_h', 'index_k', 'index_l',
↪'status', 'pdbx_r_free_flag', 'F_meas_au', 'F_meas_sigma_au', 'F_calc_au', 'phase_
↪calc', 'pdbx_FWT', 'pdbx_PHWT', 'pdbx_DELFWT', 'pdbx_DELPHWT', 'fom']
```

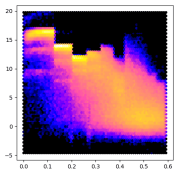
All the data is stored as strings. We can get integer or real values from the selected column in an array. In Python – in NumPy array:

```
>>> rblock.make_int_array('index_h', -1000)  # 2nd arg - value for nulls
array([-26, -26, -26, ...,  25,  26,  26], dtype=int32)

>>> rblock.make_float_array('F_meas_au')  # by default, null values -> NAN
array([12.66, 13.82, 24.11, ...,   nan,  9.02,   nan])
>>> rblock.make_float_array('F_meas_au', 0.0)  # use 0.0 for nulls
array([12.66, 13.82, 24.11, ...,  0.  ,  9.02,  0.  ])
```

We also have convenience functions that returns arrays of $1/d^2$ or just d values:

```
>>> rblock.make_1_d2_array().round(4)
array([0.2681, 0.2677, 0.2768, ..., 0.301 , 0.2782, 0.2978])
>>> rblock.make_d_array().round(3)
array([1.931, 1.933, 1.901, ..., 1.823, 1.896, 1.832])
```

Example 1



The script below renders the same colorful I/σ image as in the previous section, but it can take as an argument a file downloaded directly from the wwPDB (for example, `$PDB_DIR/structures/divided/structure_factors/de/r5deisf.ent.gz`).

```
import sys
from matplotlib import pyplot
import gemmi

for path in sys.argv[1:]:
    doc = gemmi.cif.read(path)
    rblock = gemmi.as_refln_blocks(doc)[0]
    intensity = rblock.make_float_array('intensity_meas')
    sigma = rblock.make_float_array('intensity_sigma')
    x = rblock.make_1_d2_array()
    y = intensity / sigma
    pyplot.figure(figsize=(5,5))
    pyplot.hexbin(x, y, gridsize=70, bins='log', cmap='gnuplot2')
    pyplot.show()
```

Example 2

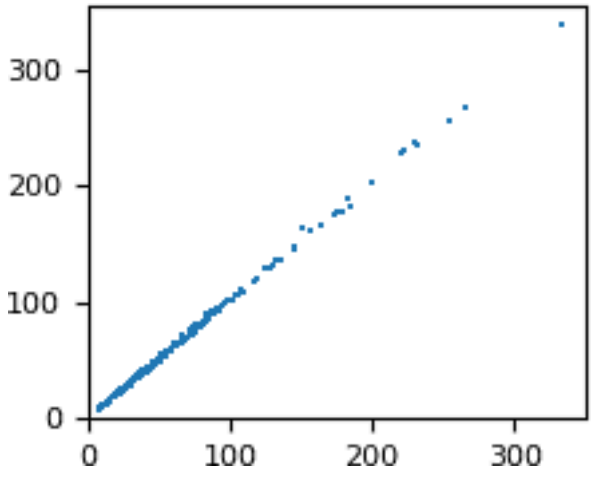
In this example we compare columns from two files using Python pandas.

Here we use the 5WKD entry, which has only 367 measured reflections, but the same script works fine even for 1000x more reflections. You'd only need to tweak the plots to make them more readable.

We use two data files (SF-mmCIF MTZ) downloaded from the RCSB website. First, we make pandas DataFrames from both files, and then we merge them into a single DataFrame:

```
import gemmi
import numpy
import pandas
from matplotlib import pyplot

MTZ_PATH = 'tests/5wkd_phases.mtz.gz'
SFCIF_PATH = 'tests/r5wkdsf.ent'

# make DataFrame from MTZ file
mtz = gemmi.read_mtz_file(MTZ_PATH)
mtz_data = numpy.array(mtz, copy=False)
mtz_df = pandas.DataFrame(data=mtz_data, columns=mtz.column_labels())
# (optional) store Miller indices as integers
mtz_df = mtz_df.astype({label: 'int32' for label in 'HKL'})

# make DataFrame from mmCIF file
cif_doc = gemmi.cif.read(SFCIF_PATH)
rblock = gemmi.as_refln_blocks(cif_doc)[0]
cif_df = pandas.DataFrame(data=rblock.make_index_array(), columns=['H','K','L'])
cif_df['F_meas_au'] = rblock.make_float_array('F_meas_au')
cif_df['d'] = rblock.make_d_array()

# merge DataFrames
df = pandas.merge(mtz_df, cif_df, on=['H', 'K', 'L'])
```

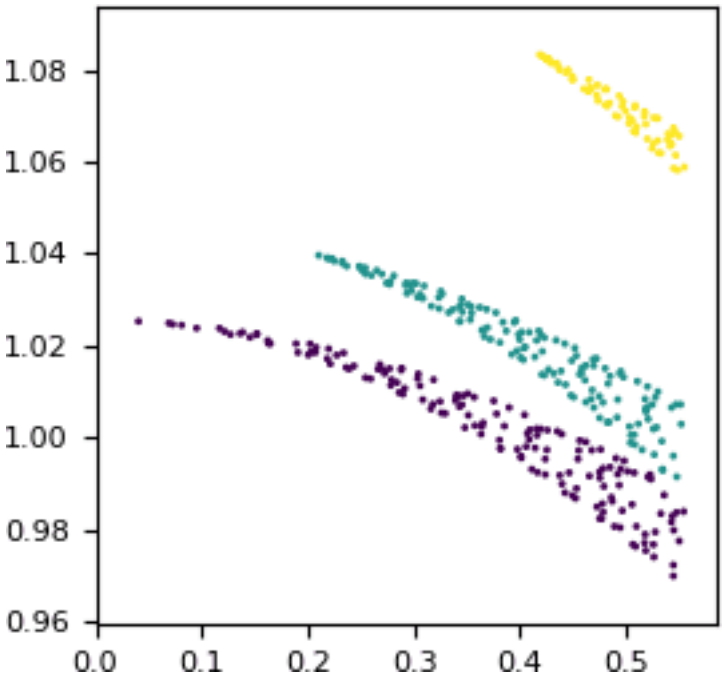
We want to compare the FP column from the MTZ file and the F_meas_au column from mmCIF. We start with plotting one against the other:




```
# plot FP from MTZ as a function of F_meas_au from mmCIF
pyplot.rc('font', size=8)
pyplot.figure(figsize=(2, 2))
pyplot.scatter(x=df['F_meas_au'], y=df['FP'],
               marker=',', s=1, linewidths=0)
pyplot.xlim(xmin=0)
pyplot.ylim(ymin=0)
pyplot.show()
```

The numbers are similar, but not exactly equal. Let us check how the difference between the two values depends on the resolution. We will plot the difference as a function of $1/d$. This extremely small dataset has only three Miller

indices *k* (0, 1 and 2), so we can use it for coloring.




```
# plot the ratio FP : F_meas_au as a function of 1/d
pyplot.figure(figsize=(3, 3))
pyplot.scatter(x=1/df['d'],
               y=df['FP']/df['F_meas_au'],
               c=df['K'],  # color by index k
               marker='.', s=16, linewidths=0)
pyplot.xlim(xmin=0)
pyplot.show()
```

Apparently, some scaling has been applied. The scaling is anisotropic and is the strongest along the *k* axis.

1.7.3 hkl CIF

In the small molecule world reflections are also stored in separate CIF files. Similarly to SF mmCIF files, we may take cif::Block and wrap it into the ReflnBlock class:

```
>>> doc = gemmi.cif.read('../tests/2242624.hkl')
>>> gemmi.hkl_cif_as_refln_block(doc[0])
<gemmi.ReflnBlock 2242624 with 7 x 71 loop>
```

The methods of ReflnBlock are the same as in the previous section.

```
>>> print(_.make_d_array(), _.make_float_array('F_squared_meas'), sep='\n')
[1.7710345  1.03448487 0.71839568 ... 0.67356073 0.69445847 0.56034145]
[173.17 246.01  11.61 ...  84.99  46.98  16.99]
```

These hkl files in the CIF format are not to be confused with the SHELX hkl files. The latter is often included in the coordinate cif file, as a text value of _shelx_hkl_file:

```
>>> cif_doc = gemmi.cif.read('../tests/4003024.cif')
>>> hkl_str = gemmi.cif.as_string(cif_doc[0].find_value('_shelx_hkl_file'))
```

(continues on next page)

(continued from previous page)

```
>>> print(hkl_str[:87])

 -1   0   0   20.05    0.74
  0   0  -1   20.09    0.73
  0   0   1   19.76    0.73
```

1.7.4 Data on 3D grid

The reciprocal space data can be alternatively presented on a 3D grid indexed by Miller indices. This grid is represented by C++ class ReciprocalGrid, which shares most of the properties with the real-space class *Grid*.

In C++, the `<gemmi/fourier.hpp>` header defines templated function `get_f_phi_on_grid()` that can be used with both MTZ and SF mmCIF data. Here, we focus on the usage from Python.

Both Mtz and ReflnBlock classes have method `get_f_phi_on_grid` that takes three mandatory arguments: column names for the amplitude and phase, and the grid size. It returns reciprocal grid of complex numbers (`ReciprocalGrid<std::complex<float>>` in C++, `ReciprocalComplexGrid` in Python). Symmetry-related reflections are filled automatically (with phase shift). All the missing values are set to 0:

```
>>> grid = rblock.get_f_phi_on_grid('pdbx_FWT', 'pdbx_PHWT', [54,6,18])
>>> grid
<gemmi.ReciprocalComplexGrid(54, 6, 18)>
```

If we'd ever need data from a single column put on a grid, we can use analogical function:

```
>>> rblock.get_value_on_grid('F_meas_au', [54,6,18])
<gemmi.ReciprocalFloatGrid(54, 6, 18)>
```

The grids in examples above have capacity to store reflections with -27<*h*<27, -3<*k*<3 and -9<*l*<9. Reflections outside of this range would be silently ignored. To check if the size is big enough you can call:

```
>>> rblock.data_fits_into([54,6,18])
True
>>> rblock.data_fits_into([52,6,18])
False
```

To access the data you can use either the buffer protocol (*in the same way* as in the Grid class), or getter and setter:

```
>>> grid.get_value(7, 1, -5)
(31.747737884521484-57.06287384033203j)
>>> grid.set_value(7, 1, -5, 12+34j)
>>> grid.get_value(7, 1, -5)
(12+34j)
```

The functions above throw IndexError if the reflection is outside of the grid. If you prefer zero instead of the error, use:

```
>>> grid.get_value_or_zero(20, 30, 40)
0j
```

We can also iterate over points of the grid.

```
>>> for point in grid:
...     pass
>>> point.value  # point is the last point from the iteration
```

(continues on next page)

(continued from previous page)

```
(-178.310546875+99.205612182617l9j)
>>> grid.to_hkl(point)
[-1, -1, -1]
```

Grid size

To get an appropriate size, we can use method `get_size_for_hkl` that has two optional parameters: `min_size` and `sample_rate`. `min_size` sets explicitely the minimal size of the grid:

```
>>> rblock.get_size_for_hkl()
[54, 6, 18]
>>> rblock.get_size_for_hkl(min_size=[64,8,0])
[64, 8, 18]
```

The actual size can be increased to make room for all reflections, to obey restrictions imposed by the spacegroup, and to make the size FFT-friendly (currently this means factors 2, 3 and 5).

`sample_rate` sets the minimal grid size in relation to d_{min}. It has the same meaning as the keyword SAMPLE in the CCP4 FFT program. For example, `sample_rate=3` requests grid size that corresponds to real-space sampling $d_{\mathrm{min}}/3$. (N.B. 3 here is equivalent to Clipper oversampling parameter equal 1.5).

```
>>> rblock.get_size_for_hkl(sample_rate=3.0)
[90, 8, 30]
```

Array layout

The `get_f_phi_on_grid` function has also two optional arguments: `half_l` and `axis_order`. The `half_l` flag is used to shrink the size of the grid in the memory. When set, the grid does not include data with negative index *l*. If the data is Hermitian, i.e. if it is a Fourier transform of the real data (electron density), the (*h k l*) reflection with negative *l* can be restored as a complex conjugate of its Friedel mate (*-h -k -l*).

```
>>> rblock.get_f_phi_on_grid('pdbx_FWT', 'pdbx_PHWT', [54,6,18], half_l=True)
<gemmi.ReciprocalComplexGrid(54, 6, 10)>
```

`axis_order` can take one of the two values : `AxisOrder.XYZ` (the default) or `AxisOrder.ZYX`. With the former – the *h* direction is along the first (fast) axis of the grid. The latter results in *l* along the fast axis. (The fast axis is first, which is a Fortran convention. This convention affected the design of the CCP4 format, which in turn affected the design of the grid classes in Gemmi.)

```
>>> rblock.get_f_phi_on_grid('pdbx_FWT', 'pdbx_PHWT', [54,6,18], order=gemmi.
↪AxisOrder.ZYX)
<gemmi.ReciprocalComplexGrid(18, 6, 54)>
```

Example

As an example, we will use numpy.fft to calculate electron density map from map coefficients. Gemmi can calculate it internally, as described in the next section, but it is instructive to do it with a general-purpose tool.

```
>>> size = rblock.get_size_for_hkl(sample_rate=2.6)
>>> full = rblock.get_f_phi_on_grid('pdbx_FWT', 'pdbx_PHWT', size)
>>> array = numpy.array(full, copy=False)
```

(continues on next page)

(continued from previous page)

```
>>> complex_map = numpy.fft.ifftn(array.conj())
>>> scale_factor = complex_map.size / full.unit_cell.volume
>>> real_map = numpy.real(complex_map) * scale_factor
>>> round(real_map[1][2][3], 5)
-0.40554
>>> round(real_map.std(), 5)
0.66338
```

Above, we could use `fftn(array)` instead of `ifftn(array.conj())`, but the inverse FFT is more appropriate here (or so I think).

Since the data is Hermitian, using complex-to-complex FFT is equivant to using complex-to-real FFT on a half of the data:

```
>>> half = rblock.get_f_phi_on_grid('pdbx_FWT', 'pdbx_PHWT', size, half_l=True)
>>> array = numpy.array(half, copy=False)
>>> real_map = numpy.fft.irfftn(array.conj()) * scale_factor
>>> round(real_map[1][2][3], 5)
-0.40554
>>> round(real_map.std(), 5)
0.66338
```

1.7.5 FFT

Internally, Gemmi is using the PocketFFT library to transform between real space and reciprocal space. This library was picked after evaluation and benchmarking of many FFT libraries.

In C++, the relevant functions are in the `<gemmi/fourier.hpp>` header, and the *gemmi-sf2map* program may serve as a code example. Like in the previous section, here we will cover only the Python interface.

Instead of using numpy.fft as in the example above, we can use `gemmi.transform_f_phi_grid_to_map()` and we expect to get the same result (wrapped in a *Grid* class):

```
>>> gemmi.transform_f_phi_grid_to_map(half)
<gemmi.FloatGrid(72, 8, 24)>
>>> round(_.get_value(1, 2, 3), 5)
-0.40554
```

Both Mtz and ReflnBlock classes have method `transform_f_phi_to_map` that combines `get_f_phi_on_grid()` with `transform_f_phi_grid_to_map()`.

`transform_f_phi_to_map` takes column names for amplitude and phase (in degrees), and optional parameters `min_size`, `exact_size` and `sample_rate` (normally only one of them is given):

```
>>> rblock.transform_f_phi_to_map('pdbx_FWT', 'pdbx_PHWT', sample_rate=4)
<gemmi.FloatGrid(120, 12, 36)>
>>> rblock.transform_f_phi_to_map('pdbx_FWT', 'pdbx_PHWT', min_size=[127,16,31])
<gemmi.FloatGrid(128, 16, 32)>
>>> rblock.transform_f_phi_to_map('pdbx_FWT', 'pdbx_PHWT', exact_size=[70,8,24])
<gemmi.FloatGrid(70, 8, 24)>
```

The grid can be accessed as NumPy 3D array:

```
>>> array = numpy.array(_, copy=False)
>>> round(array.std(), 5)
0.66338
```

and it can be stored in a CCP4 map format:

```
>>> ccp4 = gemmi.Ccp4Map()
>>> ccp4.grid = rblock.transform_f_phi_to_map('pdbx_FWT', 'pdbx_PHWT', sample_rate=2.
↪6)
>>> ccp4.update_ccp4_header(2, True)
>>> ccp4.write_ccp4_map('5wkd.ccp4')
```

To transform the electron density back to reciprocal space coefficients use function `transform_map_to_f_phi`:

```
>>> ccp4.grid
<gemmi.FloatGrid(72, 8, 24)>
>>> gemmi.transform_map_to_f_phi(ccp4.grid)
<gemmi.ReciprocalComplexGrid(72, 8, 24)>
```

Now you can access hkl reflections using `Grid.get_value()`:

```
>>> _.get_value(23, -1, -3)
(18.440279006958008+26.189243316650395j)
```

`transform_map_to_f_phi` has one optional flag: `half_l`. It has the same meaning as in the function `get_f_phi_on_grid`. When you set `half_l=True` you cannot access directly a data point with negative Miller index l, but you can use its Friedel mate:

```
>>> gemmi.transform_map_to_f_phi(ccp4.grid, half_l=True)
<gemmi.ReciprocalComplexGrid(72, 8, 13)>
>>> _.get_value(-23, 1, 3).conjugate()  # value for (23, -1, -3)
(18.440279006958008+26.189243316650395j)
```

Then again, you can use `transform_f_phi_grid_to_map()` to transform it back to the direct space, and so on...

1.7.6 Scattering factors

Atomic form factor

TODO

(also: direct calculation or FFT, tradoffs between performance and accuracy)

Anomalous scattering

The anomalous dispersion is wavelength dependent. Gemmi provides function `cromer_libermann` that calculates real and imaginary components *f'* and *f"* for isolated atoms from Z=3 to Z=92.

As the name suggests, we use the Cromer-Libermann algorithm. This algorithm, as is noted on the pyFprime website, "fails in computing *f'* for wavelengths < 0.16 Å (> 77.48 keV) for the heaviest elements (Au-Cf) and fails to correctly compute *f'*, *f"* and μ for wavelengths > 2.67 Å (< 4.64 keV) for very heavy elements (Am-Cf)."

The implementation is contained in a single C++ header file `fprime.hpp` with no dependencies. All the data is embedded in the code. The binary size after compilation is about 100kB.

Admittedly, the data tables synthesised by C.T. Chantler are more accurate. Consider using them instead. They are available from the NIST website and from the XrayDB project.

The code in `fprime.hpp` is based on the X-ray spectroscopy project Larch and should give the same results as the f1f2_cl function there. The Fortran code in Larch is, in turn, based on the Brennan and Cowan routines. Which,

in turn, were based on the original program from Don Cromer. Along the way, the code was extensively modified. Importantly, the Jensen correction has been removed (as is recommended in the chapter 4.2.6 of ITvC) and the Kissel and Pratt (1990) correction has been added. Therefore, it gives different results than the crossec program, which was contributed to CCP4 directly by Don Cromer in the 1990's.

The `cromer_libermann` function is available in both C++ and Python:

```
>>> gemmi.Element('Se').atomic_number
34
>>> gemmi.cromer_libermann(z=_, energy=10332.0) # energy in eV
(-1.4186231113544407, 0.7238969498014027)
>>> # use gemmi.hc to convert wavelength [A] to energy [eV]
>>> gemmi.cromer_libermann(z=34, energy=gemmi.hc/0.71073)
(-0.09201659699543367, 2.2333699118331087)
```

The same values can be printed from the command line program *gemmi-fprime*.

Bulk solvent correction

TODO

1.8 Gemmi program

The library comes with a command-line program which is also named gemmi; running a program is easier than calling a library function.

This program is actually a set of small programs, each of them corresponding to a subcommand:

```
$ gemmi -h
gemmi 0.3.8
Command-line utility that accompanies the GEMMI library,
which is a joint project of CCP4 and Global Phasing Ltd.
Licence: Mozilla Public License 2.0.
Copyright 2017-2020 Global Phasing Ltd.
https://github.com/project-gemmi/gemmi

Usage: gemmi [--version] [--help] <command> [<args>]

Commands:
 align          sequence alignment (global, pairwise, affine gap penalty)
 blobs          list unmodelled electron density blobs
 cif2mtz        convert structure factor mmCIF to MTZ
 cif2json       translate (mm)CIF to (mm)JSON
 contact        searches for contacts (neighbouring atoms)
 contents       info about content of a coordinate file (pdb, mmCIF, ...)
 convert        convert file (CIF - JSON, mmCIF - PDB) or modify structure
 fprime         calculate anomalous scattering factors f' and f"
 grep           search for tags in CIF file(s)
 h              add or remove hydrogen atoms
 json2cif       translate mmJSON to mmCIF
 map            print info or modify a CCP4 map
 map2sf         transform CCP4 map to map coefficients (in MTZ or mmCIF)
 mask           make mask in the CCP4 format
 mondiff        compare two monomer CIF files
 mtz            print info about MTZ reflection file
```

(continues on next page)

(continued from previous page)

```
 mtz2cif       convert MTZ to structure factor mmCIF
 reindex       reindex MTZ file
 residues      list residues from a coordinate file
 rmsz          validate geometry using monomer library
 sf2map        transform map coefficients (from MTZ or mmCIF) to map
 sfcalc        calculate structure factors from a model
 sg            info about space groups
 tags          list tags from CIF file(s)
 validate      validate CIF 1.1 syntax
 wcn           calculate local density / contact numbers (WCN, CN, ACN, LDM)
```

1.8.1 validate

A CIF validator. Apart from checking the syntax it can check most of the rules imposed by DDL1 and DDL2 dictionaries.

```
$ gemmi validate -h
Usage: gemmi validate [options] FILE [...]

Options:
  -h, --help      Print usage and exit.
  -V, --version   Print version and exit.
  -v, --verbose   Verbose output.
  -q, --quiet     Show only errors.
  -f, --fast      Syntax-only check.
  -s, --stat      Show token statistics
  -d, --ddl=PATH  DDL for validation.
  -m, --monomer   Extra checks for Refmac dictionary files.
```

1.8.2 grep

Searches for a specified tag in CIF files and prints the associated values, one value per line:

```
$ gemmi grep _refine.ls_R_factor_R_free 5fyi.cif.gz
5FYI:0.2358
$ gemmi grep _refine.ls_R_factor_R_free mmCIF/mo/?moo.cif.gz
1MOO:0.177
3MOO:0.21283
4MOO:0.22371
5MOO:0.1596
5MOO:0.1848
$ gemmi grep -b _software.name 5fyi.cif.gz
DIMPLE
PHENIX
```

Some of the command-line options correspond to the options of GNU grep (`-c`, `-l`, `-H`, `-n`). As with other utilities, option `--help` shows the usage:

```
$ gemmi grep -h
Usage: gemmi grep [options] TAG FILE_OR_DIR_OR_PDBID[...]
       gemmi grep -f FILE [options] TAG
Search for TAG in CIF files.
By default, recursive directory search checks only *.cif(.gz) files.
```

(continues on next page)

(continued from previous page)

```
To change it, specify --name=* or --name=*.hkl.

Options:
  -h, --help               display this help and exit
  -V, --version            display version information and exit
  -f, --file=FILE          obtain file (or PDB ID) list from FILE
  --name=PATTERN           filename glob pattern used in recursive grep; by
                           default, *.cif and *.cif.gz files are searched
  -m, --max-count=NUM      print max NUM values per file
  -O, --one-block          optimize assuming one block per file
  -a, --and=tag            Append delimiter (default ';') and the tag value
  -d, --delimiter=DELIM    CSV-like output with specified delimiter
  -n, --line-number        print line number with output lines
  -H, --with-filename      print the file name for each match
  -b, --no-blockname       suppress the block name on output
  -t, --with-tag           print the tag name for each match
  -l, --files-with-tag     print only names of files with the tag
  -L, --files-without-tag  print only names of files without the tag
  -c, --count              print only a count of values per block or file
  -r, --recursive          ignored (directories are always recursed)
  -w, --raw                include '?', '.', and string quotes
  -s, --summarize          display joint statistics for all files
```

This is a minimalistic program designed to be used together with Unix text-processing utilities. For example, it cannot filter values itself, but one may use grep:

```
$ gemmi grep _pdbx_database_related.db_name /pdb/mmCIF/aa/* | grep EMDB
4AAS:EMDB
5AA0:EMDB
```

Gemmi-grep tries to be simple to use like Unix grep, but at the same time it is aware of the CIF syntax rules. In particular, `gemmi grep _one` will give the same output for both `_one 1` and `loop_ _one _two 1 2`. This is helpful in surprising corner cases. For example, when a PDB entry has two Rfree values (see the 5MOO example above).

Gemmi-grep does not support regular expression, only globbing (wildcards): `?` represents any single character, `*` represents any number of characters (including zero). When using wildcards you may also want to use the `-t` option which prints the tag:

```
$ gemmi grep -t _*free 3gem.cif
3GEM:[_refine.ls_R_factor_R_free] 0.182
3GEM:[_refine.ls_percent_reflns_R_free] 5.000
3GEM:[_refine.ls_number_reflns_R_free] 3951
3GEM:[_refine.correlation_coeff_Fo_to_Fc_free] 0.952
3GEM:[_refine_ls_shell.R_factor_R_free] 0.272
3GEM:[_refine_ls_shell.number_reflns_R_free] 253
```

Let say we want to find extreme unit cell angles in the PDB. `_cell.angle_*a` will match _cell.angle_alpha as well as beta and gamma, but not _cell.angle_alpha_esd etc.

```
$ gemmi grep -d' ' _cell.angle_*a /pdb/mmCIF/ | awk '$2 < 50 || $2 > 140 { print $0; }
↪'
4AL2 144.28
2EX3 45.40
2GMV 145.09
4NX1 140.060
```

(continues on next page)

(continued from previous page)

```
4OVP 140.070
1SPG 141.90
2W1I 146.58
```

The option `-O` is used to make gemmi-grep faster. With this option the program finds only the first occurence of the tag in file. Note that if the file has only one block (like mmCIF coordinate files) and the tag is specified without wildcards then we cannot have more than one match anyway.

Searching the whole compressed mmCIF archive from the PDB (35GB of gzipped files) should take on an average computer between 10 and 30 minutes, depending where the searched tag is located. This is much faster than with other CIF parsers (to my best knowledge) and it makes the program useful for ad-hoc PDB statistics:

```
$ gemmi grep -O -b _entity_poly.type /pdb/mmCIF | sort | uniq -c
      1 cyclic-pseudo-peptide
      4 other
      2 peptide nucleic acid
   9905 polydeoxyribonucleotide
    156 polydeoxyribonucleotide/polyribonucleotide hybrid
     57 polypeptide(D)
 168923 polypeptide(L)
   4559 polyribonucleotide
     18 polysaccharide(D)
```

Option `-c` counts the values in each block or file. As an example we may check which entries have the biggest variety of chemical components (spoiler: ribosomes):

```
$ gemmi grep -O -c _chem_comp.id /pdb/mmCIF | sort -t: -k2 -nr | head
5J91:58
5J8A:58
5J7L:58
5J5B:58
4YBB:58
5JC9:57
5J88:57
5IT8:57
5IQR:50
5AFI:50
```

Going back to moo, we may want to know to what experimental method the Rfree values correspond:

```
$ gemmi grep _refine.ls_R_factor_R_free -a _refine.pdbx_refine_id mmCIF/mo/?moo.cif.gz
1MOO:0.177;X-RAY DIFFRACTION
3MOO:0.21283;X-RAY DIFFRACTION
4MOO:0.22371;X-RAY DIFFRACTION
5MOO:0.1596;X-RAY DIFFRACTION
5MOO:0.1848;NEUTRON DIFFRACTION
```

Option `-a` (`--and`) can be specified many times. If we would add `-a _pdbx_database_status.recvd_initial_deposition_date` we would get the deposition date in each line. In this case it would be repeated for 5MOO:

```
5MOO:0.1596;X-RAY DIFFRACTION;2016-12-14
5MOO:0.1848;NEUTRON DIFFRACTION;2016-12-14
```

To output TSV (tab-separated values) add `--delimiter='\t'`. What are the heaviest chains?

```
$ gemmi grep --delimiter='\t' _entity.formula_weight -a _entity.pdbx_description /hdd/
↪mmCIF/ | sort -nrk2 | head -3
6EK0    1641906.750     28S ribosomal RNA
5T2C    1640238.125     28S rRNA
5LKS    1640238.125     28S ribosomal RNA
```

With some further processing the option `-a` can be used to generate quite sophisticated reports. Here is a little demo: https://project-gemmi.github.io/pdb-stats/

The major limitation here is that gemmi-grep cannot match corresponding values from different tables (it is not possible on the syntax level). In the example above we have two values from the same table (`_refine`) and a deposition date (single value). This works well. But we are not able to add corresponding wavelengths from `_diffrn_source`. If an extra tag (specified with `-a`) is not in the same table as the main tag, gemmi-grep uses only the first value for this tag.

Unless we just count the number of value. Counting works for any combination of tags:

```
$ gemmi grep -c _refln.intensity_meas -a _diffrn_refln.intensity_net r5paysf.ent.gz
r5paysf:63611;0
r5payAsf:0;356684
```

(The file used in this example is structure factor (SF) mmCIF. Strangely these files in the PDB have extension `ent` not `cif`.)

The first number in the output above is the number of specified intensities. If you would like to count in also values `?` and `.` specify the option `--raw`:

```
$ gemmi grep --raw -c _refln.intensity_meas r5paysf.ent.gz
r5paysf:63954
r5payAsf:0
```

Gemmi-grep can work with any CIF files but it has one feature specific to the PDB data. When *$PDB_DIR* is set one may use PDB codes: just `5moo` or `5MOO` instead of the path to `5moo.cif.gz`. And for convenience, using a PDB code implies option `-O`.

The file paths or PDB codes can be read from a file. For example, if we want to analyse PDB data deposited in 2016 we may first make a file that lists all such files:

```
$ gemmi grep -H -O _pdbx_database_status.recvd_initial_deposition_date $PDB_DIR/
↪structures/divided/mmCIF | \
        grep 2016 >year2016.txt
```

The 2016.txt file file has lines that start with the filename:

```
/hdd/structures/divided/mmCIF/ww/5ww9.cif.gz:5WW9:2016-12-31
/hdd/structures/divided/mmCIF/ww/5wwc.cif.gz:5WWC:2016-12-31
```

and a command such as:

```
$ gemmi grep -f year2016.out _diffrn.ambient_temp
```

will grep only the listed cif files.

Exit status of gemmi-grep has the same meaning as in GNU grep: 0 if a line is selected, 1 if no lines were selected, and 2 if an error occurred.

Examples

comp_id check

The monomer library (Refmac dictionary) has tags such as `_chem_comp_atom.comp_id`, `_chem_comp_bond.comp_id` that are expected to be consistent with the block name:

```
$ gemmi grep _*.comp_id $CLIBD_MON/a/ASN.cif
comp_ASN:ASN
[repeated 106 times]
```

We can quickly check if the names are always consistent by filtering the output above with awk, for all monomer files, to print only lines where the block name and comp_id differ:

```
$ gemmi grep _*.comp_id $CLIBD_MON/? | awk -F: 'substr($1, 6) != $2'
comp_M43:N09
...
```

planarity

The monomer library includes planarity restraints. Each row in the `_chem_comp_plane_atom` table with the same `plane_id` represents atom belonging to the same plane. What is the maximum number of atoms in one plane?

```
$ gemmi grep _chem_comp_plane_atom.plane_id $CLIBD_MON/? | uniq -c | sort -nr | head -
↪3
 38 comp_LG8:plan-1
 36 comp_UCM:plan-1
 36 comp_SA3:plan-1
```

1.8.3 cif2json

Syntax-level conversion from CIF 1.1 to JSON. The JSON representation of the CIF data can be customized. In particular we support CIF-JSON standard from COMCIFS and mmJSON standard from PDBj (the latter is specific to mmCIF files).

```
$ gemmi cif2json -h
Usage:
 gemmi cif2json [options] INPUT_FILE OUTPUT_FILE

Convert CIF file (any CIF files, including mmCIF) to JSON.
The output can be COMCIFS CIF-JSON (-c), mmJSON (-m),
or a custom JSON flavor (default).

General options:
  -h, --help             Print usage and exit.
  -V, --version          Print version and exit.
  -v, --verbose          Verbose output.

JSON output options:
  -c, --comcifs          Conform to the COMCIFS CIF-JSON standard draft.
  -m, --mmjson           Compatible with mmJSON from PDBj.
  --bare-tags            Output tags without the first underscore.
  --numb=quote|nosu|mix  Convert the CIF numb type to one of:
                           quote - string in quotes,
                           nosu - number without s.u.,
```

(continues on next page)

(continued from previous page)

```
                          mix (default) - quote only numbs with s.u.
  --dot=STRING           JSON representation of CIF's '.' (default: null).

Modifications:
  --skip-category=CAT    Do not output tags starting with _CAT
  --sort                 Sort tags in alphabetical order.

When output file is -, write to standard output.
```

The major difference between the two is that CIF-JSON is dictionary-agnostic: it cannot recognize categories (mmJSON groups by categories), and it cannot recognize numbers (so it quotes the numbers). CIF-JSON adds also two extra objects: "CIF-JSON" and "Metadata". The minor differences are:

CIF	CIF-JSON	mmJSON
data_a	a	data_a
_tag	_tag	tag
_CasE	_case	CasE
.	false	null
?	null	null

1.8.4 json2cif

The opposite of cif2json, but currently the only supported input is mmJSON.

```
$ gemmi json2cif -h
Usage:
 gemmi json2cif [options] INPUT_FILE OUTPUT_FILE

Convert mmJSON to mmCIF.

Options:
  -h, --help           Print usage and exit.
  -V, --version        Print version and exit.
  -v, --verbose        Verbose output.
  --pdbx-style         Similar styling (formatting) as in wwPDB.
  --cif2cif            Read CIF not JSON.
  --skip-category=CAT  Do not output tags starting with _CAT
  --sort               Sort tags in alphabetical order.

When output file is -, write to standard output.
```

1.8.5 convert

Conversion between macromolecular coordinate formats: PDB, mmCIF and mmJSON.

```
$ gemmi convert -h
Usage:
 gemmi convert [options] INPUT_FILE OUTPUT_FILE

with possible conversions between PDB, mmCIF and mmJSON.
FORMAT can be specified as one of: mmcif, mmjson, pdb, ccd (read-only).
ccd = coordinates of a chemical component from CCD or monomer library.
```

(continues on next page)

(continued from previous page)

```
General options:
  -h, --help             Print usage and exit.
  -V, --version          Print version and exit.
  -v, --verbose          Verbose output.
  --from=FORMAT          Input format (default: from the file extension).
  --to=FORMAT            Output format (default: from the file extension).

CIF output options:
  --pdbx-style           Similar styling (formatting) as in wwPDB.
  -b NAME, --block=NAME  Set block name and default _entry.id
  --sort                 Sort tags in alphabetical order.
  --skip-category=CAT    Do not output tags starting with _CAT

PDB input options:
  --segment-as-chain     Append segment id to label_asym_id (chain name).
  --old-pdb              Read only the first 72 characters in line.

PDB output options:
  --short-ter            Write PDB TER records without numbers (iotbx compat.).
  --linkr                Write LINKR record (for Refmac) if link_id is known.

Any output options:
  --minimal              Write only the most essential records.
  --shorten              Shorten chain names to 1 (if # < 63) or 2 characters.

Macromolecular operations:
  --expand-ncs=dup|new   Expand strict NCS specified in MTRIXn or equivalent.
                         New chain names are the same or have added numbers.
  --assembly=ID          Output bioassembly with given ID (1, 2, ...).
  --remove-h             Remove hydrogens.
  --remove-waters        Remove waters.
  --remove-lig-wat       Remove ligands and waters.
  --trim-to-ala          Trim aminoacids to alanine.

When output file is -, write to standard output.
```

The PDB records written by Gemmi are formatted in the same way as in the wwPDB. This makes possible to use `diff` to compare a PDB file from wwPDB and a file converted by Gemmi from mmCIF. The file from wwPDB will have more records, but the diff should still be readable.

The option `--expand-ncs` expands strict NCS, defined in the `MTRIX` record (PDB) or in the `_struct_ncs_oper` table (mmCIF). It is not obvious how to name the new chains that are added. We have two options: either new names are generated (`=new`) or the chain names are not changed but distinct segment IDs are added (`=dup`).

1.8.6 tags

Lists tags from one or multiple CIF files together with some statistics.

```
$ gemmi tags -h
Usage:
 gemmi tags [options] FILE_OR_DIR[...]
List CIF tags with counts of blocks and values.
  -h, --help     Print usage and exit.
```

(continues on next page)

(continued from previous page)

```
  -V, --version  Print version and exit.
  --count-files  Count files instead of blocks.
  --glob=GLOB    Process files matching glob pattern.

Options for making https://project-gemmi.github.io/pdb-stats/tags.html
  --full         Gather data for tags.html
  --entries-idx  Use entries.idx to read more recent entries first.
  --sf           (for use with --entries-idx) Read SF mmCIF files.
```

By default, the tag statistics show in how many blocks the tag is present, and the total number of non-null values for the tag:

```
$ gemmi tags components.cif.gz
tag   block-count     value-count
_chem_comp.formula    29748   29748
_chem_comp.formula_weight     29749   29749
...
_pdbx_chem_comp_identifier.type       29338   52899
Tag count: 67
Block count: 29749
File count: 1
```

This program is run with option `--full` on the whole PDB archive to produce data for pdb-stats/tags.html.

1.8.7 map

Shows a summary of a CCP4 map file, optionally performing simple transformations.

```
$ gemmi map -h
Usage:
 gemmi map [options] CCP4_MAP[...]

  -h, --help         Print usage and exit.
  -V, --version      Print version and exit.
  -v, --verbose      Verbose output.
  --deltas           Statistics of dx, dy and dz.
  --check-symmetry   Compare the values of symmetric points.
  --write-xyz=FILE   Write transposed map with fast X axis and slow Z.
  --write-full=FILE  Write map extended to cover whole unit cell.
```

1.8.8 mask

Makes a mask in the CCP4 format. It has two functions:

- masking atoms if the input file is a coordinate file,
- using a threshold to convert a CCP4 map file to a mask file.

```
$ gemmi mask -h
Usage:
 gemmi mask [options] INPUT output.msk

Makes a mask in the CCP4 format.
If INPUT is a CCP4 map the mask is created by thresholding the map.
```

(continues on next page)

(continued from previous page)

```
If INPUT is a coordinate file (mmCIF, PDB, etc) the atoms are masked.
  -h, --help           Print usage and exit.
  -V, --version        Print version and exit.
  -v, --verbose        Verbose output.
  --from=coor|map      Input type (default: from file extension).

Options for making a mask from a map:
  -t, --threshold      The density cutoff value.
  -f, --fraction       The volume fraction to be above the threshold.

Options for masking a model:
  -s, --spacing=D      Max. sampling for the grid (default: 1A).
  -g, --grid=NX,NY,NZ  Grid sampling.
  -r, --radius         Radius of atom spheres (default: 3.0A).
```

1.8.9 mtz

```
$ gemmi mtz -h
Usage:
 gemmi mtz [options] MTZ_FILE[...]
Print informations from an mtz file.
  -h, --help       Print usage and exit.
  -V, --version    Print version and exit.
  -v, --verbose    Verbose output.
  -H, --headers    Print raw headers, until the END record.
  -d, --dump       Print a subset of CCP4 mtzdmp informations.
  -B N, --batch=N  Print data from batch header N.
  -b, --batches    Print data from all batch headers.
  --tsv            Print all the data as tab-separated values.
  --stats          Print column statistics (completeness, mean, etc).
  --check-asu      Check if reflections are in conventional ASU.
  --toggle-endian  Toggle assumed endiannes (little <-> big).
  --no-isym        Do not apply symmetry from M/ISYM column.
```

1.8.10 mtz2cif

Converts reflection data from MTZ to mmCIF.

```
$ gemmi mtz2cif -h
Usage:
  gemmi mtz2cif [options] MTZ_FILE CIF_FILE
Options:
  -h, --help             Print usage and exit.
  -V, --version          Print version and exit.
  -v, --verbose          Verbose output.
  --spec=FILE            Column and format specification.
  --print-spec           Print default spec and exit.
  -b NAME, --block=NAME  mmCIF block name: data_NAME (default: mtz).
  --skip-empty           Skip reflections with no values.
  --no-comments          Do not write comments in the mmCIF file.
  --wavelength=LAMBDA    Set wavelengths (default: from input file).

If CIF_FILE is -, the output is printed to stdout.
```

(continues on next page)

(continued from previous page)

```
If spec is -, it is read from stdin.

Lines in the spec file have format:
  [FLAG] COLUMN TYPE TAG [FORMAT]
for example:
  SIGF_native * SIGF_meas_au 12.5e
  FREE I pdbx_r_free_flag 3.0f
FLAG (optional) is either ? or &:
  ? = ignored if no column in the MTZ file has this name.
  & = ignored if the previous line was ignored.
  Example:
      ? I    J intensity_meas
      & SIGI Q intensity_sigma
COLUMN is MTZ column label. Columns H K L are added if not specified.
  Alternative labels can be separated with | (e.g. FREE|FreeR_flag).
TYPE is used for checking the columm type, unless it is '*'.
TAG does not include category name, it is only the part after _refln.
FORMAT (optional) is printf-like floating-point format:
 - one of e, f, g with optional flag, width and precision
 - flag is one of + - # _; '_' stands for ' ', for example '_.4f'
 - since all numbers in MTZ are stored as float, the integer columns use
   the same format as float. The format of _refln.status is ignored.
```

1.8.11 cif2mtz

Converts reflection data from mmCIF to MTZ.

```
$ gemmi cif2mtz -h
Usage:
  gemmi cif2mtz [options] CIF_FILE MTZ_FILE
  gemmi cif2mtz [options] CIF_FILE --dir=DIRECTORY
Options:
  -h, --help               Print usage and exit.
  -V, --version            Print version and exit.
  -v, --verbose            Verbose output.
  -b NAME, --block=NAME    mmCIF block to convert.
  -d DIR, --dir=NAME       Output directory.
  --title                  MTZ title.
  -H LINE, --history=LINE  Add a history line.
  -u, --unmerged           Write unmerged MTZ file(s).

First variant: converts the first block of CIF_FILE, or the block
specified with --block=NAME, to MTZ file with given name.

Second variant: converts each block of CIF_FILE to one MTZ file
(block-name.mtz) in the specified DIRECTORY.

If CIF_FILE is -, the input is read from stdin.
```

1.8.12 sf2map

Transforms map coefficients from either MTZ or SF mmCIF to CCP4 map.

```
$ gemmi sf2map -h
Usage:
  gemmi sf2map [options] INPUT_FILE MAP_FILE

INPUT_FILE must be either MTZ or mmCIF with map coefficients.

By default, the program searches for 2mFo-DFc map coefficients in:
  - MTZ columns FWT/PHWT or 2FOFCWT/PH2FOFCWT,
  - mmCIF tags _refln.pdbx_FWT/pdbx_PHWT.
If option "-d" is given, mFo-DFc map coefficients are searched in:
  - MTZ columns DELFWT/PHDELWT or FOFCWT/PHFOFCWT,
  - mmCIF tags _refln.pdbx_DELFWT/pdbx_DELPHWT.


Options:
  -h, --help           Print usage and exit.
  -V, --version        Print version and exit.
  -v, --verbose        Verbose output.
  -d, --diff           Use difference map coefficients.
  --section=NAME       MTZ dataset name or CIF block name
  -f COLUMN            F column (MTZ label or mmCIF tag).
  -p COLUMN            Phase column (MTZ label or mmCIF tag).
  --weight=COLUMN      (normally not needed) weighting for F.
  -g, --grid=NX,NY,NZ  Grid size (user-specified minimum).
  --exact              Use the exact grid size specified by --grid.
  -s, --sample=NUMBER  Set spacing to d_min/NUMBER (3 is usual).
  --zyx                Output axes Z Y X as fast, medium, slow (default is X Y
                       Z).
  -G                   Print size of the grid that would be used and exit.
  --timing             Print calculation times.
  --normalize          Scale the map to standard deviation 1 and mean 0.
```

The `--sample` option is named after the `GRID SAMPLE` keyword of the venerable CCP4 FFT program; its value has the same meaning.

1.8.13 map2sf

Transforms CCP4 map into map coefficients.

```
$ gemmi map2sf -h
Usage:
  gemmi map2sf [options] MAP_FILE OUTPUT_FILE COL_F COL_PH

Writes map coefficients (amplitude and phase) of a map to OUTPUT_FILE.
The output is MTZ if it has mtz extension, otherwise it is mmCIF.

Options:
  -h, --help       Print usage and exit.
  -V, --version    Print version and exit.
  -v, --verbose    Verbose output.
  -b, --base=PATH  Add new columns to the data from this file.
  --section=NAME   Add new columns to this MTZ dataset or CIF block.
  --dmin=D_MIN     Resolution limit.
  --ftype=TYPE     MTZ amplitude column type (default: F).
  --phitype=TYPE   MTZ phase column type (default: P).
```

1.8.14 sfcalc

Calculates structure factors from a model.

```
$ gemmi sfcalc -h
Usage:
  gemmi sfcalc [options] INPUT_FILE

Calculates structure factors of a model (PDB, mmCIF or SMX CIF file).

Uses FFT to calculate all reflections up to requested resolution for MX
files. Otherwise, for SMX and --hkl, F's are calculated directly.
This program can also compare F's calculated directly with values
calculated through FFT or with values read from a reflection file.

Options:
  -h, --help        Print usage and exit.
  -V, --version     Print version and exit.
  -v, --verbose     Verbose output.
  --hkl=H,K,L       Calculate structure factor F_hkl.
  --dmin=NUM        Calculate structure factors up to given resolution.
  --ciffp           Read f' from _atom_type_scat_dispersion_real in CIF.
  --wavelength=NUM  Wavelength [A] for calculation of f' (use --wavelength=0 or
                    -w0 to ignore anomalous scattering).
  --unknown=SYMBOL  Use form factor of SYMBOL for unknown atoms.
  --noaniso         Ignore anisotropic ADPs.

Options for FFT-based calculations:
  --rate=NUM        Shannon rate used for grid spacing (default: 1.5).
  --blur=NUM        B added for Gaussian blurring (default: auto).
  --rcut=Y          Use atomic radius r such that rho(r) < Y (default: 5e-5).
  --test[=CACHE]    Calculate exact values and report differences (slow).

Options for comparing calculated values with values from a file:
  --check=FILE      Re-calculate Fcalc and report differences.
  --f=LABEL         MTZ column label (default: FC) or small molecule cif tag
                    (default: F_calc or F_squared_calc).
  --phi=LABEL       MTZ column label (default: PHIC)
  --scale=S         Multiply calculated F by sqrt(S) (default: 1).
```

In general, structure factors can be calculated

- either directly, by summing contributions from each atom to each reflection,
- or by calculating an electron density on a grid and using discrete Fourier transform.

This program can measure the errors resulting from the latter method (in addition to its main function – calculation of the structure factors). The errors depend on

- the grid spacing – controlled by the oversampling `--rate=R`; the maximum spacing is $d_{\min}/2R$,
- atomic radius – we neglect electron density of the atom beyond this radius; only density contributions above the (absolute) value specified with `--rcut` are taken into account,
- Gaussian dampening (blurring) factor – artificial temperature factor B_{extra} added to all atomic B-factors (the structure factors are later corrected to cancel it out); either specified with `--blur` or picked automatically.

Choosing these parameters is a trade-off between efficiency and accuracy. B_{extra} is the most interesting one. It is discussed in the ITfC vol B, chapter 1.3 by G. Bricogne, section 1.3.4.4.5, and further in papers by J. Navaza (2002) and by P. Afonine and A. Urzhumtsev (2003). Still, I have not found a practical recipe how to pick a good value.

Increasing the dampening makes the computations slower (because it increases atomic radius), while the value of B_{extra} that gives the most accurate results depends on the resolution, oversampling, atomic radius cut-off, and on the distribution of B-factors (normally, only the minimal B-factor in the model is considered).

The option `--test` can be used to see how accuracy and efficiency depends on the choice of parameters. For example, this shell script performs a series of calculations with differing B_{extra}:

```
model=1mru.pdb
dmin=2.5
gemmi sfcalc --dmin=$dmin --test $model >cache.tsv
for i in `seq -20 5 20`; do
    printf -- "$i\t" >&2
    gemmi sfcalc --dmin=$dmin --rate=1.5 --rcut=1e-4 --blur=$i --test=cache.tsv $model
done >/dev/null
```

Running it prints:

```
-20   RMSE=0.93304  0.5495%  max|dF|=38.80  R=0.301%   0.27671s
-15   RMSE=0.37007  0.2179%  max|dF|=41.26  R=0.094%   0.28366s
-10   RMSE=0.27075  0.1595%  max|dF|=44.35  R=0.041%   0.29322s
-5    RMSE=0.27228  0.1604%  max|dF|=47.59  R=0.029%   0.30459s
0     RMSE=0.28903  0.1702%  max|dF|=50.95  R=0.029%   0.31399s
5     RMSE=0.30806  0.1814%  max|dF|=54.35  R=0.032%   0.32527s
10    RMSE=0.32847  0.1934%  max|dF|=57.92  R=0.036%   0.33360s
15    RMSE=0.35028  0.2063%  max|dF|=61.66  R=0.041%   0.34181s
20    RMSE=0.37283  0.2196%  max|dF|=65.44  R=0.047%   0.35380s
```

The error used in RMSE is the magnitude of the difference of two vectors: $|F_{approx} - F_{exact}|$. The next column is RMSE normalized by the sum of $|F_{calc}|$. Then we have maximum error for a single reflection, and the wall time of computations. We can see that in this case negative "dampening" (subtracting about $10A^2$ from all B-factors) improves both accuracy and performance.

1.8.15 fprime

Calculate anomalous scattering factors (*f'* and *f"*). Uses Cromer-Libermann algorithm with corrections from Kissel and Pratt. This and different approaches are discussed in the documentation of the *underlying functions*.

```
$ gemmi fprime -h
Usage:
 gemmi fprime [options] ELEMENT[...]
Prints anomalous scattering factors f' and f".

Options:
  -h, --help               Print usage and exit.
  -V, --version            Print version and exit.
  -e, --energy=ENERGY      Energy [eV]
  -w, --wavelength=LAMBDA  Wavelength [A]
```

Here is an example how to print *f'* and *f"* using gemmi, XrayDB, CCP4 crossec and cctbx (pyFprime is not included because it is a GUI-only program). The Chantler's data from XrayDB is probably the most reliable one:

```
$ gemmi fprime --wavelength=1.2 Se
Element  E[eV]  Wavelength[A]      f'             f"
Se      10332.0  1.2             -1.4186        0.72389

$ python3 -c "import xraydb; print(xraydb.f1_chantler('Se', 10332.0), xraydb.f2_
↪chantler('Se', 10332.0))"
```

(continues on next page)

(continued from previous page)

```
-1.4202028957329489 0.7100533627953146

$ echo -e "atom SE\n cwav 1 1.2 0\n END" | crossec | grep ^SE
SE          1.2000    -1.5173     0.7240

$ cctbx.eltbx.show_fp_fdp --wavelength=1.2 --elements=Se
Wavelength: 1.2 Angstrom

Element: Se
  Henke et al.  : f'=-1.44568 , f''=0.757958
  Sasaki et al. : f'=-1.5104  , f''=0.724000
  diff f''=-2.29 %
```

1.8.16 reindex

Reindex reflections in MTZ file.

```
$ gemmi reindex -h
Usage:
  gemmi reindex [options] INPUT_MTZ OUTPUT_MTZ
Options:
  -h, --help     Print usage and exit.
  -V, --version  Print version and exit.
  -v, --verbose  Verbose output.
  --hkl=OP       Reindexing transform as triplet (e.g. k,h,-l).
  --no-history   Do not add 'Reindexed with...' line to mtz HISTORY

Input file can be gzipped.
```

1.8.17 residues

List residues from a coordinate file, one per line.

```
$ gemmi residues -h
Usage:
 gemmi residues [options] INPUT[...]
Prints one residue per line, with atom names.
  -h, --help          Print usage and exit.
  -V, --version       Print version and exit.
  --format=FORMAT     Input format (default: from the file extension).
  -mSEL, --match=SEL  Print residues/atoms matching the selection.
  -l, --label         Print 'label' chain ID and seq ID in brackets.
INPUT is a coordinate file (mmCIF, PDB, etc).
The selection SEL has MMDB syntax:
/mdl/chn/s1.i1(res)-s2.i2/at[el]:aloc (all fields are optional)
```

Example:

```
$ gemmi residues -m '/3/*/(CYS,CSD)' 4pth.pdb
Model 3
A   85  CYS: N CA C O CB SG H HA HB2 HB3 HG
A  152  CSD: N CA CB SG C O OD1 OD2 HA HB2 HB3
```

1.8.18 align

Sequence alignment (global, pairwise, affine gap penalty). Used primarily for aligning the residues in the model's chains to the full sequence from the SEQRES record.

```
$ gemmi align -h
Pairwise sequence alignment with scoring matrix and affine gap penalty.

Usage:

gemmi align [options] FILE[...]
    Aligns sequence from the model to the full sequence (SEQRES).
    Both are from the same FILE - either in the PDB or mmCIF format.
    If the mmCIF format is used, option --check-mmcif can be used.

gemmi align [options] --query=CHAIN1 --target=CHAIN2 FILE1 FILE2
    Aligns CHAIN1 from FILE1 to CHAIN2 from FILE2.
    By default, the sequence of residues in the model is used.
    To use SEQRES prepend '+' to the chain name (e.g. --query=+A).

gemmi align [options] --text-align STRING1 STRING2
    Aligns two ASCII strings (used for testing).

Options:
  -h, --help         Print usage and exit.
  -V, --version      Print version and exit.
  --check-mmcif      checks alignment agains _atom_site.label_seq_id
  --query=[+]CHAIN   Align CHAIN from file INPUT1.
  --target=[+]CHAIN  Align CHAIN from file INPUT2.
  --text-align       Align characters in two strings (for testing).

Scoring (absolute values):
  --match=INT        Match score (default: 1).
  --mism=INT         Mismatch penalty (default: -1).
  --gapo=INT         Gap opening penalty (default: -1).
  --gape=INT         Gap extension penalty (default: -1).

Output options:
  -p                 Print formatted alignment with one-letter codes.
  -v, --verbose      Verbose output.
```

For the testing purpose, it can align text strings. For example, the Levenshtein distance can be calculated by setting the gap opening penalty to zero:

```
$ gemmi align -p --match=0 --gapo=0 --text-align Saturday Sunday
Score: -3   CIGAR: 1M2I5M
Saturday
|  |.|||
S--unday
```

This tool uses modified code from ksw2. See the *Sequence alignment* section for more details.

1.8.19 sg

Prints information about given space group.

```
$ gemmi sg -h
Usage:
 gemmi sg [options] SPACEGROUP[...]
Prints information about the space group.
  -h, --help     Print usage and exit.
  -V, --version  Print version and exit.
  -v, --verbose  Verbose output.
  --asu=N        Draw ASU in NxNxN map grid and exit. Uses N(N+1) columns.
```

1.8.20 contents

Analyses and summarizes content of a coordinate file. Inspired by CCP4 program `rwcontents`.

By default, it prints atom count, estimated number of hydrogens in the protein, molecular weight of the protein, ASU volume, Matthews coefficient and the fractional volume of solvent in the crystal.

It has options to print other information – see the help message below.

```
$ gemmi contents -h
Usage:
 gemmi contents [options] INPUT[...]
Analyses content of a PDB or mmCIF.
  -h, --help     Print usage and exit.
  -V, --version  Print version and exit.
  -v, --verbose  Verbose output.
  -b             Print statistics of isotropic ADPs (B-factors).
  --dihedrals    Print peptide dihedral angles.
  -n             Do not print content (for use with other options).
```

1.8.21 contact

Searches for contacts in a model.

```
$ gemmi contact -h
Usage:
 gemmi contact [options] INPUT[...]
Searches for contacts in a model (PDB or mmCIF).
  -h, --help     Print usage and exit.
  -V, --version  Print version and exit.
  -v, --verbose  Verbose output.
  -d, --maxdist=D  Maximal distance in A (default 3.0)
  --cov=TOL      Use max distance = covalent radii sum + TOL [A].
  --covmult=M    Use max distance = M * covalent radii sum + TOL [A].
  --minocc=MIN   Ignore atoms with occupancy < MIN.
  --ignore=N     Ignores atom pairs from the same: 0=none, 1=residue, 2=same or
                 adjacent residue, 3=chain, 4=asu.
  --nosym        Ignore contacts between symmetry mates.
  --assembly=ID  Output bioassembly with given ID (1, 2, ...).
  --noh          Ignore hydrogen (and deuterium) atoms.
  --nowater      Ignore water.
  --noligand     Ignore ligands and water.
  --count        Print only a count of atom pairs.
  --twice        Print each atom pair A-B twice (A-B and B-A).
```

1.8.22 blobs

Searches for unmodelled blobs in electron density. Similar to "Validate > Unmodelled blobs..." in Coot. For use in Dimple.

```
$ gemmi blobs -h
Usage:
 gemmi blobs [options] MTZ_OR_MMCIF PDB_OR_MMCIF

Search for umodelled blobs of electron density.

Options:
  -h, --help            Print usage and exit.
  -V, --version         Print version and exit.
  -v, --verbose         Verbose output.

The area around model is masked to search only unmodelled density.
  --mask-radius=NUMBER  Mask radius (default: 2.0 A).
  --mask-water          Mask water (water is not masked by default).

Searching blobs of density above:
  --sigma=NUMBER        Sigma (RMSD) level (default: 1.0).
  --abs=NUMBER          Absolute level in electrons/A^3.

Blob criteria:
  --min-volume=NUMBER   Minimal volume (default: 10.0 A^3).
  --min-score=NUMBER    Min. this electrons in blob (default: 15.0).
  --min-sigma=NUMBER    Min. peak rmsd (default: 0.0).
  --min-peak=NUMBER     Min. peak density (default: 0.0 el/A^3).

Options for map calculation:
  -d, --diff            Use difference map coefficients.
  --section=NAME        MTZ dataset name or CIF block name
  -f COLUMN             F column (MTZ label or mmCIF tag).
  -p COLUMN             Phase column (MTZ label or mmCIF tag).
  --weight=COLUMN       (normally not needed) weighting for F.
  -g, --grid=NX,NY,NZ   Grid size (user-specified minimum).
  --exact               Use the exact grid size specified by --grid.
  -s, --sample=NUMBER   Set spacing to d_min/NUMBER (3 is usual).
  -G                    Print size of the grid that would be used and exit.
  --timing              Print calculation times.
```

1.8.23 h

Adds or removes hydrogens. Hydrogen are put in positions based only on restraints from a monomer library.

```
$ gemmi h -h
Usage:
 gemmi h [options] INPUT_FILE OUTPUT_FILE

Add hydrogens in positions specified by the monomer library.
By default, it removes and re-adds all hydrogens.
By default, hydrogens are not added to water.

Options:
  -h, --help      Print usage and exit.
```

(continues on next page)

(continued from previous page)

```
  -V, --version   Print version and exit.
  -v, --verbose   Verbose output.
  --monomers=DIR  Monomer library dir (default: $CLIBD_MON).
  --remove        Only remove hydrogens.
  --keep          Do not add/remove hydrogens, only change positions.
  --water         Add hydrogens also to waters.
  --sort          Order atoms in residues according to _chem_comp_atom.
```

1.8.24 mondiff

Compares restraints from two monomer CIF files. It is intended for comparing restraints for the same monomer, but generated with different programs (or different versions of the same program).

The files should have format used by the CCP4/Refmac monomer library. This format is supported by all major macromolecular refinement programs.

```
$ gemmi mondiff -h
Usage:
  gemmi mondiff [options] FILE1 FILE2
Options:
  -h, --help         Print usage and exit.
  -V, --version      Print version and exit.
  -v, --verbose      Verbose output.

Minimal reported differences:
  --bond=DELTA       difference in distance value (default: 0.01).
  --bond-esd=DELTA   difference in distance esd (default: 0.1).
  --angle=DELTA      difference in angle value (default: 0.1).
  --angle-esd=DELTA  difference in angle esd (default: 1.0).
  --rel=SIGMA         abs(value difference) / esd > SIGMA (default: 0.0).
```

1.8.25 wcn

Calculates Weighted Contact Number (WCN) and a few other similar metrics.

WCN can be used to predicts B-factors (ADPs) from coordinates, and to compare this prediction with the values from refinement.

Background

Protein flexibility and dynamic properties can be to some degree inferred from the atomic coordinates of the structure. Various approaches are used in the literature: molecular dynamics, Gaussian or elastic network models, normal mode analysis, calculation of solvent accessibility or local packing density, and so on.

Here we apply the simplest approach, which is pretty effective. It originates from the 2002 PNAS paper in which Bertil Halle concluded that B-factors are more accurately predicted by counting nearby atoms than by Gaussian network models. This claim was based on the analysis of only 38 high resolution structures (and a neat theory), but later on the method was validated on many other structures.

In particular, in 2007 Manfred Weiss brought this method to the attention of crystallographers by analysing in Acta Cryst D different variants of the methods on a wider set of more representative crystals. Recently, the parameters fine-tuned by Weiss have been used for guessing which high B-factors (high comparing with the predicted value) result from the radiation damage.

Only a few months later, in 2008, Chih-Peng Lin et al. devised a simple yet significant improvement to the original Halle's method: weighting the counted atoms by $1/d^2$, the inverse of squared distance. It nicely counters the increasing average number of atoms with the distance ($\sim d^2$). This method was named WCN – weighted contact number (hmm.. "contact").

These two methods are so simple that it seems easy to find a better one. But according to my quick literature search, no better method of this kind has been found yet. In 2009 Li and Bruschweiler proposed weighting that decreases exponentially (that model was named LCBM), but in my hands it does not give better results than WCN.

In 2016 Shahmoradi and Wilke did a data analysis aiming to disentangle the effects of local and longer-range packing in the above methods. They were not concerned with B-factors, though, but with the rate of protein sequence evolution. Because the "contact" methods predict many things. Interestingly, if the exponent in WCN is treated as a parameter (equal -2 in the canonical version), the value -2.3 gives the best results in predicting evolution.

TLS

We also need to note that TLS-like methods that model B-factors as rigid-body motion of molecules are reported to give better correlation with experimental B-factors than other methods. But because such models use experimental B-factors on the input and employ more parameters, they are not directly comparable with WCN.

Unlike the TLS that is routinely used in the refinement of diffraction data, the TLS modelling described here is isotropic. It uses 10 parameters (anisotropic TLS requires 20) as described in a paper by Kuriyan and Weis (1991). Soheilifard et al (2008) got even better results by increasing B-factors at the protein ends, using 13 parameters all together. This model was named eTLS (e = extended).

The high effectiveness of the TLS model does not mean that B-factors are dominated by the rigid-body motion. As noted by Kuriyan and Weis, the TLS model captures also the fact that atoms in the interior of a protein molecule generally have smaller displacements than those on the exterior. Additionally, authors of the LCBM paper find that the TLS model fitted to only half of the protein poorly fits the other half, which suggests overfitting.

We may revisit rigid-body modelling in the future, but now we get back to the contact numbers.

Details

The overview above skipped a few details.

- While the WCN method is consistently called WCN, the Halle's method was named LDM (local density model) in the original paper, and is called CN (contact number) in some other papers. CN is memorable when comparing with WCN (which adds 'W' – weighting) and with ACN (which adds 'A' – atomic).
- These method are used either per-atom (for predicting B-factors, etc.) or per-residue (for evolutionary rate, etc.). So having "A" in ACN clarifies how it is used. To calculate the contact number per-residue one needs to pick a reference point in the residue (Cβ, the center of mass or something else), but here we do only per-atom calculations.
- The CN method requires a cut-off, and the cut-off values vary widely, from about 5 to 18Å. In the original paper it was 7.35Å, Weiss got 7.0Å as the optimal value, Shahmoradi 14.3Å.
- The CN can be seen as weighted by Heaviside step function, and smoothing it helps a little bit (as reported by both Halle and Weiss).
- Similarly to eTLS, the LCBM method has eLCBM variant that adds "end effects" – special treatment of the termini.
- Finally, these methods may or may not consider the symmetry mates in the crystal. Halle checked that including symmetric images improves the prediction. Weiss (ACN) and Li and Bruschweiler (LCBM) are also taking symmetry into account. But I think other papers don't.

Metrics for comparison

To compare a number of nearby atoms with B-factor we either rescale the former, or we use a metric that does not require rescaling. The Pearson correlation coefficient (CC) is invariant under linear transformation, so it can be calculated directly unless we would like to apply non-linear scaling. Which was tried only in the Manfred Weiss' paper: scaling function with three parameters improved CC by 0.012 comparing with linear function (that has two parameters). Here, to keep it simple, we only do linear scaling.

As noted by Halle, Pearson CC as well as mean-square deviation can be dominated by a few outliers. Therefore Halle used relative mean absolute deviation (RMAD): sum of absolute differences divided by the average absolute deviation in the experimantal values. Halle justifies this normalization writing that it allows to compare structures determined at different temperatures. This is debatable as can be seen from ccp4bb discussions on how to compare B-factors between two structures. But for sure RMAD is a more robust metric, so we also use it. It adds another complication, though. To minimize the absolute deviation we cannot use least-squares fitting, but rather quantile regression with q=0.5.

Another metric is the rank correlation. It is interesting because it is invariant under any monotonic scaling. But it is not guaranteed to be a good measure of similarity.

Results

To be wrapped up and published. But in the meantime here are some thoughts:

- The optimal exponent is slightly larger than 2; the difference is small, so we prefer to use 2 (i.e. w=1/r^2).
- Accounting for all symmetry mates (i.e. for intermolecular contacts in the crystal) improves the results – and then the cut-off is necessary.
- The optimal cut-off is around 15A – let's use 15A.
- Averaging predicted B-factors of nearby atoms helps; we use Gaussian smoothing (blurring) with σ around 2A.
- Pearson CC around 0.8 may seem high, but it corresponds to R2=0.64, i.e. it we explain only 64% of the B-factor variance. Even less of the absolute deviation – below 50%.
- Minimizing absolute deviation (with quantile regression) gives similar results as the ordinary least squares (OLS). The difference in terms of RMAS is only ~0.03.
- Combining WCN with CN is helping only a tiny bit (i.e. both are highly correlated) at the cost of additional parameter that is fitted. Combining WCN with rotation-only model (squared distance from the center of mass) increases CC slightly more, but still not much.
- Accounting for symmetry mates worsens prediction of evolutionary rates. I used data from Shahmoradi and Wilke to check this.

Program

gemmi-wcn implements combination of the CN and WCN methods above.

Being based on a general-purpose crystallographic library it handles corner cases that are often ignored. A good example is searching for contacts. For most of the structures, considering only the same and neighbouring unit cells (1+26) is enough. But some structures have contacts between molecules several unit cells apart, even with only *a single chain in the asu*.

TBC

```
$ gemmi wcn -h
Usage:
 gemmi wcn [options] INPUT[...]
Calculation of local density / contact numbers: WCN, CN, ACN, LDM, etc.
  -h, --help       Print usage and exit.
  -V, --version    Print version and exit.
  -v, --verbose    Verbose output.
  -f, --file=FILE  Obtain paths or PDB IDs from FILE, one per line.
  -l, --list       List per-residue values.
  --min-dist=DIST  Minimum distance for "contacts" (default: 0.8).
  --cutoff=DIST    Maximum distance for "contacts" (default: 15).
  --pow=P          Exponent in the weighting (default: 2).
  --blur=SIGMA     Apply Gaussian smoothing of predicted B-factors.
  --rom            Rotation only model: |pos-ctr_of_chain|^P instead of WCN.
  --chain=CHAIN    Use only one chain from the INPUT file.
  --sanity         Run sanity checks first.
  --sidechains=X   One of: include, exclude, only (default: include).
  --no-crystal     Ignore crystal symmetry and intermolecular contacts.
  --omit-ends=N    Ignore N terminal residues from each chain end.
  --print-res      Print also resolution and R-free.
  --xy-out=DIR     Write DIR/name.xy files with WCN and B(exper).
```