Validation of Correction of Macromolecular Structural Models

William Rochira

MSc by Research

University of York

Chemistry

December 2020

Abstract

Recent advances in automation in the field of computational structural biology have created a void to be filled by novel validation software. In this project, the problems and inadequacies of currently available validation tools are identified, and the requirements of a novel validation tool are both ascertained and addressed. The development of a new validation software package is described in detail, starting with the development of the front-end interface and the back-end calculations, followed by the integration of these two components to produce an all-in-one validation package, which can calculate its own comprehensive per-residue validation metrics and present them in a compact, interactive, graphical interface, so as to allow the intuitive and thorough analysis of a protein model's quality that is understandable at a glance. This interface features a novel graphical representation of validation, which plots multiple validation metrics along concentric axes such that correlations between those metrics are immediately apparent, and poorly-modelled regions are emphasised to the user. The software can be run standalone, or plugged into new or existing validation pipelines, and can incorporate calculated metrics from other validation services such as MolProbity (1). It supports multi-model comparison in its single view, and runs with negligible time penalty, making it especially suitable for evaluating incremental changes that result from automated or manual iterative model building. To showcase its extensibility and pluggable design, the integration of this package into the existing CCP4i2 (2) software suite is described. Finally, the package is analysed both quantitatively and qualitatively, and potential avenues for future work are outlined.

List of Contents

Abstract 2

List of Contents 3

Acknowledgements 6

Author's Declaration 7

1 Introduction 8

1.1 Proteins 8

1.1.1 Protein structure 8

1.1.1.1 Primary structure 8

1.1.1.2 Secondary structure 8

1.1.1.3 Tertiary structure 9

1.1.1.4 Quaternary structure 9

1.1.1.5 Dynamics 9

1.1.2 Protein function 10

1.2 Protein structure determination 11

1.2.1 Background 11

1.2.2 Experimental methods 12

1.2.2.1 X-ray crystallography 12

1.2.2.2 Cryogenic electron microscopy 13

1.2.2.3 Nuclear magnetic resonance 14

1.2.2.4 Other methods 15

1.2.2.5 Summary 15

1.2.2 Protein structure solution pipeline 16

1.2.3 Protein structure validation 16

1.3 Project goals 21

2 Methods 23

2.1 Interface 23

2.1.1 The interface format 23

2.1.2 The graphics 25

2.1.3 The report 37

2.2 Metrics 45

2.2.1 Preparation 45

2.2.2 Outline 45

2.2.3 Initial framework 46

2.2.4 Model-only metrics 47
2.2.4.1 B-factors 48
2.2.4.2 Bond geometry 48
2.2.4.3 Ramachandran conformation 51
2.2.4.4 Rotamer conformation 52
2.2.4.5 Others 54
2.2.5 Density fit analyses 55
2.2.5.1 Background 55
2.2.5.2 Existing implementations 57
2.2.5.3 Building a scoring method 58
2.2.5.4 Implementation 59
2.2.5.5 Complications 60
2.2.6 Percentiles library 62
2.3 Combining the two modules 69
2.3.1 Initial adjustments 69
2.3.2 Testing with real-world data 70
2.3.3 Chain view changes 71
2.3.3.1 Metric polarity synchronisation 73
2.3.3.2 Sequence alignment 75
2.3.3.3 Discrete metric implementation 76
2.3.3.4 Missing-residue shading 82
2.3.3.5 Animation 83
2.3.3.6 Plot formula revision 84
2.3.4 Residue view changes 87
2.3.4.1 New design 87
2.3.4.2 Distribution indicators 91
2.3.5 Report changes 92
2.4 Optimisations 95
2.5 Implementation in CCP4i2 96
2.5.1 Introduction 96
2.5.2 Existing validation task 97
2.5.3 Task redesign plan 99
2.5.4 Initial development 100
2.5.5 Multithreading 103

2.5.6 MolProbity integration 104
3 Results and discussion 106
3.1 Metric accuracy 106
3.1.1 Ramachandran likelihood score 106
3.1.2 Rotamer likelihood score 107
3.1.3 Density fit score 108
3.2 Timing 109
3.3 Interface 112
3.4 CCP4i2 implementation 113
4 Conclusions and future work 117
5 Bibliography 120

Acknowledgements

Many thanks to my supervisors during this project. In particular to Dr. Jon Agirre, who always made himself available, and helped me out throughout the entirety of the project, not just concerning technical questions, to which he would consistently provide constructive insight, but also with valuable personal direction in regards to my future career. Thanks to everyone at YSBL for help whenever I asked for it, and for welcoming me into a lovely community and working environment.

Author's Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, university. All sources are acknowledged as references.

Parts of this thesis have been published by the author:

Rochira W, Agirre J. Iris: Interactive all-in-one graphical validation of 3D protein model iterations. Protein Sci. 2020 Oct 19;67:386.

1 Introduction

1.1 Proteins

Proteins are macromolecular biomolecules composed of polymers of amino acids (polypeptide chains) which are ubiquitous in all domains of life. They are crucial to virtually all biological processes, performing functions such as reaction catalysis, cell signalling, and providing structural support. Individual proteins' constituent amino acid sequences are encoded in genetic material, to be translated by the cell into a polypeptide chain. Despite the one-dimensional nature of this information, protein synthesis leads to the production of incredibly consistent, complex three-dimensional structures, which arise due to intramolecular interactions between the side-chains of amino acid residues in a chain during the process of 'folding'. The median length of a protein in a eukaryotic organism is 472 amino acid residues (3), which corresponds to a molecular mass of roughly 52 kilodaltons.

1.1.1 Protein structure

Protein structure is often broken down into the four-tiered hierarchy described in the following subsections.

1.1.1.1 Primary structure

The primary structure is the one-dimensional sequence of amino acids in each chain in the protein, joined by covalent peptide (amide) bonds. There are twenty different proteinogenic amino acids encoded by the standard genetic code, with many other modified amino acids produced by certain modified translation mechanisms (4).

1.1.1.2 Secondary structure

The secondary structure consists of the local three-dimensional conformations of regular motifs caused by intramolecular hydrogen bonding between the amino hydrogen and carboxyl oxygen

atoms in the main-chain of amino acid residues. By far the most common types of secondary structure are α-helices and β-sheets (composed of β-strands), but there are many other (albeit rarer) types, including turns and bends. On average, 60% of the residues in any given folded protein are found in regular α-helices or β-sheets (5). Areas of a chain that do not form a recognised motif are termed 'random coils'.

1.1.1.3 Tertiary structure

The tertiary structure is the overall three-dimensional conformation of a single polypeptide chain, held together by a number of intramolecular interactions. The four principal interactions involved in tertiary structure formation are: hydrophobic and hydrophilic interactions, disulfide bridges, hydrogen bonds, and ionic bonds. In some cases, coordination of metal ions can be critical to the tertiary structure.

1.1.1.4 Quaternary structure

The quaternary structure is the arrangement and interactions between the component subunits (individual chains) within a multi-subunit complex. The quaternary structure of some complexes will also include cofactors or other biomolecules, such as nucleic acids.

1.1.1.5 Dynamics

Even once fully folded, a protein's structure is not necessarily fixed; protein structures can be dynamic. A change in structure can be triggered by a number of events, including a change in the electrochemical environment, allosteric regulation by an effector molecule, or simply occur as a function of a particular structure's inherent flexibility.

1.1.2 Protein function

A protein's aptitude for its function is dictated by its structure. There are many different classes of proteins, and the scale at which structure impacts function is different for each.

At the most intricate level, a protein's function may depend on the atomic-level position and conformation of individual amino acids. This is especially true in the case of specific binding, a process ubiquitous throughout the protein classes. Examples include signalling proteins and receptors (protein-protein specificity), ribosomes (protein-nucleic acid specificity), and transport proteins (protein-ligand specificity). Specific binding is particularly important for enzymes, which are proteins that catalyse biochemical reactions by converting one or more substrate molecules into different product molecules in a small pocket of the protein termed the active site. Like the other classes of proteins mentioned, some residues of the active site are involved in the specific binding and orientation of substrate molecules. However, enzymes take the atomic-level function a step further, in that some individual amino acid residues actively participate in the chemical reaction during catalysis. The position and orientation of these residues is therefore especially critical to the enzyme's functionality.

On a slightly broader scale are functions that occur over an area of many amino acid residues, typically as a result of those residues sharing some general property. For example, areas of hydrophobic residues can be used as a surface for hydrophobic interactions, either with a hydrophobic face on another protein, or, for instance, for positioning and orienting itself within a membrane (6). Additionally, pockets of residues with a shared property can be used to alter the chemical properties of the side-chains within it. For example, by surrounding a residue with a number of non-polar side-chains, its uncharged state becomes more thermodynamically favoured, affecting its pK_a. This is termed the microenvironment effect, and is instrumental in creating active sites that have the required chemical properties in many enzymes (7).

Over the very widest scale are functions that depend on the physical properties of the entire molecule as a whole. Such is the case with many structural proteins, which are used to provide structural support to certain structures, through imposing rigidity. One of the best examples of such proteins is collagen, the most abundant mammalian protein. Collagen is a fibrous structural protein found in connective tissue. It has high tensile strength, meaning it can withstand great stretching or pulling forces without breaking. Each collagen molecule is formed of three polypeptide chains, which are wrapped around each other to form a triple helix structure, held

together by interstrand hydrogen bonds. To enable this conformation, every third amino acid in each chain must be a glycine residue, which is the most conformationally flexible amino acid (8).

The full range of protein functions is far greater than this overview might imply; each class of proteins comprises a diverse set of functions and modes of action, and each depends on the arrangement of amino acids in the protein's three-dimensional structure. To summarise: the structure of a protein is critical to its function because it determines the mechanisms by which it can interact with other molecules, whether they be ligands, cofactors, or other proteins.

1.2 Protein structure determination

1.2.1 Background

Since proteins are so ubiquitously involved in biological processes, an intricate level of understanding about their individual structures, and therefore functions, is immensely valuable. Such a level of understanding lays the foundation for the creation of new hypotheses about how proteins can be affected, modified, or controlled. As such, protein structure determination is critical to so many aspects of computational biology, either as individual structures of target proteins, or as entire databases of proteins, such as the Protein Data Bank (PDB) (9).

The field in which an understanding of protein structures has proven especially useful is drug design. Historically, drug development has been performed by trial and error, by screening random compounds. However, the advent of protein structure determination brought about systematic structure-based drug design, in which the structure of a target protein can be analysed to calculate the binding modes and affinities of a huge database of different ligands (prospective drug molecules) using high-throughput approaches. Another example is the field of protein engineering: the process of developing artificial proteins, which can be used to develop custom enzymes to biologically catalyse reactions (10,11), to develop proteins for use as drugs, such as rilonacept (12), or even to develop artificial molecular machines (13).

Protein structure determination primarily involves building an atomic model to fit some experimentally observed data. In the following subsection, the most common experimental methods are outlined.

1.2.2 Experimental methods

1.2.2.1 X-ray crystallography

X-ray crystallography is an experimental technique used to determine the arrangements of atoms within a crystal, developed after the discovery of X-rays (14) and their diffractive interactions with crystals (15) in 1895 and 1912 respectively. X-ray crystallography is performed by firing a beam of electrons through a crystal and recording the diffracted radiation to yield a diffraction pattern: a cross-sectional image of the reciprocal space which contains information about the atomic arrangement within the crystal (*Figure 1*). If many such images are taken with the sample placed at various orientations, the diffraction patterns can be decoded to a real space electron density map by applying a Fourier transform, and the atomic structure of the crystal can be inferred. X-ray crystallography on proteins is commonly referred to as protein crystallography, or macromolecular crystallography (MX).

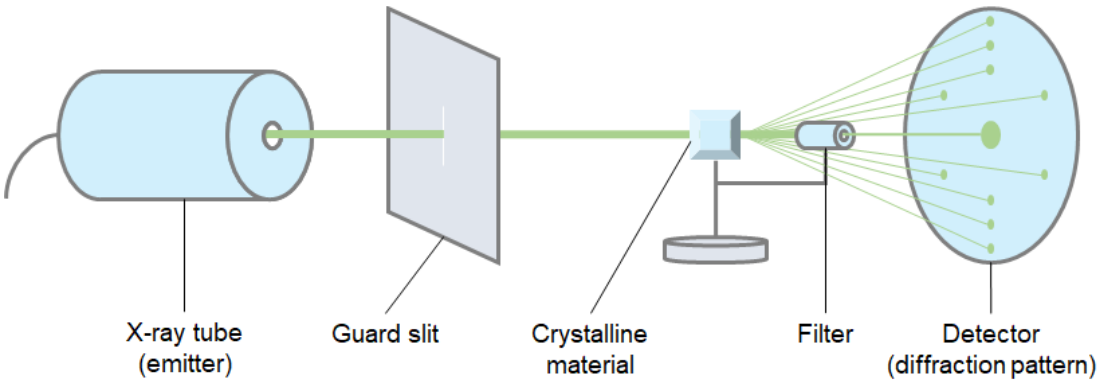

Figure 1: Simplified diagram of an X-ray diffraction experiment.

In 1934, John Desmond Bernal and Dorothy Crowfoot Hodgkin discovered that consistent and reliable diffraction patterns of protein crystals could be obtained if they are X-rayed in their mother liquor (the solution that remains after crystallisation). Until that time, X-ray crystallography of proteins had only been attempted on dry protein crystals, which had given only poor results. With their discovery, Bernal and Hodgkin took the first X-ray photographs of protein crystals, in the form of crystalline pepsin, thereby yielding the first protein diffraction pattern (16). After a couple of decades, the structures of larger proteins started to be solved, the first of which was the structure of sperm whale myoglobin by Sir John Cowdery Kendrew in 1958 (17). Once the potential applications of this technique were realised, the field of structural molecular biology was born.

In the early days of MX, X-rays were produced by in-lab X-ray sources, and diffraction patterns had to be interpreted manually, by performing Fourier transform analysis by laborious manual calculation. Francis Crick, for example, was an experienced crystallographer (18), and MX was famously used by Rosalind Franklin, James Watson, and Francis Crick to solve the structure of DNA in 1953 (19).

MX remains by far the most common method for protein structure determination. In recent years, the process has become substantially more automated, and computational techniques have become integral to the solution process. Although in-lab X-ray sources are still widely available, most modern-day MX is performed using synchrotron radiation, at a beamline facility such as the Diamond Light Source (20).

1.2.2.2 Cryogenic electron microscopy

The basic principle of electron microscopy (EM) involves using an accelerated beam of electrons to illuminate a sample, exploiting the wave-like characteristics of electrons. An electron beam can have a wavelength many orders of magnitude shorter than that of a photon beam, and as a consequence, can capture images at substantially higher resolutions.

Since its advent, the potential for the application of EM in molecular structural biology was well appreciated. However, its practical application was limited, due to the radiation damage caused to samples by the high energy electrons, and the fact that the microscopy had to be performed under vacuum, leading to evaporation of water in the samples. In the early 1980s, it was discovered that by performing EM under cryogenic conditions (temperatures approaching absolute zero) these damaging effects could be lessened (21,22). This is known as cryogenic electron microscopy (cryo-EM). In 1984, a seminal paper from a group at the European Molecular Biology Laboratory featured images of adenovirus embedded in a vitrified layer of water (23). This paper is widely considered to mark the beginning of modern cryo-EM (24–26). Since then, cryo-EM has continued to become more prevalent in the field (*Figure 2*).

Cryo-EM on proteins is most often conducted as a single-particle technique, whereby the imaged sample contains a dispersion of many instances of the molecules of interest, all at different orientations. Therefore, the result of the experiment is a number of images of the target molecule, which can be processed to produce a three-dimensional map of the molecule.

Unlike MX, cryo-EM does not require that samples be crystalised, and also requires a much smaller amount of sample. Historically, one of the main disadvantages of cryo-EM compared to other techniques has been its comparatively low resolution. However, substantial recent advances in image-processing algorithms and detector hardware, such as the advent of direct electron detectors, have led to the so-called 'resolution revolution', yielding data at much higher resolutions (27,28).

1.2.2.3 Nuclear magnetic resonance

After cryo-EM, the next most common method is nuclear magnetic resonance (NMR) spectroscopy (*Figure 2*), a technique that involves recording a spectrum of excitation frequencies emitted by atomic nuclei in the presence of a strong constant magnetic field when perturbed by a weak oscillating magnetic field. The precise frequency of excitation given off by a particular nucleus is dependent on its chemical environment; hence, a frequency spectrum can be used to identify particular chemical environments within a sample compound and build up a picture of the compound's overall structure.

NMR is performed on highly pure aqueous solutions of sample, and hence is only suited to water-soluble proteins. Proteins that have majoritarily hydrophobic externals, such as membrane proteins, are not suited to regular NMR analysis. However, there is a variant of NMR, known as solid-state NMR (ssNMR) which is specifically suited to such cases (29–31).

Since proteins can be such large molecules, a single NMR spectrum will inevitably contain overlaps, within which multiple individual nuclei emit the same excitation frequency, making the spectrum impossible to decipher. Hence, protein NMR normally consists of a multi-dimensional approach, in which peaks from NMR spectra of different nuclei are correlated to produce more informative data.

In protein NMR, structure calculations are performed by applying restraints to the output of multidimensional NMR experiments, in order to obtain a model. In contrast to MX and cryo-EM, protein NMR does not generate an electron density map. Nowadays, protein NMR is more commonly used to investigate protein interactions (32–35).

1.2.2.4 Other methods

There are other, less common, methods applied in macromolecular structure determination. For example, there are purely computational methods such as protein structure prediction, in which a protein's secondary or tertiary structure can be *predicted* from its amino acid sequence if it is known. Such methods were first introduced in the 1960s (36,37), and recent developments in the applications of deep learning have led to impressive breakthroughs in the field, resulting in systems that can predict the full three-dimensional structures of proteins with accuracy approaching that of experimental methods, such as DeepMind's AlphaFold (38,39).

1.2.2.5 Summary

Even today, MX is still by far the most popular technique, with over 140,000 macromolecular structures having been solved by MX and made publicly available (40) (*Figure 2*).

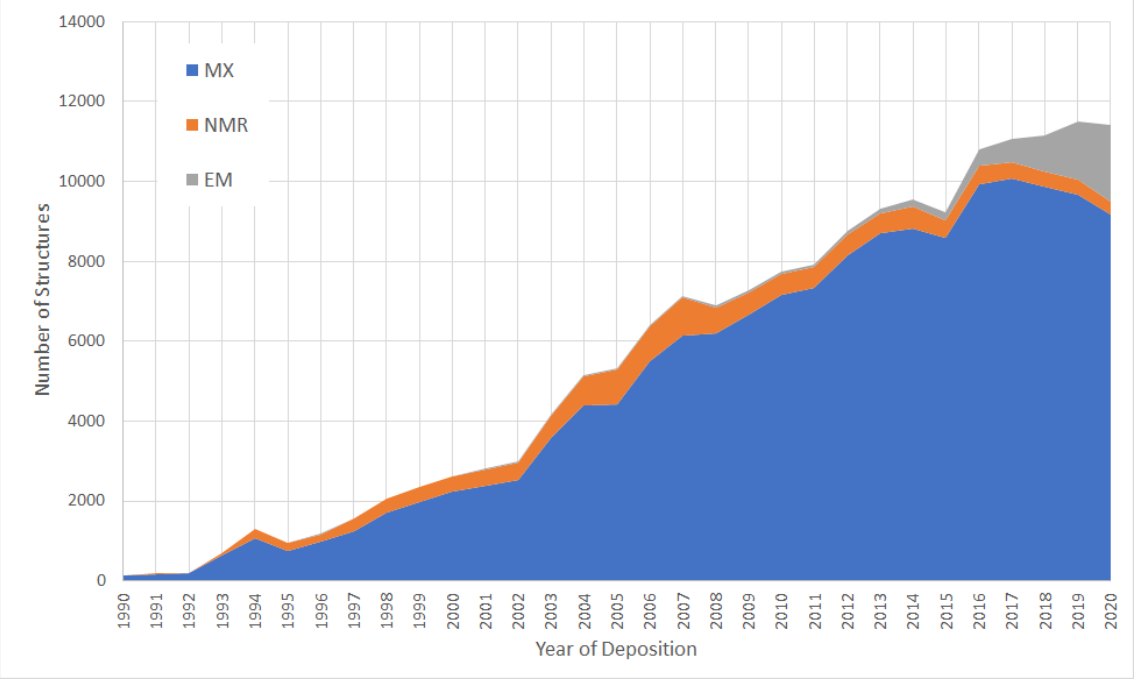

Figure 2: Number of structures deposited in the PDB between 1990 and 2020, by experimental method. Data from wwPDB, accurate as of December 2020 (41).

1.2.2 Protein structure solution pipeline

Collecting experimental data is merely the first step in a multi-step process of protein structure determination. Because of its prominence, this subsection discusses the structure solution pipeline for MX (*Figure 3*). Once data has been collected, in the form of diffraction patterns, also known as *reflection data*, there are several essential processing steps to be performed before a finished atomic model can be produced. Each of these steps is performed with the help of specialist software, many of which are bundled with model building suites such as CCP4 (42) or PHENIX (43).

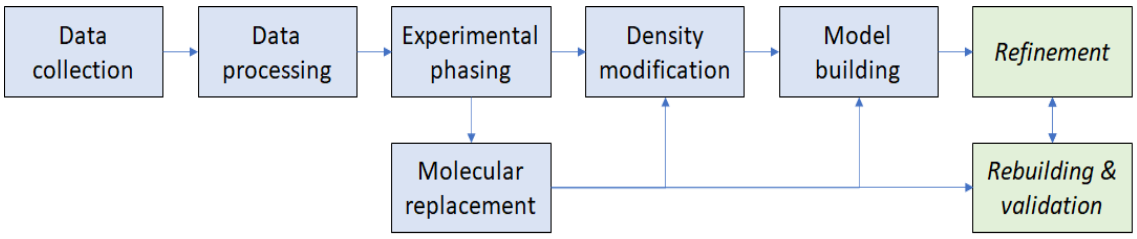

Figure 3: X-ray structure solution pipeline. Adapted from Kevin Cowtan, 2012 (44).

Every step of the structure solution pipeline involves unavoidable uncertainties, from the experimental errors introduced in the early stages, to the subjective decisions made during model building. Once a preliminary model has been created, it needs to be refined in an iterative cycle of refinement and validation, which mitigates against such uncertainties.

1.2.3 Protein structure validation

Validation of protein structure is performed using validation metrics, which provide information about various aspects of the atomic model. These can pertain either to individual parts of the model (local criteria) or to the model as a whole (global criteria).

Some validation metrics can be calculated from just an atomic model, using a model file: these are text files, primarily containing a list of atoms and their associated coordinates. The most ubiquitous format of model file is the Protein Data Bank format (*.pdb), the column order for which is shown in *Table 1*. Other types of model file include the newer PDBx/mmCIF format (*.cif) which is more extensible, but contains essentially the same data. Other validation metrics can only be calculated with access to the experimental data, in addition to the model file. Experimental-data files are specific to the experimental method. In the case of MX, experimental

data comes in the form of reflection data, a collection of X-ray diffraction observations. These data are most commonly stored in the MTZ format (*.mtz). From these data, an electron density map can be calculated, which can in turn be used to calculate density fit validation metrics.

Columns	Field Name	Data Type	Definition
1-6	record	String	e.g., "ATOM " for an atom
7-11	serial	Integer	Atom serial number
13-16	name	String	Atom name
17	altLoc	Character	Alternate location indicator, to indicate alternate conformation
18-20	resName	String	Residue name
22	chainID	Character	Chain identifier
23-26	resSeq	Integer	Residue sequence number
27	iCode	Character	Residue insertion code, used to differentiate between two residues with the same numbering
31-38	x	Float	Orthogonal coordinates for X (Å)
39-46	y	Float	Orthogonal coordinates for Y (Å)
47-54	z	Float	Orthogonal coordinates for Z (Å)
55-60	occupancy	Float	Occupancy
61-66	tempFactor	Float	Temperature factor (B-factor)
77-78	element	String	Element symbol
79-80	charge	String	Atomic charge

Table 1: Column order of the coordinate section of the PDB file format. From these data, model-only metrics including geometric analyses can be calculated. As shown in the third-to-last column, the model file also contains atomic B-factors. Hence, B-factor is also a model-only metric, despite the fact that B-factor values are originally determined using the experimental data. *Data from wwPDB* (45).

Model-only metrics inform about properties of an atomic model as a standalone entity, such as the bond geometry. This covers attributes such as deviations from ideal bond lengths, angles,

planes, and dihedrals. The result of these analyses is the detection of outliers: rare atomic arrangements, deemed unlikely to occur. Each outlier is either the result of an improbable but real feature of the protein structure (meaning the model is correct, and should be respected) or an error in the protein model (in which case the model should be corrected). In order to establish which of these two possibilities is the case for each outlier, the model must be compared to the electron density map derived from the experimental data, to assess the probability that the atoms in question were modelled correctly. This is frequently done manually, by visually comparing the model and electron density map in molecular modelling packages such as Coot (46) or CCP4MG (47), but can also be performed by applying local reflections-based validation metrics, including measures of electron density fit quality and B-factor. The real space correlation coefficient (RSCC) and real space R (RSR) are the most commonly applied metrics of local fit quality, and analysis has revealed that both show individual biases in their assessments of model accuracy (48).

Today, validation metrics can be produced in several different ways, the most prevalent of which being software suites such as CCP4 (42) and PHENIX (43), options and plugins in molecular modelling packages such as Coot and CCP4MG, and independent web services (*vide infra*). In recent years, the number of potential routes for model validation procedures has increased substantially, having developed from the smallest beginnings just decades ago. The demand for new validation metrics and more accurate refinement procedures is ever-increasing (49), and is sustained by periodic realisations that previously-deposited models are often imperfect (50–55).

The development of the structure validation process started following the inception of the field of macromolecular crystallography. In the beginning, there was no model refinement, since computational power was not widely available, especially not to the required extent. It was only in 1971 that the first automated least-squares refinement algorithm was published by Robert Diamond (56), which marked the start of computational protein structure refinement. The only available 'validation metrics' at this stage were the global indicators of R-factor and resolution.

Refinement remained a highly computationally intensive procedure. To tackle this problem, geometric restraints and constraints on atomic geometry were introduced to the refinement process. These served to reduce the dimensions of the least-squares matrix used in minimisation calculations by most refinement programs, which in turn reduced the computational intensity of model refinement, and were used in both small molecule (57,58) and macromolecular (59,60)

crystallography. These restraints and constraints would go on to themselves become useful metrics to highlight geometric irregularities in models: the birth of geometric model validation.

Over the following years, the exponential increase in available computational resources was paralleled by a growth in the number of macromolecular structure determination programs. The first validation software package, PROCHECK (61), was developed in the 1990s, and provided a variety of summary outputs, including a page of per-residue stereochemical analysis plots. These local analyses, although simplistic, proved to be extremely helpful for users, immediately guiding them towards areas of the model that may require further improvement or analysis. Comparably, the WHAT_IF (62) validation report, WHAT_CHECK (54), conducted various geometric validation analyses, as well as some analyses that were not present in PROCHECK, such as suggested side-chain flips and unsatisfied donors and acceptors (63).

The 1990s also saw the development of tools such as ProSA (66), which produced a single summary line graph of local model quality against residue number, as well as ERRAT (67), which plotted a nine-residue moving-window bar chart of a summary error value, and VERIFY3D (64,65), which produced a twenty-residue moving-window scatter plot of a protein's 3D profile score. In a similar vein to PROCHECK, these local summary plots were especially helpful in highlighting poor-quality areas of a structure.

Coot (46) transformed the field with its interactive output, by building upon interactivity introduced by software such as O (68). Coot featured scrollable self-updating charts to display the results of its diverse selection of integrated validation tools, many of which were built on the Clipper C++ libraries (69). These charts were presented in pop-up interfaces, and featured residue-by-residue charts for both reflections-based and geometric metrics.

In 2007, MolProbity rapidly became one of the most ubiquitous pieces of modern validation software, and still is today. MolProbity produces reports that feature high-quality geometric analyses, produced using proprietary methods of hydrogen-placement and all-atom contact analysis. Self-described as a "structure-validation web service", MolProbity geometry-based validation reports can be generated either using one of a few web-based MolProbity servers, or via the MolProbity libraries bundled in suites such as CCP4 and PHENIX. In the latter implementations, a local MolProbity server is initialised, which can be called upon by validation tools of the suite to perform back-end metrics calculations. The outputs can then be processed or presented by the validation tool as required.

PHENIX's Polygon (70) brought a one-shot graphical representation of overall model quality in the form of its radar chart, which could illustrate the values of several different quality indicators in a single view by plotting them along coloured bar charts emanating from a shared origin. This software was highly successful, and provided the basis for similar features in other multi-metric reports.

Along similar lines are the well-known Worldwide Protein Data Bank (wwPDB) summary quality sliders, which are featured on the summary page for every structure in the PDB. The sliders present a range of model-wide validation metrics in the form of percentile rankings, providing a single-view representation of how a model compares to similar models in the PDB on several different scales. These sliders were incorporated into the OneDep system, which was introduced by the wwPDB in 2014 (71). The full OneDep report features residue sequence plots which flag geometry outliers.

Today, computational structure validation is rich with a diverse array of software tools, including those mentioned here. Each brings valuable functionality to the table; however, because these features are scattered amongst so many different programs and suites, typical workflows involve running several different programs in series to obtain the required array of metrics, and obtain a comprehensive picture of the outcome of a refinement procedure. For example, one might start in molecular modelling software such as Coot, then apply a geometric validation suite such as MolProbity, and finally deposit to the PDB via the OneDep service.

The logical evolution from this manual refinement process, and towards a fully-automated iterative process, has been a long-time goal in the field. A large portion of the model building process has been automatable since even the 1990s, with the release of O and the programs that worked in concert with it, like OOPS (72), which featured automated procedures that greatly reduced the need for user input. The road to fully automated model building was paved by the ARP/wARP suite (73), which was able to produce essentially complete models from just the experimental data alone, thus pairing the model building and refinement processes.

More recently, software such as PHENIX's AutoBuild (74) has brought the field significantly closer to realising this goal. AutoBuild applies a repeated cycle of rebuilding and refinement to result in a largely complete model. Fully automated systems like these often enable the user to export the latest model file at each iteration of refinement, so that they can compare data from various steps along the overall process to follow the actions and progress of the automated procedure.

In summary, many of the most successful features in computational structure validation are spread across several validation tools, with no one program ticking all of the boxes. In order to remedy this, novel validation software should incorporate as many of those features as possible. In addition, they should not only perform both model-only and reflections-based analyses on a per-residue basis, but to be consistent with recent developments in automation, they should support integration within both new and existing model-building pipelines as an automatable task with low run time.

1.3 Project goals

The overarching goal of this project was to design and create a pluggable standalone validation software package to address the specific needs described in the previous subsection. It should be an all-in-one validation package that can calculate its own per-residue validation metrics, and also permit the incorporation of metrics from other validation services such as MolProbity. It should display all these metrics in a compact, interactive graphical interface that enables at-a-glance comparison between stages of automated model building. Finally, it should run quickly enough to be used either interactively or at the end of a new or existing validation pipeline with a negligible time penalty.

In this work, the development of such software is described and discussed, starting with the graphical interface, then the development of the integral metrics module, and the integration of these two parts to produce a complete all-in-one package. As an example of its integrability, the implementation of this package within the CCP4i2 graphical user interface is explained. Finally, the software is tested and analysed both quantitatively and qualitatively.

Software	Geometric analysis	Density fit analysis	Per-residue analysis	Supports integration	All-in-one graphics	Interactive
Coot (Validation menus)	Yes	Yes	Yes	Yes	No	Yes
MolProbity (Web server report)	Yes	No	Yes	Yes	No	No
Polygon (Comprehensive validation)	Yes	Yes	No	No	Yes	No
wwPDB (Validation sliders)	Yes	Yes	No	No	Yes	No
This Project	Yes	Yes	Yes	Yes	Yes	Yes

Table 2: Overview of some of the validation tools mentioned. All the programs identified have long run times, which are exacerbated in some cases by simple, but mandated, manual input. Coot performs all the desired analyses, but provides them in individual horizontally-scrolled bar charts, rather than in an all-in-one graphic. Similarly, MolProbity, which performs excellent per-residue geometric (but not reflections-based) analyses, provides its output as a vertically-scrolled table. Polygon and wwPDB both provide an all-in-one overview of a model, but not one with residue-by-residue analyses. *From Rochira and Agirre, 2020* (75).

2 Methods

Before designing the validation package, a few key aspects of its format had to be decided: the most important of which being which language to use – a traditional programming language or a scripting language. The most relevant examples of each in this context are C++ and Python, respectively. The primary difference between the two types of language is how they are run, with C++ code being compiled to machine code when it is first built, and Python code being read on-the-fly by an interpreter each time it is run. Compiled programs are inherently more efficient than interpreted programs, making them faster. They are also freestanding, in that they do not need an interpreter to be installed to run. However, scripted programs have their own advantages: their code is often more compact and easily readable (which is especially true in the case of Python vs C++), and they can be modified and re-run without waiting for a compiler every time, making them a good fit for quick prototyping and end-user customisability.

In the end, Python was chosen for the back-end of the package. Increasingly prevalent in the field, the Python interpreter is a component of all the major crystallographic software packages, ensuring compatibility for the package as a freestanding program or a plugin. In addition, language binding can be applied to include C++ code within a Python script to achieve the best of both worlds.

2.1 Interface

2.1.1 The interface format

Once Python had been decided as the back-end language, the first stage of package design was to decide the format of the front-end interface and to prototype a dummy user interface. The function of the interface would be to render the validation graphics, present them to the user, and then handle user interaction, updating the graphics in real-time. In performing these functions, the interface should also comply with the overarching project goals of fast execution, and maintaining compatibility with existing software packages.

Having Python as the back-end language meant the options for interface design were narrowed down to two broad possibilities: it could either be written as a Python-based graphical user

interface (GUI) or as a freestanding report exported by the Python script. Each method has its own advantages and disadvantages.

A Python-based GUI could be written using one of many available frameworks, including: PyQt, the Python binding to the Qt GUI toolkit; tkinter, the Python binding to the Tcl/Tk GUI toolkit; or wxPython, the Python wrapper for the GUI API wxWidgets. Launching a window directly from the script would enable a completely self-contained package; with the interface interactions being handled by Python code too, both the front and back ends would be written in the same language, and packaged alongside one another. This would make for a neat and extensible overall design, making customisation by the end user all the more intuitive. Indeed, many of the available Python-based crystallographic packages use this format for their own GUIs. For example, the CCP4i2 interface is based on PyQt, and the PHENIX interface on wxPython.

Alternatively, a freestanding report could take the form of a Portable Document Format (PDF) file, or a Hypertext Markup Language (HTML) document, with the latter being more suitable for the user interaction requirements of this project. However, the disadvantage of this format is that in contrast to the Python-based GUI solution, the code that handles interface interactions would have to be written in another language, with the most practical choice being JavaScript (JS). The only way around this would be so contrived as to defeat the purpose of an all-Python solution (such as requiring each user to run a local web server written in Python using the Django framework). However, there are a number of significant advantages to this format. For example, the HTML/JS format is well supported: every modern device has a built-in browser that can render HTML pages and parse JS, and the Python-based GUI options all support some sort of webviews that could render the HTML within a Python-based GUI. In addition, once a report has been produced and saved, it can be reloaded at a later date without having to be regenerated by the Python script. Finally, an HTML/JS solution, written properly, would be robust and stable.

With these considerations in mind, the HTML/JS report format was selected for the project interface. The next stage of interface design was deciding a method for graphic generation. Graphics could either be generated by the Python script and exported as part of the HTML report; or be generated by the JS code upon the loading of the HTML file by the user. In either case, pre-existing graphing packages such as Matplotlib (76) (for Python) or Chart.JS (77) (for JS) were ruled out. Despite the high level of customisability offered by such packages, the extent to which they would have to be customised to fit the exact requirements of this project would be

so great as to make their application counterproductive. The graphics would have to be bespoke, and custom-built from the ground up.

Knowing that the graphics would have to be drawn from scratch, there were two main routes to investigate. The first was to use the HTML5 canvas: a JS-controlled HTML5 element for drawing bitmaps in *immediate mode*, in which the image is drawn and retained in the client's memory when the page is loaded. This method was dismissed almost immediately, as its disadvantages were so numerous. Firstly, the fact that the image would have to be redrawn each time the page is loaded would lead to unnecessarily high loading times. Secondly, because the resulting image would be a single, flat bitmap, there would be no scope for interaction with individual 'elements' of the image, as they would be a purely visual construct. Lastly, the HTML5 canvas element may not even be supported by some of the integrated web-browsers in existing packages, as those browsers are often somewhat outdated.

The more favourable option was to generate graphics in the Scalable Vector Graphics (SVG) image format from the Python back-end. SVG images are text files, based on the Extensible Markup Language (XML) which can be rendered by all major modern web browsers, as well as most image viewers. The widely-supported nature of the format made it a good candidate for this project. The SVG images could either be saved as individual files, and then referenced within the report using one of a few compatible HTML elements (e.g., *img*, *svg*, *object*), or more appropriately, they could be embedded directly within the HTML file. Because both HTML and SVG files are XML documents, the SVG markup could be directly included within the HTML document in an inline implementation, such that child elements of the SVG file become a part of the overarching Document Object Model, and therefore children of the HTML document. Consequently, all elements of the SVG image would become accessible by any JS code that loads with the HTML document, enabling the desired JS-based interaction.

2.1.2 The graphics

With the HTML/SVG/JS format settled upon, the next stage of design was to prototype the graphics. The centrepiece of the package was to be a single chart that would represent the metric values for every residue in a chain in a single view. The basic idea behind this was a circular chart in which each residue of the chain is represented as a sector of the circle, and all of a residue's validation metrics are represented together within that residue's sector.

To quickly prototype chart designs, as well as the algorithms for their generation, ideas would be designed programmatically using a Python script. Before designing the actual graphics, a custom class was written to generate synthetic data, mimicking the metrics that could be extracted from a model. The flow of this class is shown in *Figure 4*.

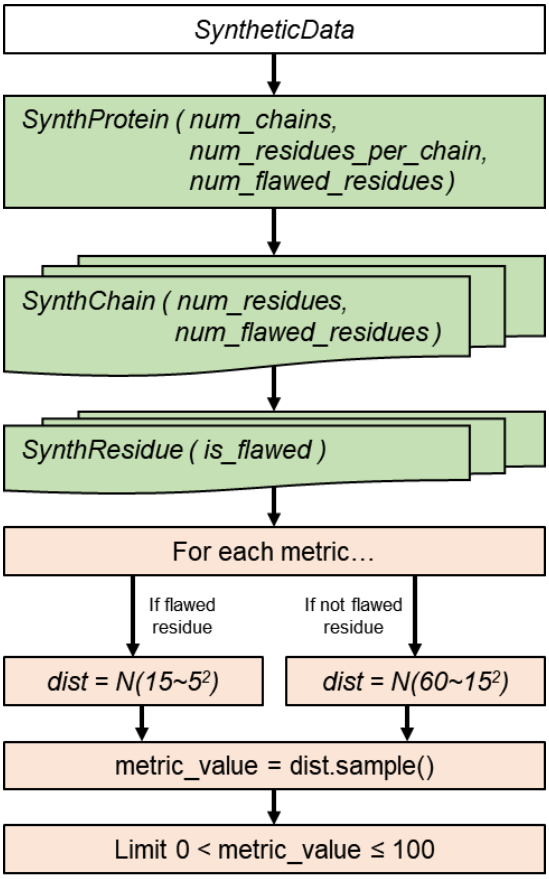

Figure 4: Flowchart outlining the synthetic data-generation class. For each residue, the custom class generated a synthetic metric value for each of five imaginary metrics; these values were generated by sampling from one of two normal distributions: either $N(60 \sim 15^2)$ or $N(15 \sim 5^2)$, with the latter distribution applied to a 'patch' of poor-quality residues somewhere within the chain.

Once the data generation was taken care of, design prototyping was started. In order to combat the subjectivity intrinsic to the graphic design process, a group of non-experts was assembled to periodically provide unbiased feedback. Throughout the design process, the group would be sent a design and a brief summary of the information that the graphic was intended to convey, and be asked for their individual opinions.

A Python module named *charts* was created to house the graphics-generation code. The initial designs were generated in the SVG format, using the open-source Python library svgwrite (78). The first idea trialled was a stacked radial bar chart, in which each residue was represented by a bar around the circle, containing each of the residue's validation metrics stacked on top of one another (*Figure 5*).

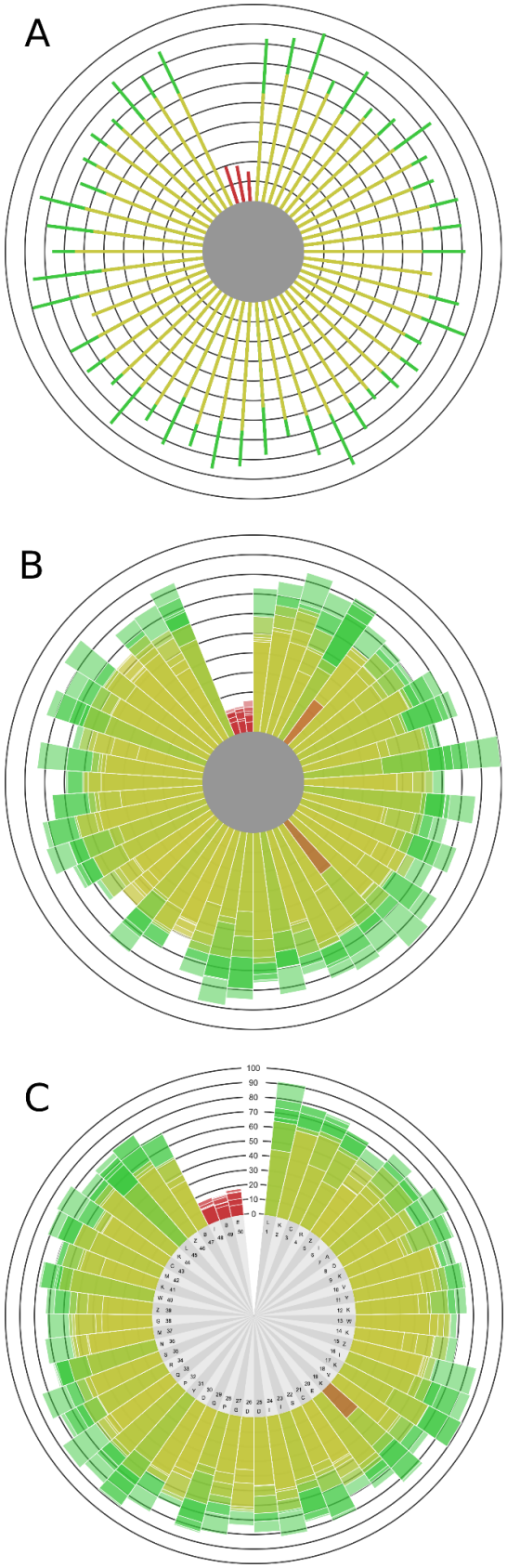

Figure 5: First designs of the chain-view chart. In these charts, each sector contains all the residue's validation metrics stacked on top of one another. The axis gradations for all metrics

are illustrated by the concentric black circles. In the first version (A), each residue was drawn as a rectangular bar extending out from the edge of the inner circle in the centre of the chart. The purpose of the inner circle was to prevent the overlap that would otherwise appear between bars at the dead centre of the circle, and to provide a clear separation between bars at lower values. In the next version (B), residues were drawn as sectors, forming a contiguous drawn area, and maximising the available space. To ensure that the divide between residues was still clear, a white border was added to each sector. In contrast to the stacked bars seen in the first version, sectors in this chart comprise a number of overlapping, opaque components, which each represent an individual metric value. Because each of these components has its own white border, it is possible to identify each individual metric value for a residue from this chart alone. In the final version (C), a number of minor changes were made. Firstly, the inner circle was expanded, further compressing the axis, and making it easier to read. The fill of the inner circle was changed from solid grey to alternating shades, as an additional measure of delineation between residues. The extra space within the inner circle was utilised to label residues with their respective sequence number and one-letter amino acid code. Finally, numerical axis labels were added.

Although this design made it very easy to spot residues with poor metrics across the board, it had a few flaws:

1) It was very difficult to precisely read metric values off the chart, a limitation of the axis style chosen.
2) It was not possible to identify which of a residue's metrics score well or poorly for an individual residue, and consequently, it is impossible to track the trend in any particular metric across a sequence of residues.
3) A residue scoring very well in one or two metrics would often disguise the fact that it scored very poorly in others, because the large green bars are much more visually striking than the small red bars beneath it, despite the fact that the small red bars are of greater significance, given that the primary goal of this graphic is to emphasise the areas of worst quality in a chain.
4) When generated for synthetic chains of higher residue counts, the chart would become increasingly difficult to interpret, and the labels would become unreadable (*Figure 6*).

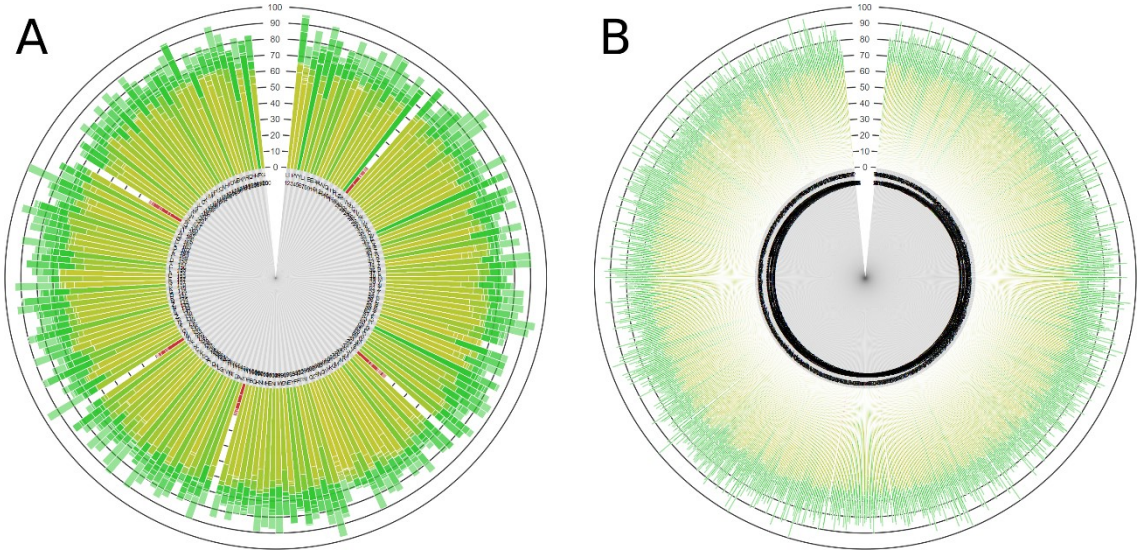

Figure 6: Chain-view charts illustrating the effect resulting from increasing the length of the synthetic chain to 200 residues (A) and 1,000 residues (B). Given that the median length of a protein in eukaryotic organisms is 472 amino acid residues, neither of these would be an unusually high number of residues.

In an effort to tackle the first two of these limitations, it was decided that a residue-view chart should be shown alongside the chain-view, which would show the details of an individual residue selected by the user via interaction with the chain-view chart. In this way, an individual residue's metric breakdown could be shown in detail, allowing the user to track the trend of an individual metric through the chain. The residue-view was designed as a form of radar chart, shown in *Figure 7*.

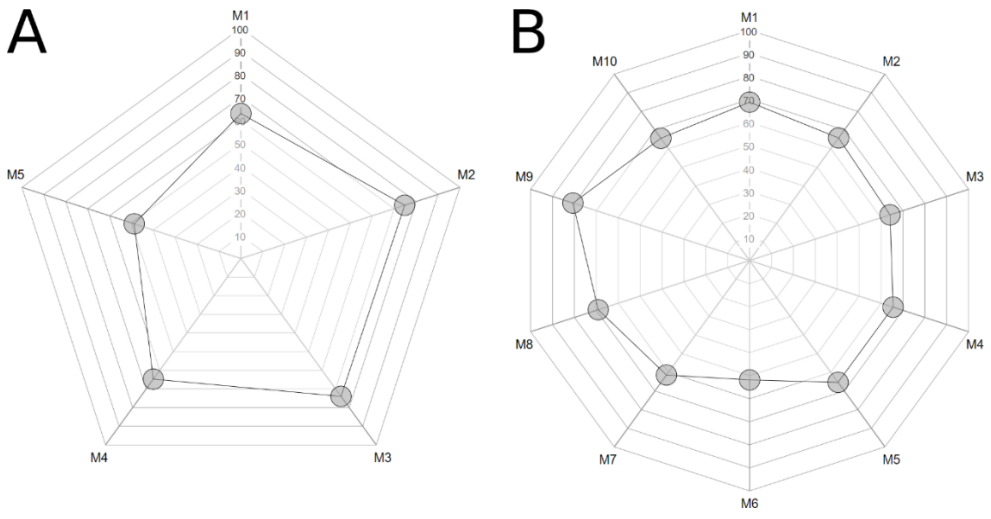

Figure 7: Radar chart residue-view, with five (A) or ten (B) different metrics. The shape of the radar chart is automatically updated based on the length of the metric-names array passed as

the only argument for the radar chart-generating function, an example of the procedural design made possible with this method of graphics generation. Like the chain-view chart, the radar chart axes were labelled on a scale from 0 to 100 (corresponding to the output value range of the synthetic data class) but with the axis length equal to the full radius of the chart, making it much easier to read off a precise value.

Although the residue-view chart was ideal for clarification of individual metric values for each residue, it could only be applied to one residue at a time, so did not entirely alleviate the second of the two problems it targeted. Ideally, the chain-view chart alone should illustrate the trend in a metric across the chain, without requiring an arduous process of manual checking from the user. Therefore, to solve this, and the rest of the limitations outlined, the chain-view chart would have to be redesigned.

The next conception of the chain-view chart was a series of radial line graphs plotted on concentric circular axes, where each axis would represent a different validation metric, and clockwise progression around the axes would correspond to the progression through the amino acid sequence of a protein chain. The hope for this design was that areas of poorer model quality would correlate with worse validation metric scores in several different metrics, making them easier to spot (*Figure 8A*). Initial ideas for this design were trialled using the same synthetic data generation mechanism as was used with the previous design. Each axis plotted the absolute metric value, with the axis limits set to [0, 100] and the origin therefore equal to 50. The results of these tests were disappointing (*Figure 8B*); the oscillations from the residues within the 'high-quality' normal distribution were so erratic that it was very difficult to identify the patch of low-quality residues, making it essentially useless. In an effort to reduce the noise produced by the uninteresting residues, an axis transformation was applied, whereby the square deviation from the mean was plotted instead (*Figure 8C*). This almost entirely alleviated the problem.

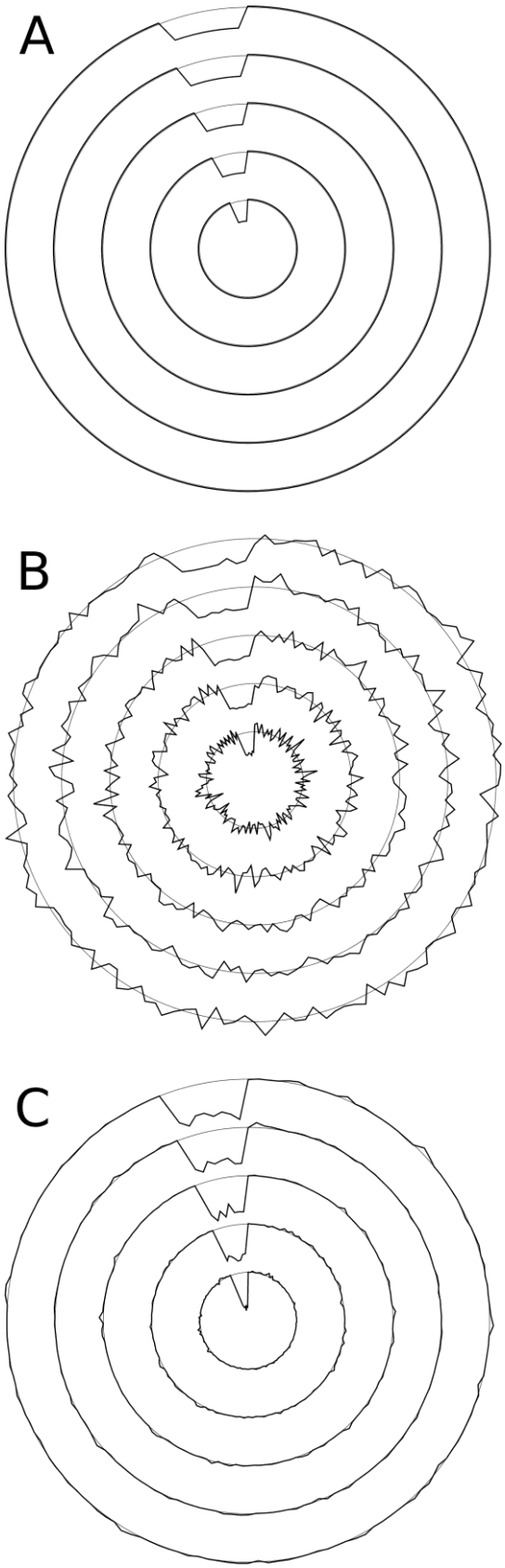

Figure 8: Originally-illustrated example for the idea behind chain-view chart (A), the result of the first test with synthetic data (B), and the result of the same test with the axis

transformation applied (C). The patch of poor residues is found between the eleven and twelve o'clock positions in all three charts. In the final chart, metric values are plotted as squared deviations from the mean metric values for the chain, so only the most extreme values are noticeable.

Satisfied with the axis transformation for the time being, it was decided that this design would be developed further. Firstly, a gap was added at the 12 o'clock position for axis labels. Then, each metric axis was designated an individual colour, to make each metric immediately identifiable from the chain view alone. Next, the area between each axis and its line plot was shaded in the same colour, to make an area's distance from the axis more immediately apparent. Finally, a sector residue-divider was added around the outside of the chart to introduce a sense of the size of the protein chain being viewed, as well as to provide some direction to the user to ease the selection of an individual residue for display on the residue-view chart once user interaction had been implemented. The result of these changes is shown in *Figure 9*.

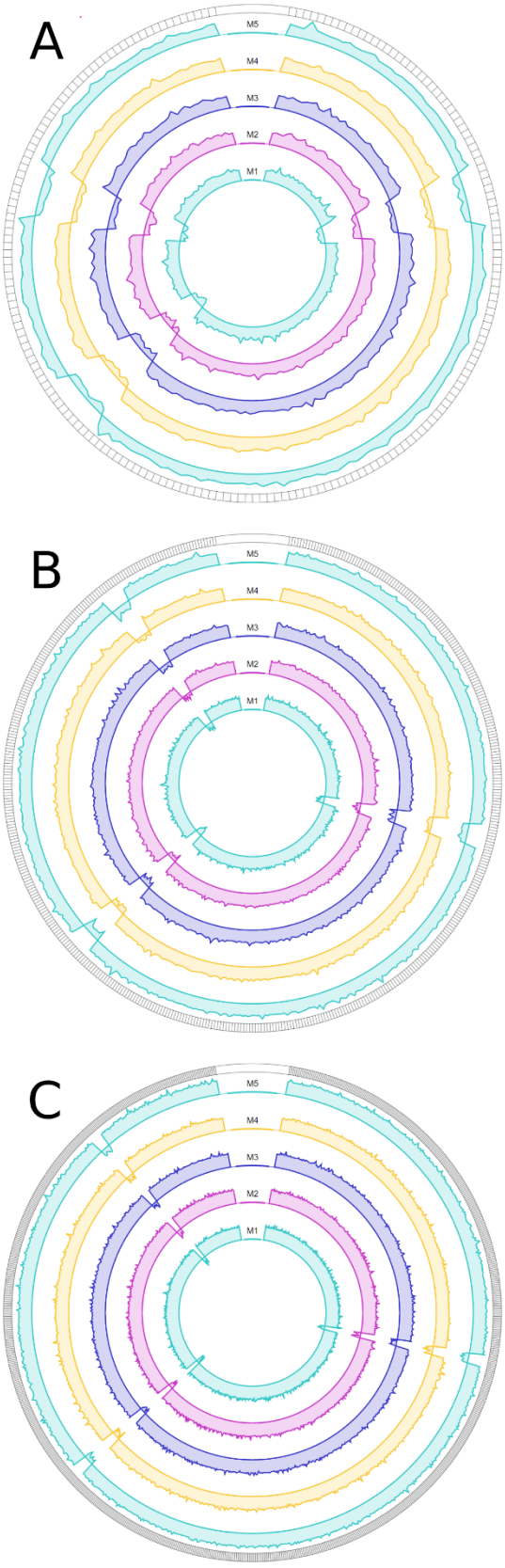

Figure 9: Updated chain-view chart. Produced from synthetic chains with 200 residues (A), 500 residues (B), and 1,000 residues (C). This design solved all of the problems arising from the

previous design: it is immediately apparent which residues score well or poorly in any given metric, the areas of worst quality are clearly emphasised, and the chart is equally readable with both low or high residue counts.

One of the most-desired features of the validation report was that it should be able to compare different iterations of a model within the same chart. To accomplish this, the synthetic data class was restructured to produce two different *SyntheticProtein* objects with similar metrics values, to represent the iterations of a protein model both before and after some hypothetical refinement process. Different 'ghosting' methods were trialled, to show the values corresponding to the 'previous' iteration of the protein alongside those of the 'current' iteration. The results of these tests are shown in *Figure 10*.

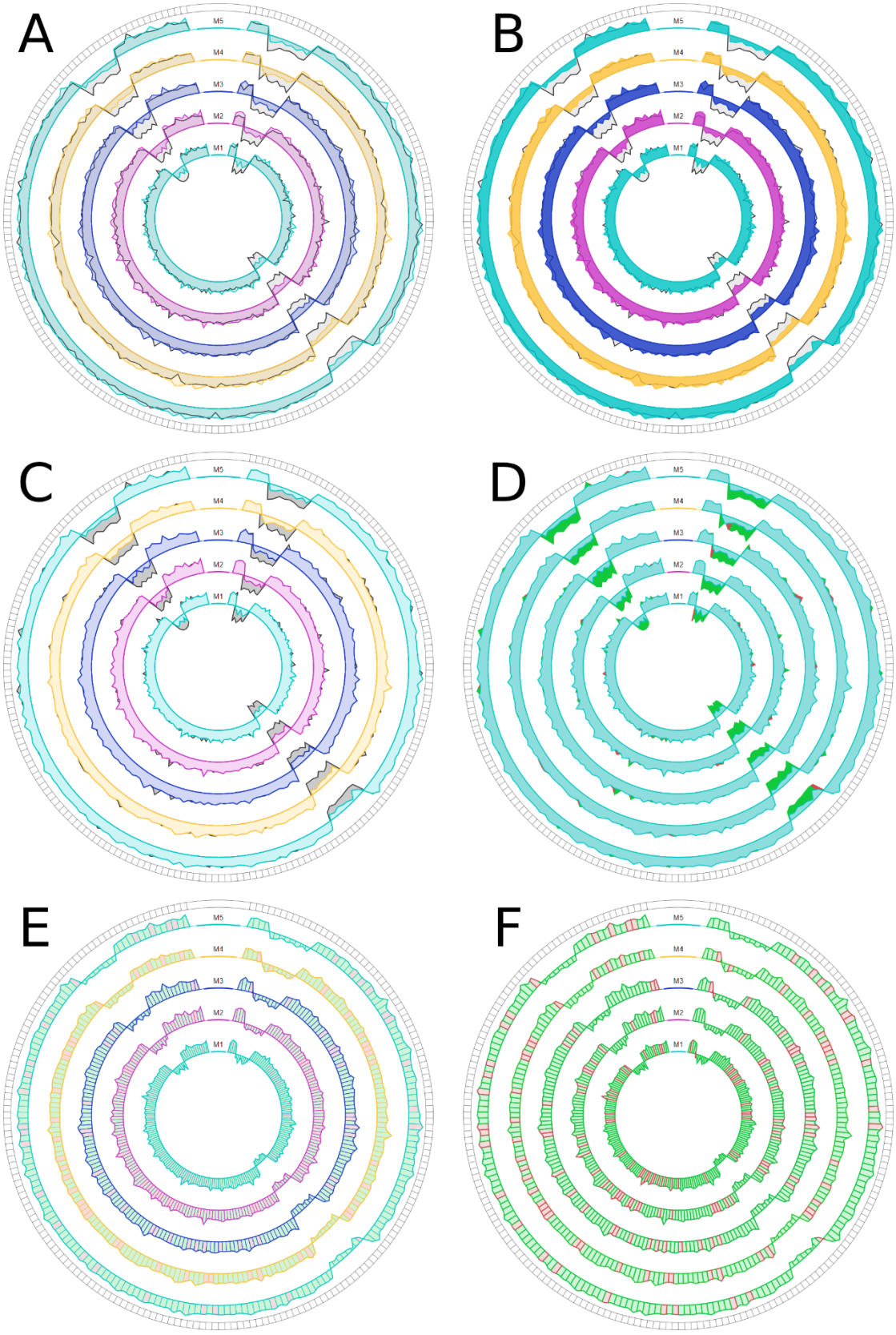

Figure 10: Selection of different chain-view ghosting designs. All charts were generated from the same synthetic chain, of length 200 residues. Charts A, B, and C are all based on the same

idea: for each metric axis, the previous model data is drawn as a grey shaded area behind the latest model data's individually coloured shaded area. The only difference between these three charts lies in the opacities of the various layers. Of the three, Chart C was preferred by the majority of testers. Chart D is based on a slightly different idea. Again, the previous model data is drawn behind the latest model data, but the latest model data is all shaded in cyan, and the previous model data is shaded a different colour for each residue: green if the metric value for that residue improved across the iteration, or red otherwise. Charts E and F are based on a different idea, a hybrid of the principles of the other charts. Rather than drawing separate areas for both the latest and previous model data, only one area is drawn for each metric, with the shape representing the metric values of the latest mode, and the individual segments that comprise the area individually filled and outlined with a colour that represents the residue's metric value improvement over the iteration, as in chart D. User feedback indicated that the charts featuring per-residue colouration were too 'busy' to be easily interpreted, ruling out charts D, E, and F. Of the first three charts, Chart C was widely selected as the most readable combination of opacities, and so it was selected as the working design with which to proceed.

2.1.3 The report

As both charts had arrived at a suitable stage of development, the next task was to design the HTML interface that would display them. A submodule named *report* was created as a part of the *interface* model, alongside the *charts* submodule.

It was decided that the most effective way to design the layout of the HTML report would be to use the open-source package Bootstrap (79). A template was adapted from open-source code available on the Bootstrap website (80) which provided suitable view compartmentalisation. The adapted template is shown in *Figure 11*.

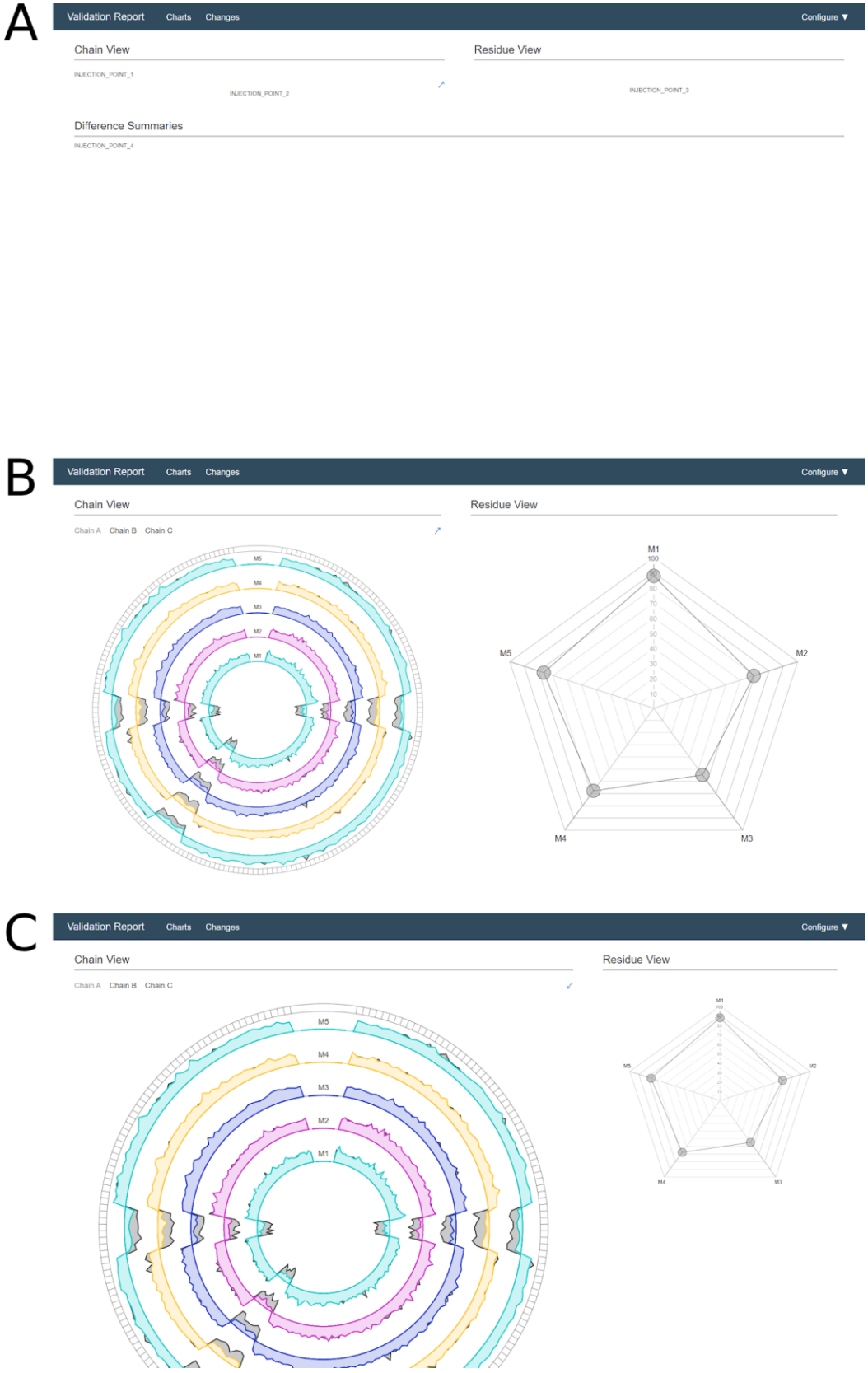

Figure 11: First HTML template designed for the report. The report is divided vertically into sections, which are either scrolled between manually or jumped to by using the buttons on the

left side of the navigation bar, which is frozen in place. The topmost section of the report is to contain the chain-view and residue-view charts, presented side-by side. The rest of the template constitutes placeholders for intended additions, including a section for verbose tabulated data. Likewise, the *Configure* button on the right-hand side of the navigation bar was designed to trigger a dropdown menu containing options for on-the-fly configuration, to be designed at a later date. The injection point strings are used as flags for the Python script to insert the HTML code specific to each report. Injection points 2 and 3 are the flags for the chain- and residue-view charts respectively. Injection point one is the flag for the chain-selector bar, shown above the chain-view chart. Because the chain- and residue-view charts have similar aspect ratios (roughly 1:1) their respective columns must be equally sized for them to line up vertically. However, the chain-view chart is much more information-dense than the radar chart, and the user is likely to favour a larger chain view. Hence, the resize arrow in the upper-right corner of the chain-view chart was added, which can be used to expand the chain-view column and contract the residue-view column.

Once the HTML template had been designed, the next step was to write the JS code that would enable user interaction by tying the SVG elements together. The basic premise behind this was that there would be two JS files associated with each report. One file would contain the code common to every report; for example, the functions hooking the methods of the embedded SVG images. This file would be hardcoded and packaged with the template. The other file would contain the metrics data specific to that model, and would be individually generated by the Python script for each report. In this way, the groundwork for the JS was laid; its layout is shown in *Figure 12*.

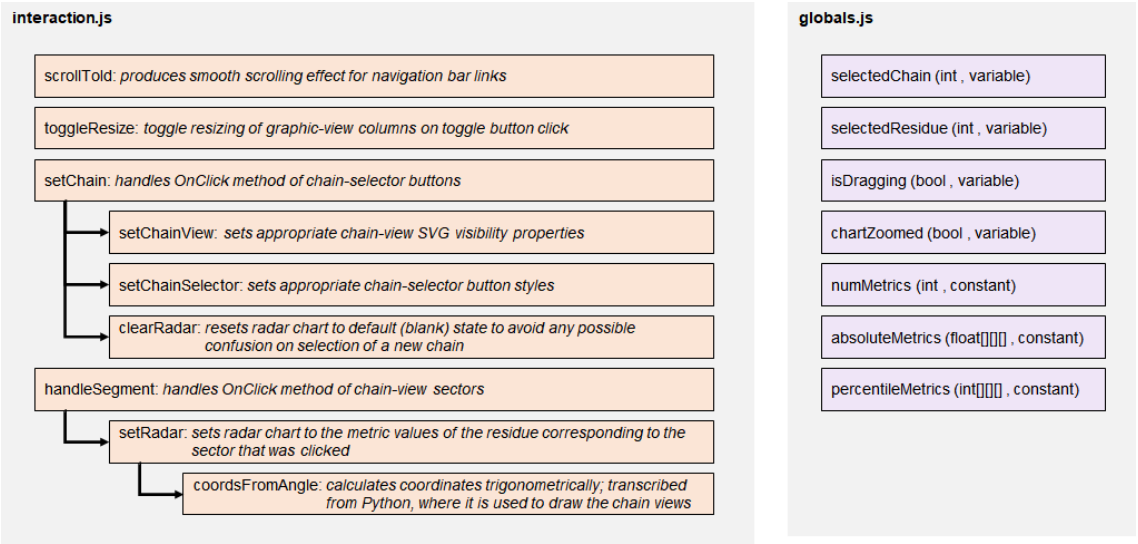

Figure 12: Functions and variables of the primitive report JS.

With the basic JS written and the report functional, some areas for potential improvement became apparent. For example, it was difficult to know if the correct residue had been selected, since there was no indication as to which sector was last clicked, or indeed which residue this corresponded to. To rectify this, two steps were taken: a sector selector was added to the chain-view chart (*Figure 13A*), and a residue info text line was added to the HTML template, above the radar chart, to be updated by the JS with a description of the selected residue.

Another area for improvement was that the radar chart was not as informative as it had the potential to be; for example, it made no use of user interactivity, or of colour. It was felt that these design elements could be usefully applied. To address the former, hooks were added for the *OnMouseOver* and *OnMouseOut* events of the radar plot-point circles, which are triggered when the user's cursor enters and exits each circle, respectively. A function was then added such that hovering over any circle would trigger the appearance of a bubble directly above it, containing the numerical percentile value represented by that point, which would disappear once the cursor was moved away. To address the latter, inspiration was taken from POLYGON (70), and coloured bar chart-like distribution representations were added to each of the radar chart axes, to provide the user with an indication of the chain's distribution for each metric, and where the selected residue point falls within that distribution (*Figure 13B*). User feedback on the along-axis distribution representations was generally unfavourable. Although the information conveyed was appreciated, it was generally felt that the implementation made the chart too cluttered, and so it was removed. In place of these, the radar-setting JS code was modified such

that it would fill the plot-point circles with a colour corresponding to the position of that particular value within the chain's distribution of values for that metric (*Figure 13C*).

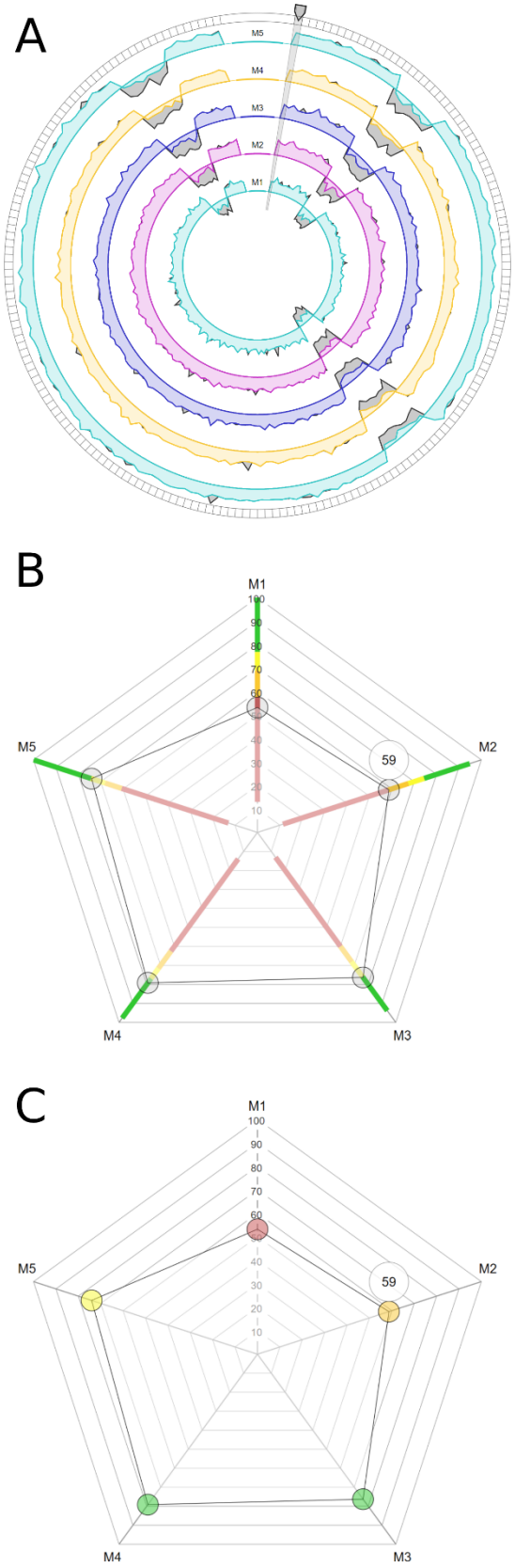

Figure 13: Updated chain-view chart (A) and radar charts (B, C). Note that while charts B and C represent the same synthetic residue, this residue does not correspond to that selected in

chart A. Chart A shows the selector added to indicate the position of the currently-selected residue. In charts B and C, the user's cursor is hovering over the plot-point circle for the metric *M2*.

The whole report generation procedure was tested thoroughly and repeatedly using synthetic data. After the necessary updates to the JS code (*Figure 14*), the report was fully functional. An example report is shown in *Figure 15*, and the final structure of the *interface* module is shown in *Figure 16*.

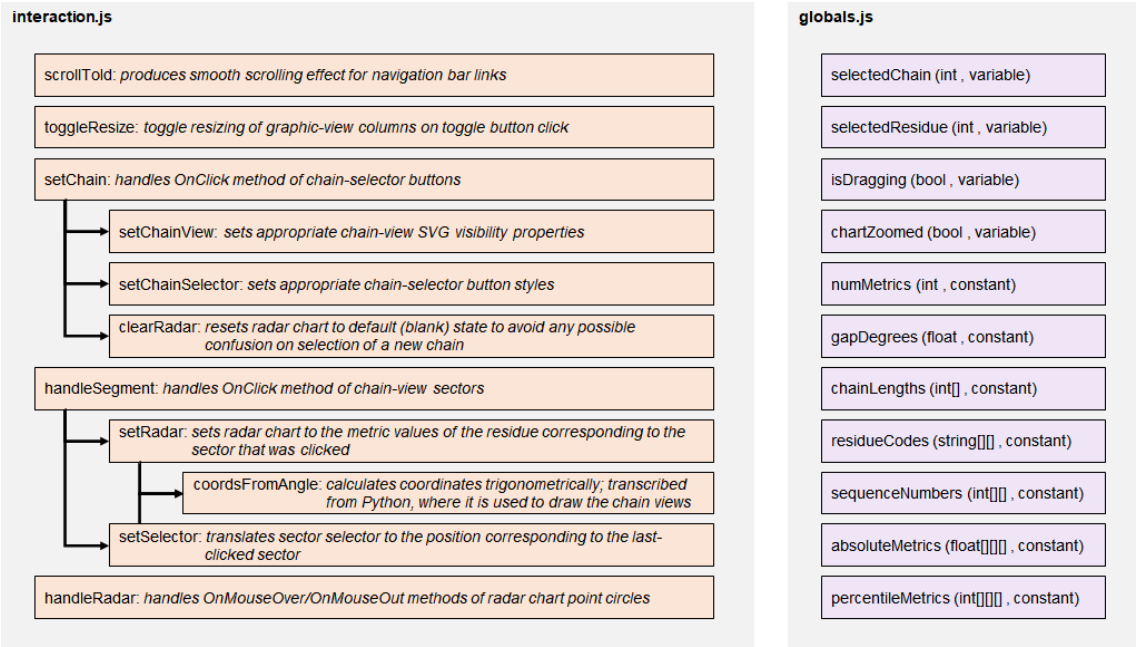

Figure 14: Updated functions and variables of the report JS.

Figure 15: Example validation report generated using the initial design of the interface module.

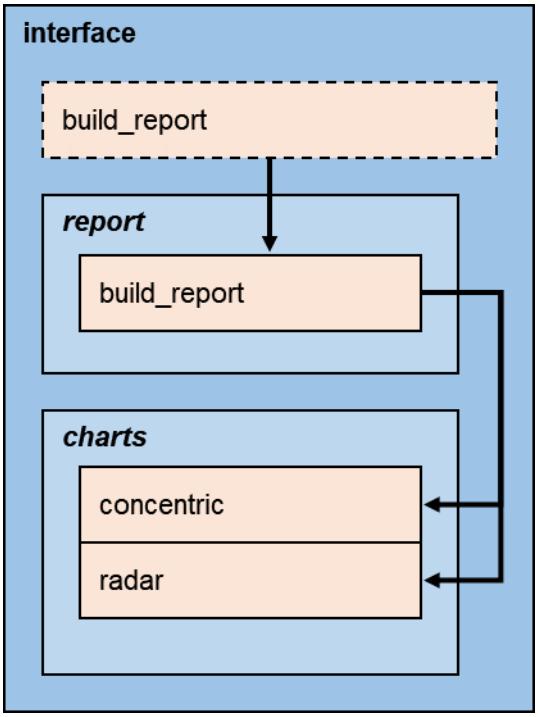

Figure 16: Structure of the interface module at this stage of the project. The *build_report* function of the *interface* module constructor calls the *build_report* function of the *report* submodule, which uses (synthetic) model data to call functions of the *charts* submodule to generate the chart SVGs, and then builds the HTML report around those SVGs.

With this, the preliminary version of the report was complete, and the project progressed to the next stage: generating real-world metrics data.

2.2 Metrics

2.2.1 Preparation

Before development of the *metrics* module was started, a library of model data was assembled to enable testing of the module both during and following its development. To do this, the entire PDB-REDO database was downloaded. PDB-REDO is an automated re-refinement and rebuilding procedure that has been performed on every model in the PDB that has experimental data associated with it (81–83). The result is a database containing the pre- and post-refinement model and experimental data for every structure, of which there were over 135,000 at the time of writing.

In addition to the PDB-REDO database, a selection of unrefined models was precured; these models were outputs of the Buccaneer (84) software for automated model building. These would be used to test the software on incomplete models, as examples of models from early stages in the refinement-validation cycle, to more closely emulate real use-cases.

2.2.2 Outline

The plan for the *metrics* module was to create as comprehensive an array of per-residue metrics as could realistically be done in the timeframe of the project. These metrics would comprise both model-only and reflections-based metrics.

With Python already established as the language for the module, it was immediately decided that the built-in metrics calculations would be based on the highly efficient Clipper and MiniMol libraries (69). Though originally written in C++, the Clipper-Python C++ bindings make it possible to use the library from a Python interface (85). The framework of the library was to be based on the MiniMol objects, a hierarchical system of classes that encapsulate model data. Reflection data is handled by separate Clipper classes (see *Section 2.2.5*). The basic idea was to create a custom class hierarchy that would take objects from each level of the MiniMol cascade as

arguments, inheriting their attributes and performing metrics calculations alongside iteration through the MiniMol cascade (*Figure 17*).

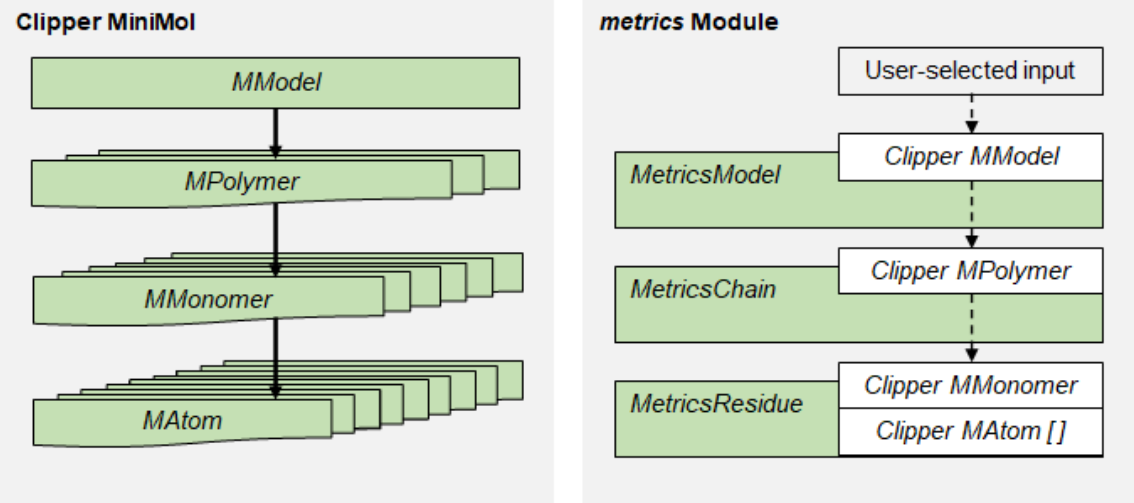

Figure 17: MiniMol cascade (left), and its intended implementation within the metrics module (right). The *MModel* class corresponds to a protein model, the *MPolymer* class to a chain in that model, the *MMonomer* class to a residue in that chain, and the *MAtom* class to a constituent atom of that residue. Looping through the model file in this way would enable access to the base attributes of objects on each level, which could then be used to calculate metrics based on the relevant values stored in the model files.

2.2.3 Initial framework

The first stages of development were to construct the foundations of the module: some class that could open and read a model file, then iterate through the MiniMol cascade to inherit various properties at each level. Reflection data would be handled later (*Section 2.2.5*). Of course, a prerequisite for this was to install the Clipper-Python package. Unfortunately, there was no available distribution of the Clipper-Python package for the chosen operating system and Python environment under the Python package manager (pip), only empty placeholder packages. To circumvent this, the CCP4 suite was installed, which comes with an ad-hoc version of the module as part of its own CCP4-Python environment.

Once Clipper-Python was installed, the basic framework of the module was developed; a class was written that could load an *MModel* object from a model file, and iterate through the MiniMol cascade, printing out various attributes of the MiniMol object corresponding to the

level of iteration; for example, each *MPolymer*'s chain letter code, and each *MMonomer*'s amino acid type.

The script was tested on a few models from the PDB-REDO database mirror, and the output was judged to be satisfactory. However, this exercise revealed an unforeseen obstacle: *MMonomer* objects were not necessarily valid amino acid residues: they could be incomplete residues, or other molecules altogether, such as water. Therefore, a function was written that would, for a given *MMonomer*, determine whether or not it was an amino acid residue, by checking if its code corresponds to a valid amino acid. Because the development of the *metrics* module would involve the creation of a number of other functions similar to this one, to execute various calculations and algorithms (and many of these functions would be generally useful utilities) a separate *utils* module was created specifically to house them, rather than including them as methods of the *MetricsResidue* class (*Figure 18*). This was done so that other scripts would be able to call these functions as standalone entities, rather than having to initiate an entire *MetricsModel* object.

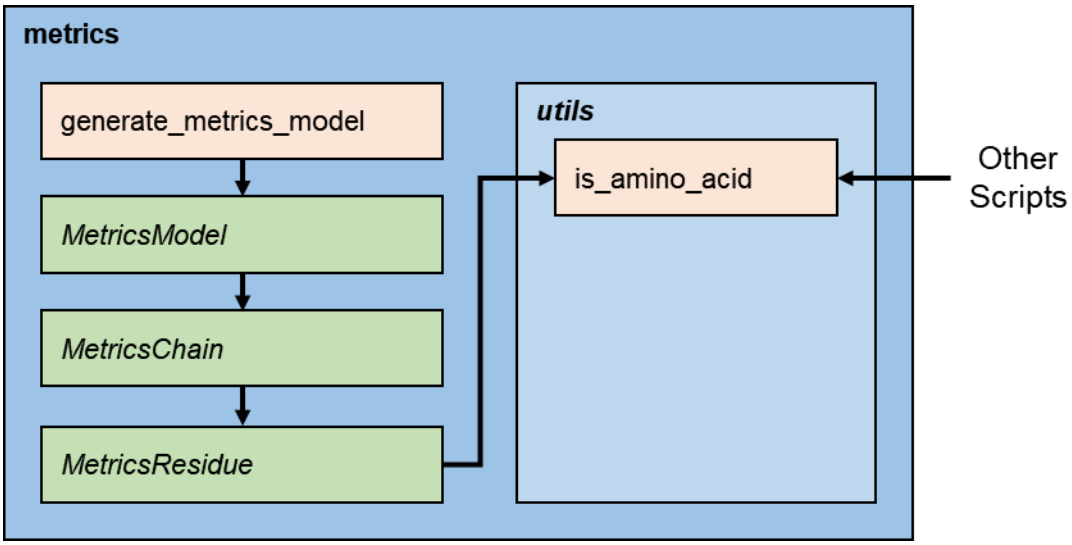

Figure 18: Structure of the metrics module at this point.

2.2.4 Model-only metrics

With the basic framework of the module established, the project progressed to calculating metrics from the now-accessible MiniMol attributes.

2.2.4.1 B-factors

The first metrics developed were the B-factor analyses, which were straightforward to implement. Each Clipper *MAtom* object has a method to get its orthogonal isotropic displacement value (U value) which is related to B-factor by a constant (*Equation 1*). Therefore, a function was written to enumerate the *MAtom* objects of a given *MMonomer* object, convert each U value to a B-factor, and append that B-factor to an array, from which values including the minimum, maximum, mean, and standard deviation could be calculated and returned. This function was added to the utils module, to be called by the *MetricsResidue* constructor upon initialisation, which would set the returned values as attributes of each *MetricsResidue* instance.

$$B = 8\pi^2 U$$

Equation 1: Formula for calculating B-factor (B) from isotropic displacement value (U).

2.2.4.2 Bond geometry

The next task was to utilise the coordinates data to calculate bond geometries. Before this could be done, a function was written to classify the atoms of an *MMonomer* object as belonging to either the main-chain or side-chain, so that operations unique to either of those groups could be performed more readily. This function was added to the *utils* module, also to be called by the *MetricsResidue* constructor.

Next, the bond geometry calculation functions were written, the first of which was a function to calculate bond lengths, both along the main-chain and side-chain of each residue. While this would not lead to the calculation of a validation metric, it was a useful check function; it had been noted that some of the unrefined test models would elicit *MMonomer* objects that had missing atoms or chemically unfeasible bond lengths, as a result of some imperfect refinement step. Scoring such residues on validation metrics designed for chemically feasible residues would be misleading and unreliable, so they should not be treated as real amino acids, and should instead be considered unscorable. For this reason, a function to check atomic composition and bond lengths was written for the *utils* module, to be called by the pre-existing *is_amino_acid* function, to more thoroughly discriminate between *MMonomer* objects that represented real amino acid residues and those that did not.

The next values to calculate were the bond torsion angles, which would be used to determine the energetic favourability of the various elements of an individual residue's geometric conformation, shown in *Figure 19*. These could be calculated with some simple matrix operations, the general formula for which is *Equation 2*. A function was written to do this, and again incorporated in the *utils* module to be called by the *MetricsResidue* constructor.

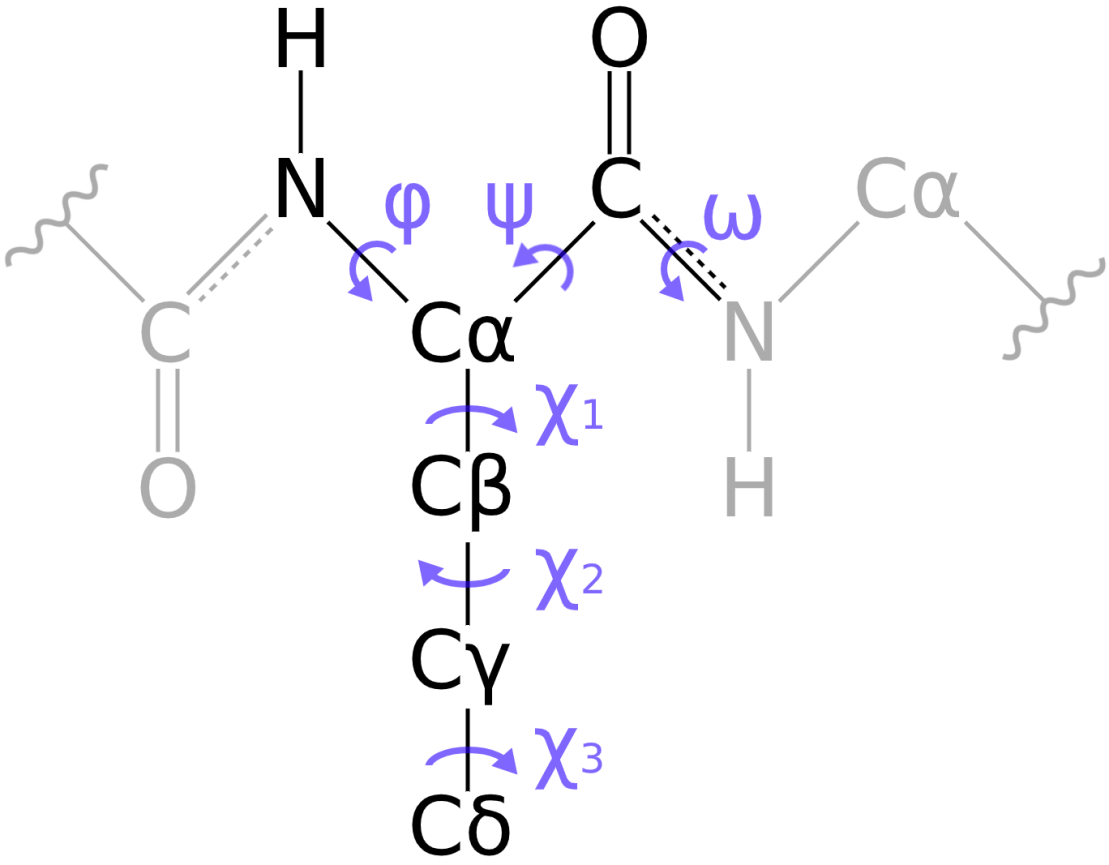

Figure 19: The various amino acid bond torsion angles. The atoms and bonds in black represent a hypothetical amino acid residue in some polypeptide chain. The main-chain torsion angles are phi (ϕ), psi (ψ), and omega (ω), with the former two used to characterise main-chain (Ramachandran) conformation. Omega values tend to be very close to 180 degrees, as a result of the bond's significant pi character; consequently, omega deviation can be a useful validation metric in itself. The chi (χ) angles are found along an amino acid's R-group; the number of them is dependent on the amino acid type. These angles are used to categorise side-chain (rotamer) conformation.

$$\varphi = \text{atan2}(u_2 \cdot [(u_1 \times u_2) \times (u_2 \times u_3)], |u_2|(u_1 \times u_2) \cdot (u_2 \times u_3))$$
$$Where\ u_n = r_{n+1} - r_n$$

Equation 2: Formula for calculating dihedral angles from three-dimensional coordinates; where u_n is the n^{th} bond vector, and r_n is the n^{th} bond coordinates vector.

For an array of torsion angles to become a meaningful validation metric, it needs to be compared to a probability distribution to ascertain a likelihood score for that conformation. These probability distributions are calculated from reference data: a selection of data curated from known high-confidence structures. The Richardson lab has published a public repository of reference data for different types of residue geometry, based on thousands of high-resolution, quality-filtered protein chains, called Top8000 (86). The Top8000 data has reference data for three categories of geometry: main-chain torsion, side-chain torsion, and torsion about the Cα atom, known as CaBLAM (87). For each of these three categories, the dataset contains a set of contour-grids: regularly spaced, multidimensional probability distributions. A rendering of one such example of these grids is shown in *Figure 20*.

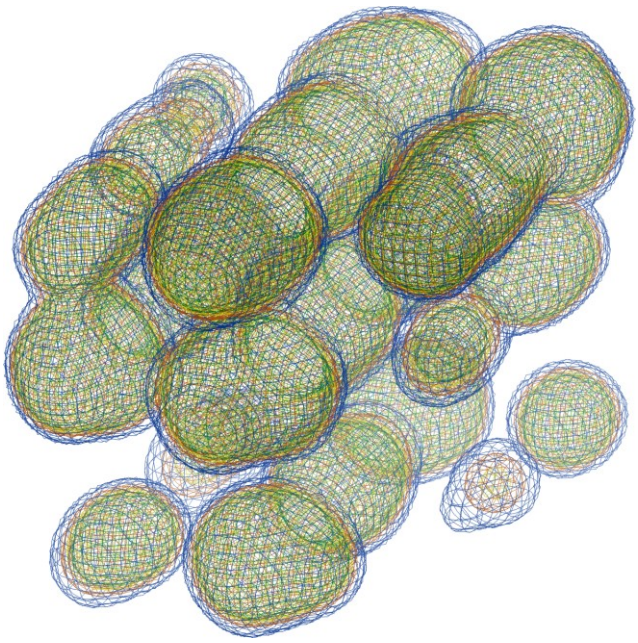

Figure 20: Top8000 contour grid probability distribution for chi angles of the methionine side-chain. The methionine side-chain has three chi angles, so the probability distribution can be plotted as contours in three-dimensional space, where each spatial dimension represents one side-chain torsion angle. This image was taken with the KiNG software (88) by loading the Kinemage file included with the reference data.

Therefore, the next stage of the project was to implement these data in order to convert the now-calculable dihedral angles into a significant likelihood score. This was no small task, as it needed to be done while bearing the overarching goals of the project in mind; specifically, keeping the run time low and the overall size of the package small. With a program like this, the tolerance for time spent loading data on execution is very low. This is because, in contrast to a program such as Coot, which is expected to spend some time loading resources to memory on initialisation, which then remain in memory to then be called upon for a number of analyses, this program would have to load up and shut down every single time it performs an individual analysis. Because of this, an increase in load-up or shut-down time would have a direct impact upon the run time for each individual analysis. This concept is illustrated in *Figure 21*.

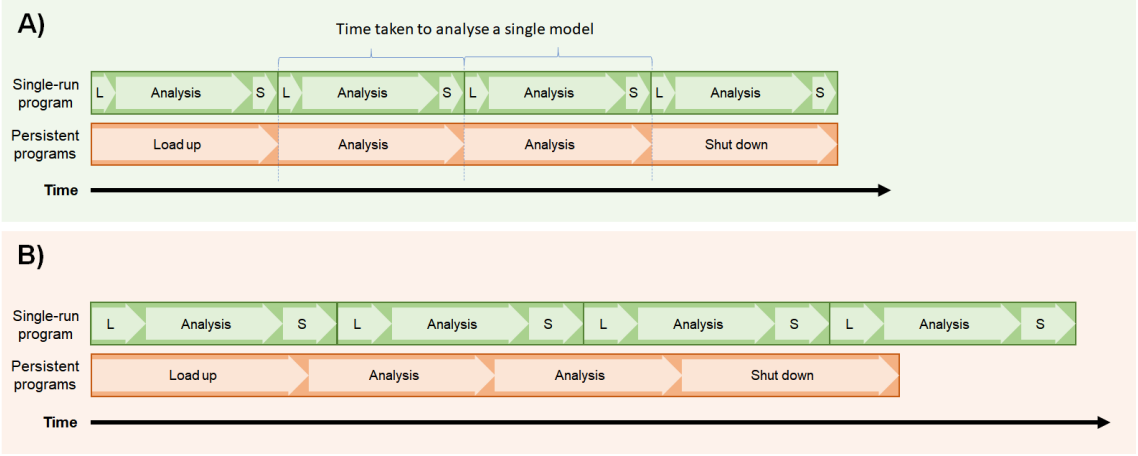

Figure 21: Time allocation in a single-run script compared to a persistent program. Before an increase in load-up and shut-down times (A), and after an identical increase in both load-up and shut-down times for both classes of program (B). As the load-up or shut-down times increase, the length of the single analysis is directly affected in the case of a single-run program, which is not so for persistent programs. As a consequence, this overhead needs to be kept to a minimum.

2.2.4.3 Ramachandran conformation

Looking first at the main-chain bond torsions: the Clipper libraries already include the Top8000 data for Ramachandran configurations, which is implemented in a calculator class that has a high-accuracy interpolation method built in. This class is hooked in the Clipper-Python bindings, which made this the obvious choice for calculating Ramachandran conformation quality in the

metrics module, especially because this route would comply with both the aforementioned goals: the size of the package would not increase (as the Clipper library would already be installed as a dependency) and the run time taken both to initialise the calculator on start-up and to make individual queries would be low, thanks to the low-overhead C++ calculations utilised by the class. Once again, a *utils* function was written to decide and execute the suitable Clipper Ramachandran calculator method.

2.2.4.4 Rotamer conformation

In the case of side-chain torsion angles, there was no Clipper class to do the work. Instead, the data had to be implemented manually. The first tests investigated the best way to load all the data to memory on start-up. The contour grid data in the Top8000 repository are provided as plaintext files specific to each amino acid type, totalling 37.4 megabytes in size. Each of these files was loaded to a dictionary, with the keys being arrays of chi angles, and the values being the associated probabilities. This process took 2080±80 milliseconds (n=100), which was a high – but acceptable – figure, and required roughly 200 megabytes of memory, which was also acceptable.

Although the contour grids were regularly spaced, the files encoding them had a number of missing data points. This meant that for a given array of calculated chi angles, it would not be possible to simply calculate the coordinates of the closest points on the contour grid, because those points could be missing. This problem could be tackled a few different ways. The first would be to repackage the data in a structure more conducive to fast lookups, reducing the time taken to find the closest points with an associated probability value. This method was tested by loading the data into a *k*-d tree: a binary search tree that places each data point as a node in k-dimensional space. This was accomplished by using the *KDTree* class of the SciPy library (89), the *query* method of which will find the nearest neighbours for any given coordinates. A search operation on a binary search tree runs in logarithmic time ($O(\log n)$) in the worst case, and one operation can return the set of all nearest neighbours. This is significantly faster than looking up the coordinates as keys in a dictionary, for which the worst-case performance is linear time ($O(n)$), and may need to be repeated multiple times. Unsurprisingly, this approach proved to be significantly faster than dictionary lookup, but in practice remains slow and memory intensive.

Dissatisfied with the *k*-d tree approach, the next method tried was to preprocess the data by re-interpolating it, with the goal of producing a new regular dataset with no missing data points. Theoretically, the result would be regular arrays with almost instantaneous lookups. The first interpolation tests were performed again using the SciPy library, specifically its *interpolate* module. Interpolation was performed at each integer degree in the feasible range of each chi dimension. This was a time consuming and computationally intensive process. Interpolating the seven two-dimensional side-chains took 1.40±0.07 seconds altogether (n=100 repeats), and interpolating the three three-dimensional side-chains took 43320±30 seconds altogether (n=5 repeats), or roughly twelve hours. Interpolating any of the four-dimensional side-chains to this level would have required more memory than was available, so could not be attempted. In any case, the result of such interpolation, even for the three-dimensional side-chains, were data structures that were gigabytes in size. Clearly, interpolation was not a viable option, at least to such a high level of precision. Some of these difficulties could be overcome by instead writing a C++ program to perform the interpolation. But even then, preprocessing in this way would result in huge libraries that would have to have been packaged with the module. These would then take a very long time and a lot of memory to load on start-up, if sufficient free memory were even available.

Since every one of these methods was found to be unsatisfactory in some way, a completely different method was conceived. The Top8000 rotamer data is also provided for each of the rotameric canonical amino acids in another form: a set of *central values*, which lists the mean and standard deviation of the bond torsions for each recognised rotamer. These files are much smaller than the contour grids, totalling 36.7 kilobytes in size. As a result, these lists could be loaded to memory almost instantly on run time. To take advantage of these data, a new score was devised: a given array of calculated chi angles would be compared to the array of means and standard deviations of each recognised rotamer for that amino acid, so as to calculate a score. The overall score for that array of chi angles would therefore be the best of all the calculated scores. The formula applied for the score (*E3 3*) was similar to a multidimensional z-score, such that the lower the score, the more likely it is that the array of chi angles fits the distributions of chi angles of a recognised rotamer.

$$Score = \min_{i} \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(\frac{\chi_n - \mu_{\chi i_n}}{\sigma_{\chi i_n}}\right)^2}$$

Equation 3: Formula used to calculate a continuous rotamer score from the central values lists; where i is the index that enumerates the recognised rotamers for a residue, N is the number of chi dimensions applicable to a particular residue, $\chi_n$is the n^{th} chi angle of the residue, and $\mu_{\chi i n}, \sigma_{\chi i n}$ are the mean and standard deviation of chi angles of the indexed rotamer, respectively.

By applying this method, the load time on start-up was negligible (on the order of milliseconds) as was the memory usage (on the order of kilobytes). Lookup times were also extremely short, averaging 0.0203±0.0002 milliseconds per residue in testing. Consequently, this method was chosen for implementation in the *metrics* module. The central values data were repackaged and wrapped with their corresponding functions in a dedicated submodule, named *rotamer*.

2.2.4.5 Others

The last two coordinates-based metrics of interest were atomic clash score and hydrogen bonding satisfaction.

The atomic clash score is a measure of the number of pairs of unbonded atoms in the model that are infeasibly close to one another; in other words, pairs of atoms that could not in reality physically be so close to one another without some electrochemical repulsion driving them apart. This is an inherently sound metric for model quality, and has long been implemented in the MolProbity validation software (1).

Hydrogen bonding satisfaction is a metric that attempts to measure the number of hydrogen bond-conducive geometries in a model; that is, the number of pairs of residues that are aligned in such a way that a hydrogen bond would theoretically form between them. In proteins, a hydrogen bond most commonly occurs when a carbonyl-oxygen atom shares electron density from a lone pair of electrons to the σ* antibonding orbital of a nearby NH group (often described as the NH group *donating* a proton to the oxygen *acceptor* atom). Intrapeptide hydrogen bonds play a crucial role in conferring structural stability in proteins (90), and both theory and

experiment suggest that the likelihood of finding an unsatisfied intrapeptide hydrogen bond in a protein is very low (91). Hence, it can reasonably be assumed that a model with a higher frequency of hydrogen bond satisfaction is more likely to be accurate than one with a lower frequency.

There was a common obstacle with implementing either of these two metrics: both of them require the coordinate data of all the hydrogen atoms in the structure to be present in the model data. Most structure determination methods are not sensitive enough to be able to detect hydrogen atoms, because of their small size. Consequently, most atomic models do not include the coordinates of hydrogen atoms. Thus, a prerequisite to calculating either of the aforementioned metrics is to first calculate the positions of all the structure's hydrogen atoms, and add them to the model. This would have been such a time-consuming task that it was decided that these metrics would not be implemented in the *metrics* module at this stage of the project.

2.2.5 Density fit analyses

2.2.5.1 Background

With the model-only analyses essentially complete for the time being, the project progressed to density fit analyses. The goal of such analyses is to determine the extent to which a model agrees with the electron density map calculated from the experimental data. The first step in applying some measure of density agreement was to decide what metric should be applied, with the requirements again being: 1) low initial overhead time at start-up, and calculation time per-residue; 2) to be non computational-resource intensive; 3) to be accurate to a satisfactory level.

Broadly speaking, there are two ways to calculate a density fit metric. The first way involves calculating an *observed* electron density map from the reflection-data file, and a *calculated* electron density map for the model file, then applying these maps to calculate metrics. The most common metrics calculated from these two maps are the metrics of RSCC (*Equation 4*) and RSR (*Equation 5*), both of which are determined by comparing the differences in electron densities at a number of discrete points within a local area of the model, a fairly computationally expensive process. Both of these metrics have been demonstrated to express individual biases (48). These maps can also be used to calculate a difference density map (*Equation 6*), which is used in molecular modelling software to visualise areas where the modelled electron density is

incompatible with the experimental data. Typically, these areas are rendered as coloured isosurfaces, with green denoting positive density (where some experimentally evident electron density corresponds to empty space in the model) and red denoting negative density (where some modelled electron density corresponds to empty space in the experimentally-derived map).

$$RSCC = \frac{\sum[(\rho_{obs} - \langle\rho_{obs}\rangle) \cdot (\rho_{calc} - \langle\rho_{calc}\rangle)]}{\sqrt{\sum(\rho_{obs} - \langle\rho_{obs}\rangle)^2 - \sum(\rho_{calc} - \langle\rho_{calc}\rangle)^2}}$$

Equation 4: RSCC; where ρ_{obs} is the density of the *observed* map, and ρ_{calc} is the density of the *calculated* map.

$$RSR = \frac{\sum|\rho_{obs} - \rho_{calc}|}{\sum|\rho_{obs} + \rho_{calc}|}$$

Equation 5: RSR; where ρ_{obs} is the density of the *observed* map, and ρ_{calc} is the density of the *calculated* map.

$$diff = (m|F_{obs}| - D|F_{calc}|) \cdot e^{2\pi i \phi_{calc}}$$

Equation 6: Difference density calculation; where m is an estimate of the cosine of the error in the phase, F represents the amplitudes, ϕ_{calc} represents the calculated phases, and D is a scale factor used to account for the arbitrary difference in the scale of the amplitudes between the observed and calculated data.

The other broad way to calculate a density fit metric completely eliminates the need to calculate an electron density map for the model. A fit score can be calculated based solely on the electron density values extracted from the observed electron density map, by extracting the electron density values at the coordinates corresponding to the atoms in the model file. This method is extremely fast; not only is there no need to generate an electron density map for the model, but there are fewer queries made as to the electron density of the observed electron density map (one query per atom) than there would be if accurately calculating RSCC or RSR.

2.2.5.2 Existing implementations

Of the two aforementioned broad scoring methods, the latter was decidedly more suitable for implementation in this project. Research into the source code of other validation software revealed examples of where this approach is taken. The first software investigated was Coot, which was chosen because it too uses the Clipper library for many of its underlying calculations. The Coot source code (92) revealed that the density fit score given for each residue is just the average of the atomic density scores (the density in the observed map at the atom's coordinates) weighted by atom occupancy.

To uncover more source code implementing similar density-scoring methods, public repositories were searched for keywords of the Clipper library. Of particular interest was a density-scoring class written for the program SLOOP, which was eventually incorporated into Coot. As written in the comments, the class scores protein fragments "based on the position of the [densities] in a cumulative density distribution based on a Gaussian distribution derived from [the] mean and variance of the map". The process by which this is performed is as follows:

1. Query the map for density values at the coordinates of the fragment's main-chain atoms (N, Cα, C)
2. Convert the density values into z-scores, by subtracting the mean map density value, and dividing by the standard deviation of map density values
3. Convert the z-scores into probability values, by applying the Gaussian distribution cumulative distribution function
4. Take the natural logarithm of these density z-scores
5. Return the sum of the resulting log values

Because the individual point probabilities are converted to log probabilities, their sum is a log-likelihood value for the fit of the fragment as a whole (to some portion of the electron density of a map). This, of course, differs to the Coot density fit score in that the map densities at each atom are converted to log probabilities before being summed.

The finally-investigated source code was a Python script written by Paul Bond (93), which implemented a slightly different mechanism to calculate a density fit score. The script was written as a plug-in for Coot, to automatically 'prune' away the side-chain atoms of residues that were deemed poorly modelled by a machine learning algorithm, by removing them from the model. In this script, each *Atom* object has a property *density*, to which is assigned the map's

density at the coordinates of the atom, but also the property *density_norm*, which takes the values of the *density* value, and divides it by the atom's proton number. This was done to enable the comparison of the electron densities of atoms with differing proton numbers. The script calculates a few different measures of residue density fit, and in all of them, the normalised value is used, as opposed to that of *density*. For each set of atoms, a special z-score was calculated, where in place of the mean and standard deviation, the median and median absolute deviation are used, respectively. This method was employed because, the oretically, it should be more statistically robust to the range of various distributions of electron densities that might be found in an electron density map, and empirically, it was found to yield better results in training the machine learning model than using a standard z-score.

2.2.5.3 Building a scoring method

To decide a suitable scoring method for this project, the variable elements discovered in the existing scoring implementations would be systematically addressed one by one.

The first variable to address was which specific atoms of each residue would be scored; for example, just the main-chain atoms, just the side-chain atoms, or every atom in the residue. After seeking advice from experienced crystallographers, it was decided that the most useful way to present the scores would be to score the main-chain and side-chain separately.

The next point to address was the method of electron density grid point recall to be adopted: specifically, whether to use interpolation (either linear or cubic) or the simpler and more efficient, but less accurate, solution of closest-point approximation. Given the overarching goal of the project, this was a variable for which the best choice was almost certainly the most efficient one, so the method of closest-point approximation was selected.

Another factor to decide on was whether or not to apply atom density score normalisation like that applied in the Coot side-chain pruning script, the point of which is to make density scores comparable across atoms of various sizes. The only situation in which it makes sense not to apply such normalisation is one in which all of the atoms being assessed are always going to be of similar sizes, as in the case of the fragment scoring function from the SLOOP source code, where only nitrogen and carbon atom densities are being queried. In this case, atoms of various sizes would be involved. Hence, it would make sense to apply a normalisation technique.

Next to decide was how to process the normalised density scores, if at all. In other words, whether to treat each atom's score as, for example, its normalised density score, a normalised density z-score, a probability, or a log-likelihood score. The log-likelihood score was the most mathematically sound choice, so that was selected.

Along the same lines: given that the objective of the software is to determine a density score for each residue in a model, a decision needed to be made as to how to process the array of individual atom scores into a residue score. For example, the residue score could be the sum, the arithmetic mean, or the geometric mean of the array of atom scores, either weighted in some way, or not. Because log-likelihood scores were being used to score each atom, it would make most sense mathematically to use the sum of those values as the residue score, since in probability theory, the overall log-likelihood of intersection equals the sum of the log-likelihoods of the individual events (given independence).

Therefore, these choices were combined to create a suitable fit score equation.

$$Score_{residue} = \sum_{n=1}^{N} -\log\left[NormCDF\left(\frac{\rho_{atom} - \mu_{\rho_{map}}}{\sigma_{\rho_{map}}}\right)\right]$$

Equation 7: Formula used to calculate density fit score for an individual residue; where N is the number of atoms in the residue, $NormCDF$ is the cumulative density function of the standard Gaussian distribution, ρ_{atom} is the electron density at the coordinate of a particular atom, normalised by its proton number, and $\mu_{\rho_{map}}, \sigma_{\rho_{map}}$ are the mean and standard deviation of the map electron density respectively.

2.2.5.4 Implementation

The actual implementation of the density fit score was packaged in a dedicated submodule of the *metrics* module, called *reflections*, named as such because at this point, only crystallography data was to be supported. This submodule introduced the *ReflectionsHandler* class, an instance of which would be initialised with the program and perform all the pre-processing on the reflection data to allow the main thread to query it for electron density data, as part of the flow of the metrics calculations. The flow of this class is outlined in *Figure 22*.

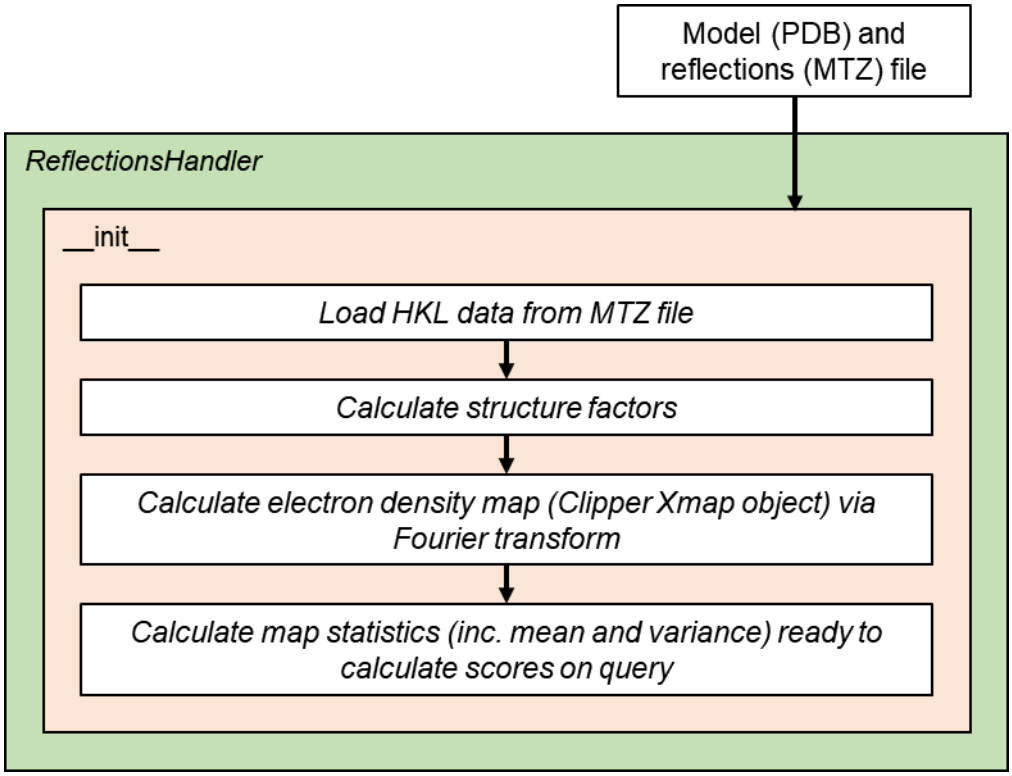

Figure 22: Flow of the constructor of the ReflectionsHandler object.

2.2.5.5 Complications

A few difficulties were encountered when actually writing the code for this class. All were a result of missing or broken bindings in the Clipper-Python package. It should be noted that at this stage of the project, there were a number of available versions of the Clipper-Python package. A few aspects differed between these versions, including:

1. Operating system or Python version compatibility (Python 2.7 vs. Python 3.x)
2. Missing or additional functions and methods
3. Some object properties being implemented as methods, rather than as variables, or vice versa
4. Inclusion and functionality of the Intel Math Kernel Library (MKL)

The first difference is only significant in that this necessitated development using different versions of the Clipper-Python library under different environments, subjecting development to different combinations of the latter three differences. The fourth is especially significant: use of the Intel MKL is a requirement of some variants of the Clipper C++ library to perform the fast

Fourier transform (FFT), which is used to calculate electron density maps from reflection data. This library would throw hardware incompatibility errors under a number of environments, making it impossible to calculate any density fit-based metrics. Fortunately, most versions of the Clipper-Python package use the free FFTW library (94) rather than the Intel MKL, which was never problematic.

These problems were resolved by writing a submodule containing a long conditional import statement with nested try-catch statements, to be imported by those files that needed access to Clipper-Python. In this way, the *metrics* module could automatically determine the most suitable import method, and then assign a global variable with a value signifying which version of the Clipper-Python library had been imported; then, each time Clipper is called in the code, a conditional statement was used to ensure the call is suitable for the version that had been imported.

In addition to the surmountable problems posed by these differences, some Clipper methods were completely inaccessible in any usable version of the Clipper-Python bindings, including both the linear and cubic interpolation methods of the *Xmap* class. As such, point queries from the map were restricted to returning the density at the closest point in the grid, resulting in less accurate results, but with the upside of slightly faster queries. Fortunately, as mentioned earlier, closest-point approximation had been selected as the most suitable method for this project.

Another important decision when designing the class was the method used to extract data from the MTZ reflection-data files. A diffraction experiment produces a list of reflections; each reflection is indexed by its Miller indices (H, K, L), and has some number of associated columns of data, which come in pairs. All MTZ files will include at least the reflection intensities and their standard deviations (I, SigI). From the intensities, the native amplitudes and their standard deviations can be calculated (F, SigF); every MTZ file will also have these columns. The other column pair of interest is the structure factors, which comprise the calculated amplitudes and the calculated phases (F, Phi). These are calculated using the amplitudes and the model file, and are required to produce an electron density map. The structure factors may or may not be present in any given MTZ file; in the PDB-REDO MTZ files, for example, the structure factors have already been calculated, and those columns are included. Therefore, when presented with an MTZ file that includes structure factors, the *metrics* module could either choose to recalculate them itself, or use those from the file (since the two may differ). In the end, it was decided to recalculate the structure factors for every file, regardless of available columns. This ensures

comparability between results generated from MTZ files from various sources, at the cost of very slightly increased initialisation time for the *ReflectionsHandler* object.

2.2.6 Percentiles library

Since the purpose of the interface was to show multiple different validation metrics concurrently, being able to express metric values on a normalised scale was critical. If this were not done, and the metric values were left as absolute values, with arbitrary, incomparable units, it would be misleading to present them alongside one another and attempt to infer correlations between them. The distribution of values for each metric cannot be assumed to be identical, either across a single structure or a population of structures. Hence, with the *metrics* module suitably complete, the next task was to produce a percentiles library to accompany it.

Originally, the percentiles library was to be generated by calculating validation metrics for every residue of every structure in the PDB-REDO database. In order to do this, a script was written that imported the *metrics* module, applied it to each set of files in the local PDB-REDO directory, and stored the generated metrics values to one large variable, which would then be serialised for later analysis.

A significant problem was encountered in testing this script. For a variety of reasons, a small number of PDB model files could not be loaded by the MiniMol class. Because of the error handling applied in the Clipper-Python module, this error was uncatchable, and would halt the execution of the entire Python script. Two easy fixes were tested. The first was to add a routine to read through every PDB file and predict whether or not it would be read successfully; however, that ended up being unreliable; the second was to have a shell script launch and supervise the metrics generation script, waiting for it to crash and restarting it after appending the ID of the offending model file to a list of incompatible model IDs to be ignored. This ended up being too slow.

To speed up the process, the analysis program was parallelised using Python's multiprocessing library, which enables multithreading across individual child processes, as opposed to just multiple threads on the same physical core. With the 8-thread, 16-core processor being used for the analyses, 16 worker threads were spawned, with one worker thread sharing a logical core with the mostly-dormant main (controller) thread. The script was structured in this way so that the main thread could monitor the status of all the worker threads. Thus, if an uncatchable

Clipper error caused any of the workers to crash, the main thread could remove the halted thread from the worker pool and spawn a new thread to replace it.

The resulting array of metrics values contained between 14-17 million entries. Arrays different in length due to the fact that some metrics return null values for some residues; for example, residues at the start and end of a chain do not have Ramachandran conformations, and glycine and alanine residues do not have rotamer conformations. The distributions of metric values were visualised in histograms (*Figure 23*).

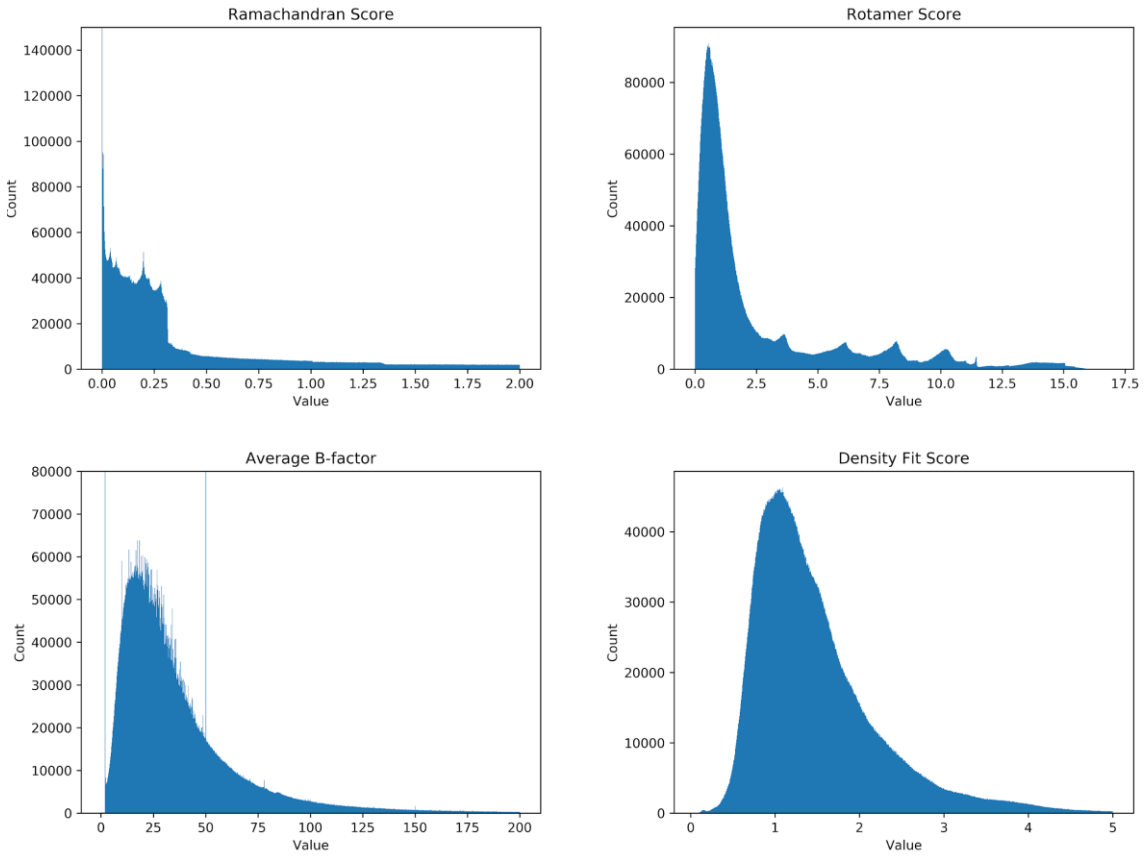

Figure 23: Distributions of metrics values. The average B-factor and density fit score distributions are smooth and unimodal. The spikes in average B-factor at 0 and 50 are likely a result of default values in modelling or refinement software. Conversely, the Ramachandran score and rotamer score distributions are not unimodal or smoothly distributed. In the case of Ramachandran score, this was attributed to the effects of constraints and restraints applied manually or by refinement software and, in the case of rotamer score, to a mathematical side-effect of oversimplification in treating all chi dimensions as perfectly Gaussian distributions.

Following in the footsteps of other programs that compare a model's metric values to wider distributions of values from other structures (1,70), it was decided that to produce percentile data from a more representative sample, the percentile values presented to the user should be based on distributions of metric values from a sample of structures of similar quality. This could be done by binning the calculated metrics values via some global quality indicator, the most commonly used of which being resolution and average B-factor.

The first thought was to bin metric values by *both* resolution and B-factor, to produce the most precise binning possible. Before this was attempted, the correlation between resolution and B-factor was assessed, to ensure that binning in both dimensions was warranted (*Figure 24*). Unsurprisingly, there was some correlation between the two metrics. However, with a low R^2 value of 0.51, it was decided that the two-dimensional binning might still be worthwhile, so it was implemented.

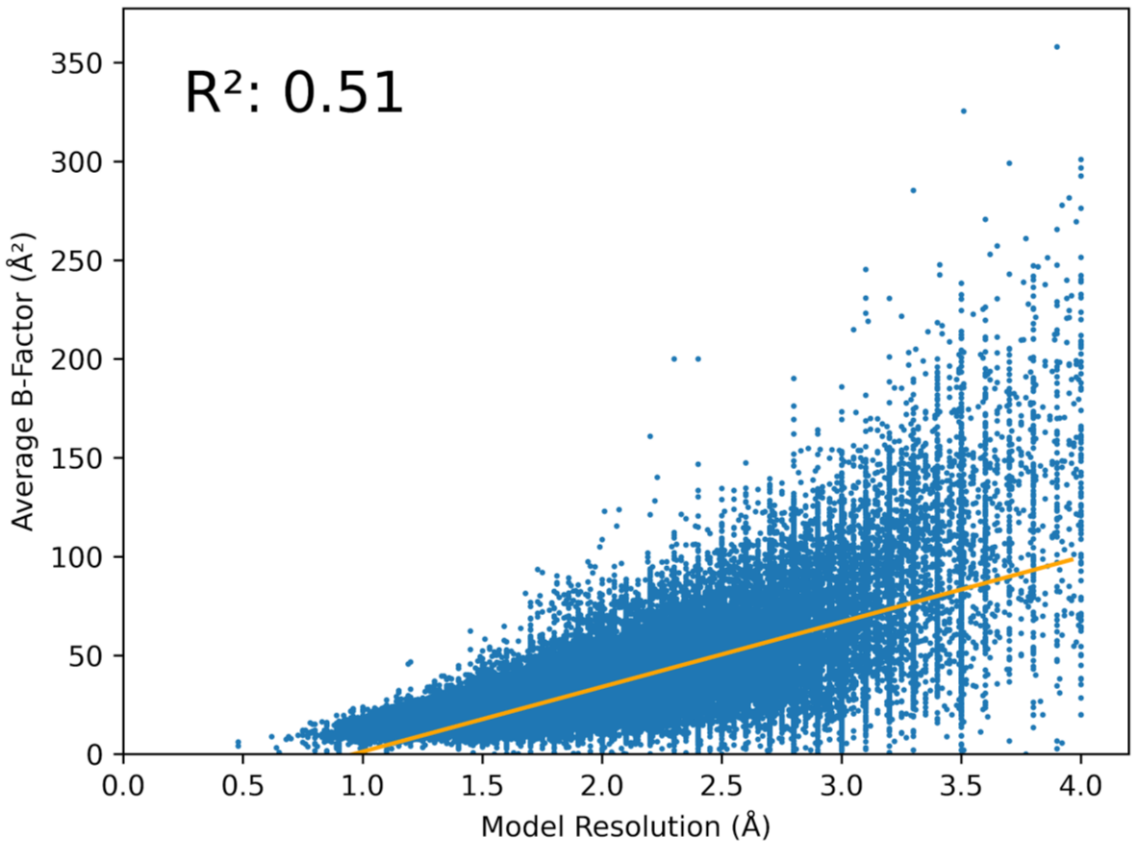

Figure 24: Correlation between resolution and average B-factor for all models in the PDB-REDO database. The orange line is the line of best fit.

To implement the two-dimensional binning, the metrics generation script was modified such that prior to analysis, every model's resolution and mean B-factor were ascertained, and stored in a dictionary. The 10th, 20th, ... 80th, 90th percentile values were determined for both resolution and mean B-factor, which were used as threshold values to produce two sets of ten bins (<10th, 10-20th, ... 80-90th, >90th). Then, metrics values were calculated for each model. Rather than storing the values for each metric in a one-dimensional array, as before, they were instead stored in a three-dimensional array, where the first two dimensions were both of length ten, and corresponded to the resolution and B-factor bin indices. Each model's metric values were placed in the corresponding bin.

The first analysis to be performed on the binned metric data was to see how the distribution of each metric varied with each of the two bins. This was done by flattening each metric's three-dimensional array into two two-dimensional arrays, and plotting the distribution for each metric against the bin index for each of the two bin dimensions (*Figure 25*).

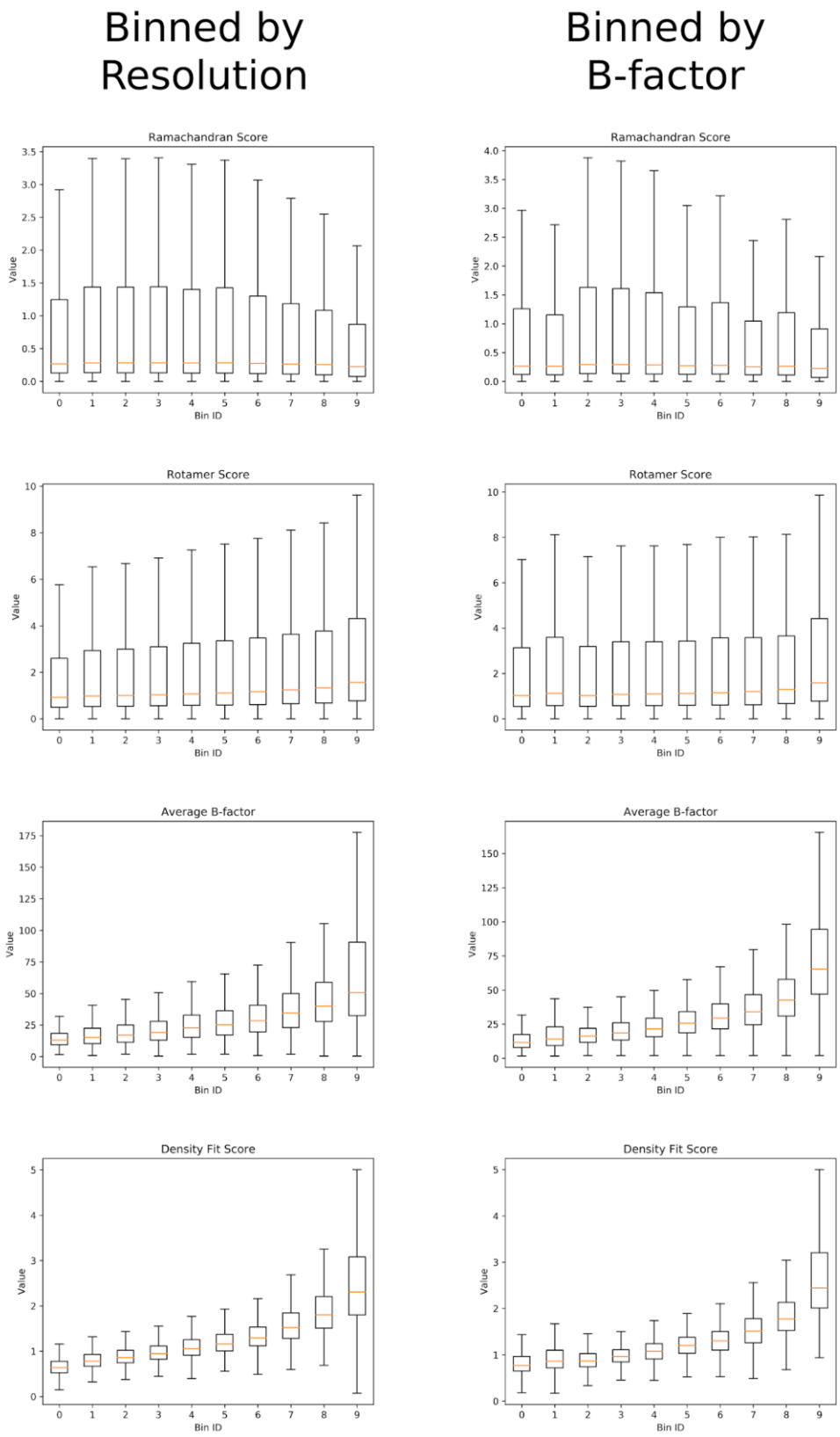

Figure 25: Graphs illustrating how metric value distributions vary across bins divided by resolution (left) and mean B-factor (right).

Since the trends were so similar across both bin dimensions, it was decided to collapse the binning down to just one of the two dimensions. Of the two, it made the most sense to use resolution, given that it was the only one technically independent of all the individual residue metrics being assessed.

Some of the structures in the PDB-REDO database were deposited many decades ago, before the establishment of modern model-building processes such as the application of constraints and restraints (*Figure 26*).

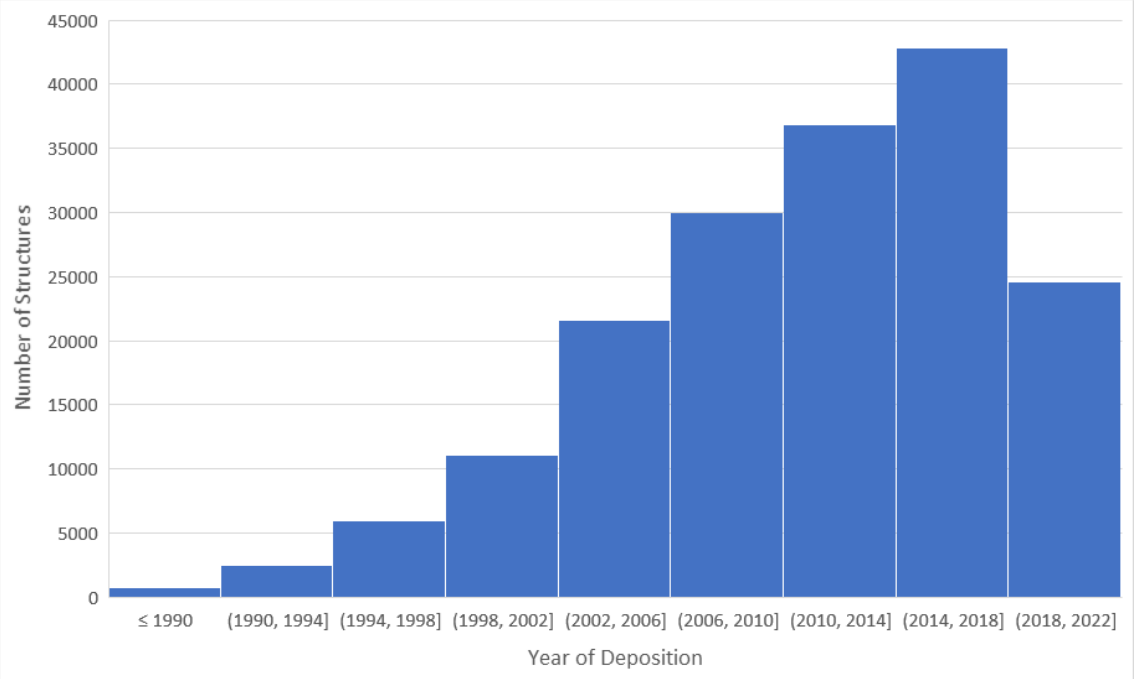

Figure 26: Number of structures deposited in the PDB over time. Data from wwPDB, accurate as of December 2020.

To refine the sample pool and make it more representative of the standard of quality that is expected of new structures, the sample pool was restricted only to structures deposited after the year 2010. The changes resulting from this restriction were surprisingly minimal. The overall metric value distributions were almost identical, and the resolution bin thresholds only changed slightly, as illustrated in *Figure 27*. Despite the triviality of these changes, the year restriction was kept, since the sample size was still very large.

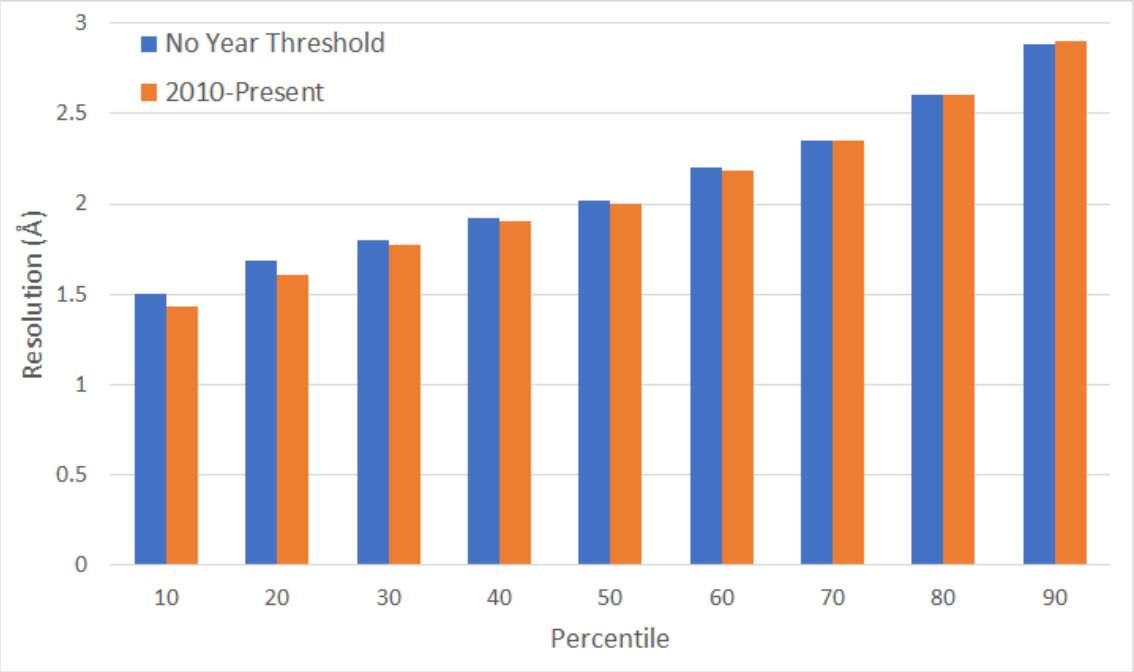

Figure 27: Analyses of the change in resolution bin thresholds after sample pool refinement by year of deposition.

Satisfied with the obtained distributions of the metrics values, the *percentile* function of the NumPy library (95) was used to calculate percentile values for each resolution bin of each metric. These were to be included with the library as comma-separated values (CSV), since it was decided that data that had to be included with the final package should be included in a human-readable format where viable, to maximise interpretability by the end user. Values were calculated at each integer percentile in the range [1-99]. This way, a given calculated metric value could be assigned a percentile number in the range [1-100].

These data were implemented in a submodule named *percentiles*, which implemented functions to: 1) load the percentiles data CSV file; 2) determine a model's resolution bin; and 3) return a percentile for a given metric value and resolution bin. The submodule was tested thoroughly with calculated and synthetic data.

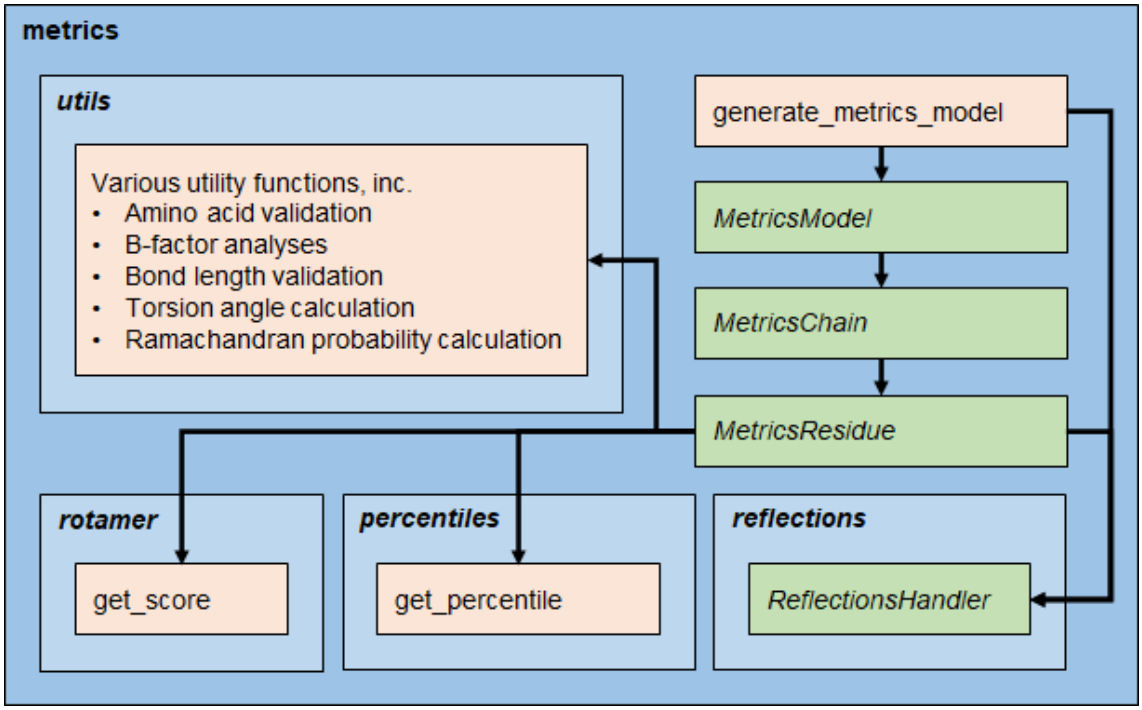

Figure 28: Structure of the metrics module following its initial development.

2.3 Combining the two modules

2.3.1 Initial adjustments

At this stage in the project, there were two independent Python modules: the *interface* module, and the *metrics* module. The next task was to combine these as two submodules in a single package, whereby they could be co-utilised by a single call to the parent package.

In principle, this was straightforward. The only job involved was to change the input datatype of the interface module from the *SyntheticProtein* class created before, to some sort of primitive type that the metrics module could be modified to produce. The data required for the interface module was: 1) the names of the metrics to be plotted; 2) the metric and percentile values for each residue; and 3) the shape of the data, i.e. the number of models, the number of chains per model, and number residues per chain. It was decided that the simplest way to package these data would be that the metric names be included as part of a package-wide definitions file, and for the metric values to be packaged in a multidimensional array of length-2 tuples (metric value and percentile value). This way, the length of the arrays would intrinsically imply the shape of the data. In addition to simplicity, the reason this format was chosen over some custom class

was so that the *metrics* and *interface* submodules could be used independently by the end user, without much difficulty. i.e., the resulting package could be used just to generate metrics values for the user to use in custom code, or, the charts and HTML report could be produced using data from a different source.

Some differences between the synthetic and real-world data still needed to be accounted for. For example, the synthetic data class always elicited data in the range [1-100]. Even on a percentile scale, the distribution of values in real-world data could be very different. Very good or very poor-quality structures may elicit, for example, distributions of percentile values in the range [5-30] or [70-95], respectively. Given that the purpose of the chain-view chart was to draw the modeller's attention to the worst parts of each individual chain, the range of each axis should correspond to the individual chain's distribution of the metric values represented by that axis. Otherwise, the residue metric values for a poor-quality model would appear to be poor across the board, making it difficult to ascertain the areas of *particularly* low quality within that chain, and vice versa for a high-quality model. Thus, the chain view chart generation function was adapted to calculate the baseline value (from which deltas would be calculated) as the mean metric value across all residues of a pair of corresponding chains across both model iterations, with the resulting squared-difference values normalised accordingly.

Another difference to be accounted for was that, as noted in *Section 2.2.6*, real-world metric values could be null in some cases, which was not so for the synthetic data. To cope with this in the short-term, the chain-view generation function was modified to plot small circular null-point markers at points where the line-graph could not be plotted. This would theoretically leave a break in the line graph. The most accurate way to resolve this would be to leave it broken, and then start a new plot after the null point. This meant the structure of the chart would have to be modified to potentially allow numerous line elements per axis. However, since this was only to be a temporary measure, a simpler route was taken: at each null point, the line graph would be plotted at the axis, and carry on as one continuous line element.

2.3.2 Testing with real-world data

With the differences accounted for, test validation reports could be produced from real-world metric data, to assess the efficacy of the validation package as a whole. A selection of test structures was made, to cover a variety of attributes, as shown in *Table 3*. For each of these

structures, data was obtained for both before and after a refinement step (or re-refinement, in the case of the PDB-REDO structures).

Structure Code	Protein Name	Source	Model Properties	Citation
1vme	Flavoprotein	Buccaneer	2 medium length chains; incompletely refined	(96)
2ask	Artemin	PDB-REDO	2 small chains; re-refined	(97)
2a0z	Toll-like receptor 3	PDB-REDO	1 large chain; re-refined	(98)

Table 3: Structures used to perform the first tests of the fully assembled package.

2.3.3 Chain view changes

The first tests were performed, and the resulting chain-view charts (*Figure 29*) were reviewed.

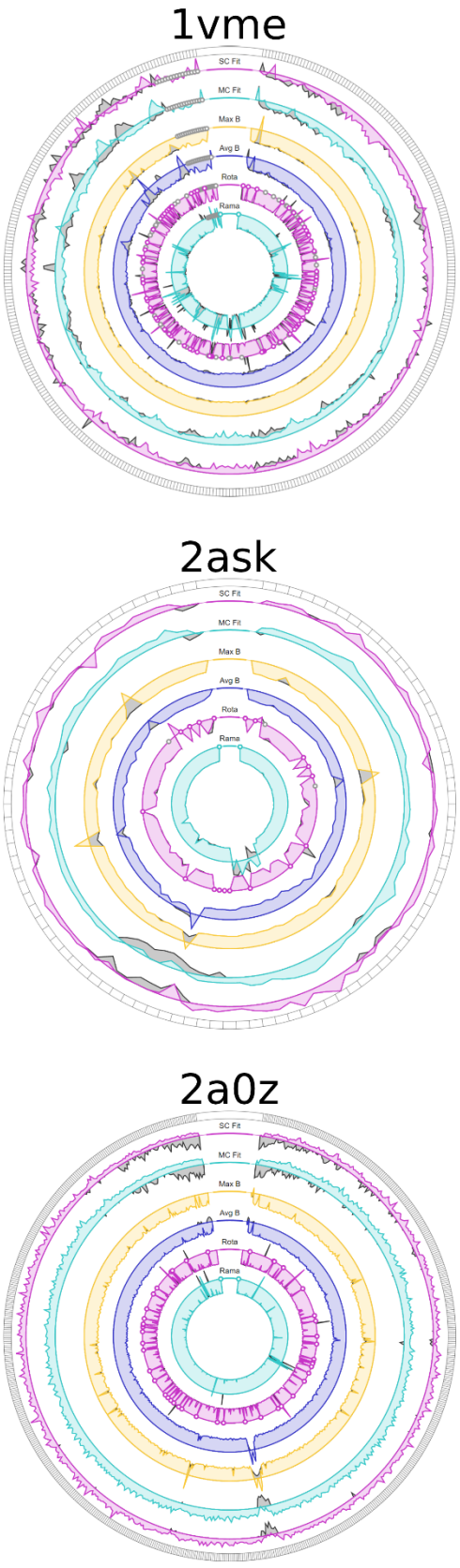

Figure 29: First chain-view charts produced by the fully assembled package. These charts were generated using chain A of each of the three models.

2.3.3.1 Metric polarity synchronisation

There was a clear inverse correlation between B-factor and electron density fit quality, due to the fact that B-factor values are better when low, and density fit scores are better when high. Therefore, with metrics plotted in their native polarities, it is not immediately apparent if a given peak or a trough represents a good or bad value. It was decided that the only logical way to mitigate against this would be to ensure that every axis of the chain-view chart followed the same 'goodness polarity', the most sensible direction being for bad values to be represented by troughs, pointing inwards, and good values represented by peaks, pointing outwards. To implement this, the package needed to have some definition of the polarity of each metric. This was added to the package-wide definitions file, alongside the metric names. Once this was done, there were two potential routes for actually applying the polarities: the first was to let the chain-view chart function just flip an axis' values where necessary, and the second was to implement the polarities in the percentile determination function itself, such that the 100th percentile would correspond to a smaller value, in the case of metrics such as B-factor. Ultimately, the latter option was chosen, if not just for the fact that it was the more logical choice, but because it would also solve the problem of incompatible percentile polarities for the residue-view chart, which would otherwise suffer in a similar way to the chain-view chart. The chain-view charts were regenerated, and the results are shown in *Figure 30*.

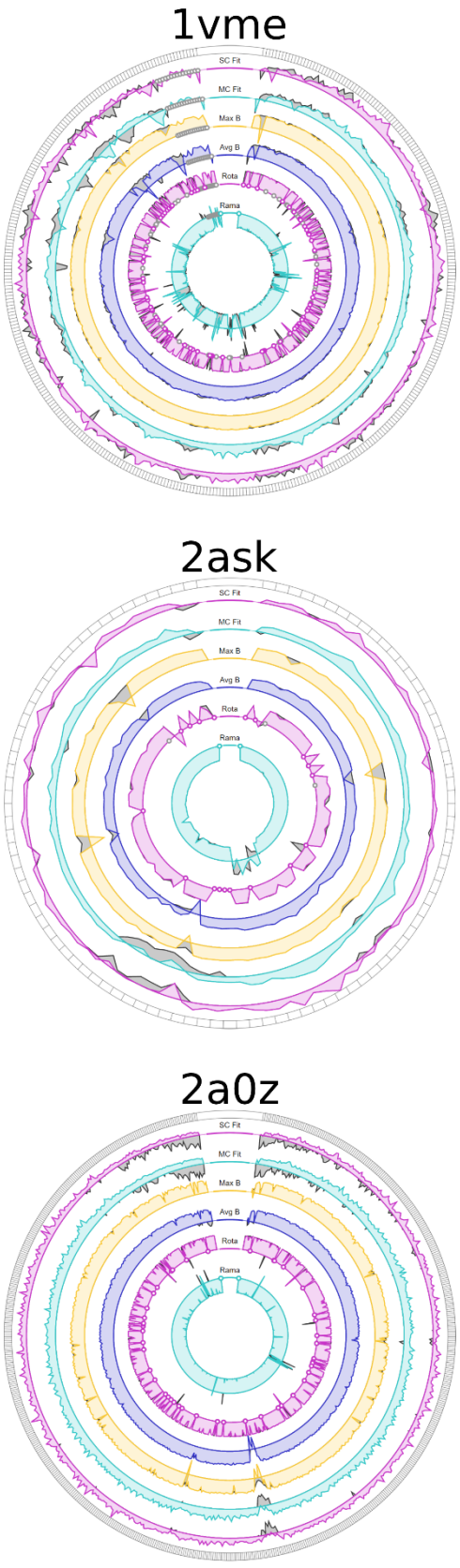

Figure 30: Chain-view charts after polarity-correction measures were added. These charts were generated using chain A of each of the three models.

2.3.3.2 Sequence alignment

Another problem, only apparent in the chart of the Buccaneer model, was that the data from the previous model iteration (represented by the grey shaded area) appeared to be misaligned with the data from the latest model iteration. In addition, the final few residues of the previous iteration appeared to have null-point markers in every single metric. Unsurprisingly, manual review of the model files revealed that between these two iterations of the model, the number of residues differed. The refinement procedure had removed some amino acids from the model file for the latest iteration, from the very start of the chain. Because, at this stage, all provided iterations of a model had been assumed to have the same configuration of chains and residues, they were automatically aligned by their index. As a result, the chain in question had been misaligned. To prevent this from happening, the residue sequences for each chain would have to be aligned before generation of the chain-view chart. This could be done in one of two ways: either based on residues' amino acid type types, or their sequence number. The former was chosen for the reason that a refinement program may decide to renumber residues accordingly after it deletes some of them, whereas an amino acid code is far less likely to change, and sequence-alignment algorithms are mostly robust to a few such changes occurring, providing the number of unchanged residues is high enough. For the sake of simplicity and robustness, pairwise sequence alignment (PSA) was chosen over multiple sequence alignment (MSA). The downside of this decision was that it would eliminate the possibility of comparing more than two model iterations in one graphic, but it was decided that if this ended up being a desirable feature, MSA could be added in later. A Python implementation of the Needleman-Wunsch PSA algorithm was written and added to the *utils* module. Then, the *build_report* function of the *interface* submodule was modified to call, for each chain, a residue-alignment function, which would perform PSA, and then for any gaps found, insert a null (*NoneType*) residue into the data array passed to the chart-generation functions. These null residues would then be handled by the chain view chart-generating function in the same way as a null metric value: by adding a set of null-point markers across every axis for that residue's sector.

For the sake of thoroughness, a procedure was also added to align the chains of each model iteration, in case entire chains become deleted from the model file. Unfortunately, the procedure for chain-matching is less robust than for residue sequences, because pair-alignment cannot be performed in the same way; the sequence of chains is much shorter, and in the same way that individual residues might not retain the same sequence number, an individual chain might not retain the same ID code following the removal of another chain.

After implementing these changes, the chain-view chart for the Buccaneer model was regenerated and the two iterations appeared properly aligned (*Figure 31*).

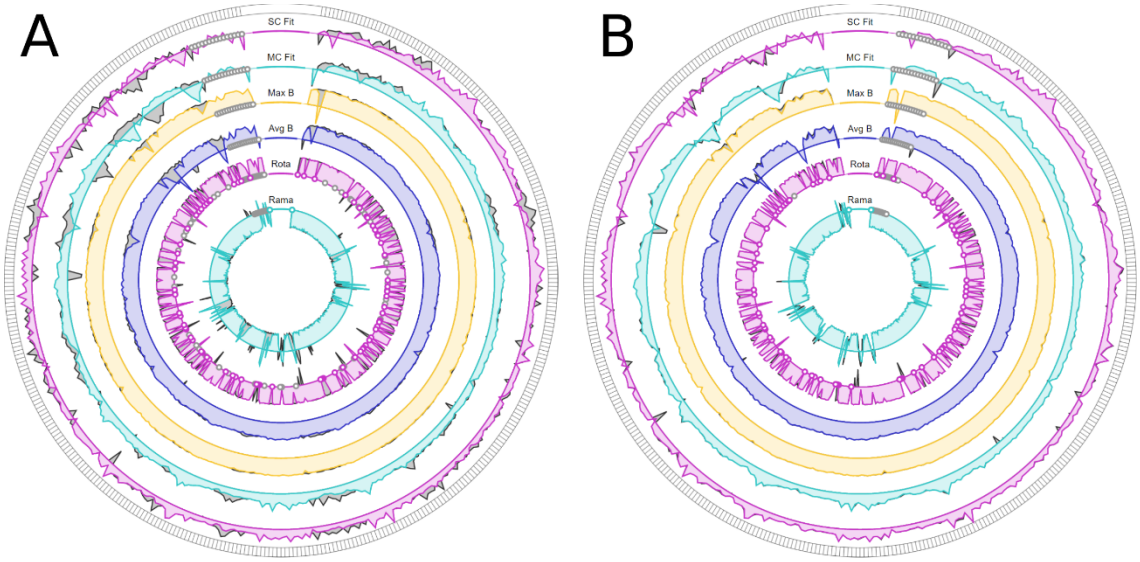

Figure 31: Chain-view chart for chain A of the Buccaneer model (1vme) before (A) and after (B) after alignment measures were introduced.

2.3.3.3 Discrete metric implementation

The rotamer and Ramachandran scores appeared to be very erratic and confusing. This was thought to be because the distribution of likelihood scores that result in energetically favourable conformations is quite wide, so comparing the exact scores of these metrics is often meaningless. For example, a Ramachandran likelihood score of 0.5 indicates an almost equally favourable conformation to a score of 0.9, despite the large delta. But in contrast, a Ramachandran likelihood score of 0.1 indicates a much less favourable conformation than a score of 0.5, despite the same delta. Because of this, these two metrics are typically presented by other validation tools as discrete classifications (normally: *outlier*, *allowed*, or *favoured*) based on likelihood thresholds. It was decided that it would be more effective to apply the same approach in this validation tool, by implementing a new type of axis to the chain view chart that could represent discrete classifications, rather than a continuous line score.

However, before this could be done, the *metrics* submodule had to be modified to determine a classification for those metrics. In the case of main-chain conformation, this was simple; all that needed to be done was to apply thresholds to the likelihood score that was already being

generated and presented as the continuous Ramachandran score. The thresholds applied to Ramachandran likelihood scores differ between validation tools; the default thresholds applied by the Clipper library are 0.0005 and 0.01, but the thresholds applied by Coot are 0.02 and 0.002. To maintain concordance with Coot's classifications (for reasons explained later, see *Section 3.4*), the latter thresholds were applied to the *metrics* submodule.

The case of side-chain conformation was more complicated. Because the score being calculated was non-standard, based on the central values lists, exact probabilities were not directly calculable. It was possible to estimate probabilities by approximating each chi dimension as a Gaussian distribution, but testing revealed poor correlation between the estimated probabilities and reliable probabilities calculated by MolProbity, and this route was ultimately abandoned. Therefore, a different approach had to be taken. The only reliable way to obtain a likelihood score for rotamer conformation was to use the contour grid reference data. As described in *Section 2.2.4.4*, many efforts had already been made to obtain a meaningful likelihood score from these data in a suitable way, none of which had been effective. Essentially, the reason these data had previously been troublesome was that they were too large, meaning that to implement them in a suitable way would require too many system resources, and take too long to load. However, given that now the only desired output was discrete classification, and high precision was no longer required, a whole new avenue opened up for investigation.

If implemented for discrete classifications, the data could be greatly compressed in several ways. By default, the value provided at each point on the contour grids were floating point numbers, technically requiring a double-precision floating point data type (64 bits, 8 bytes) to be precisely represented, or at the very least, a single-precision floating point data type (32 bits, 4 bytes) to be stored *precisely enough*. Instead, each of the values provided in the contour grid data could now instead be represented by an integer value corresponding to the classification they represent (1: *outlier*, 2: *allowed*, 3: *favoured*). This would reduce the size of the required data type to two bits (one quarter of a byte).

Doing this reduced the size of the data quite substantially, but still the most significant factor in the size of the data persisted, which was storing the coordinates of each value. To eliminate the need to do this, the data for each contour grid could be flattened to a one-dimensional array of values, where the index of each value corresponds to the calculable index of its coordinates in a theoretical ordered array of n-dimensional coordinates (*Equation 8*). The only way this would be possible is if the data consisted of points that were equidistant from one another in every

dimension, and a value were present for every possible point. In the original data, the latter criterion was not met; many of the data points were not included in the contour grid files. If these missing points were also represented by a fourth classification (0: *unknown*), the data would become both complete and regular, and the need for storing the coordinates of each point would disappear. The addition of this new classification was still compatible with the two-bit integer type chosen previously, which can represent up to four classifications.

$$index = \sum_{n=1}^{N} \left[nint\left(\frac{\chi_n - \mathbf{X}_{n_0}}{\mathbf{X}_{n_1} - \mathbf{X}_{n_0}} \right) \cdot \prod_{m=n+1}^{N} \dim\left(\mathbf{X}_m\right) \right]$$

Equation 8: Formula used to calculate the relevant index in the compressed rotamer library for a given array of chi angles; where N is the number of chi dimensions applicable to a particular residue, χ_n is the nth chi angle of the residue, X_n is the regularly spaced array of chi values known in the nth chi dimension for that residue type, thus $(X_{n1} - X_{n0})$ represents the width of the spacing in that dimension, and $dim(X_m)$ is the number of known points in the m^{th} dimension for that residue type. $nint$ is the nearest-integer rounding function.

Therefore, for each contour grid, the missing data points were added, and the points were converted to an array of integer values. The NumPy library (95) was used to compress the Python-default 64-bit integers into 8-bit integers, and then a custom routine was used to compress each sequential set of four 8-bit integers into a single 'compressed' 8-bit integer (as 2-bit integers are not a natively supported data type). The arrays of compressed 8-bit integers were then added to a dictionary keyed by amino acid type. The dictionary was then serialised using Python's *pickle* module, and compressed using gzip. The result is a single file with a size of 147 kilobytes, a 265x reduction from the original data. A library loading function was written which applied a NumPy-based bitmasking routine to decompress the data and load it to memory on the millisecond scale.

		Chi 1 (°)					
		2	4	6	8	10	12
Chi 2 (°)	10			0.0024	0.0023	0.0022	0.0014
	20		0.0133	0.0062	0.0082	0.0043	0.0000
	30	0.0001	0.0154	0.4215	0.4882	0.0131	0.0007
	40	0.0019	0.0052	0.5798	0.5987	0.0053	0.0018
	50	0.0010	0.0033	0.0076	0.0123	0.0162	
	60	0.0013	0.0017	0.0022	0.0027		

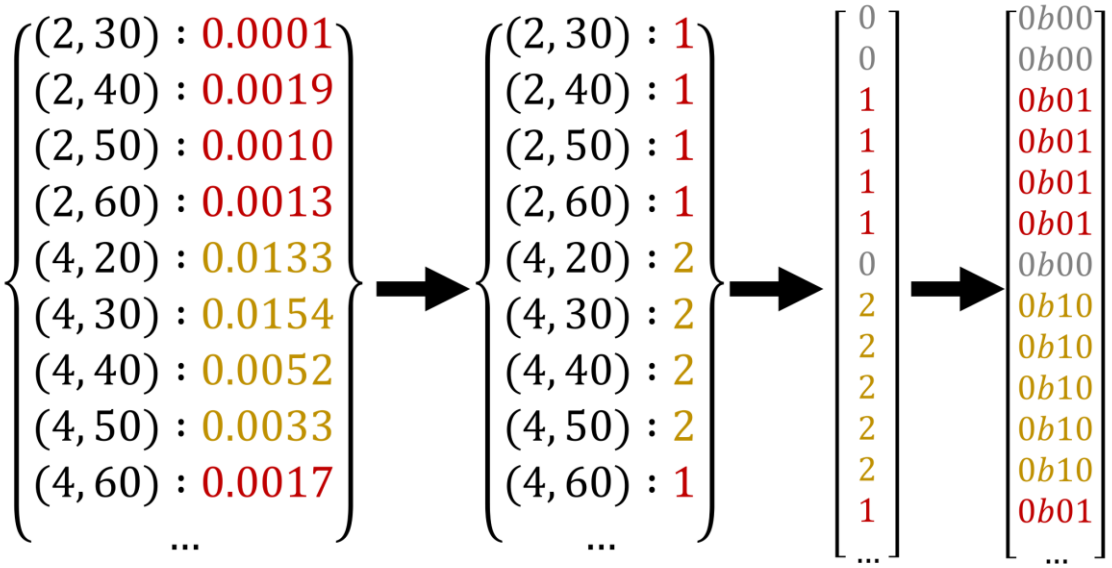

Figure 32: Visualisation of the rotamer library compression. The topmost figure shows a contour grid for a hypothetical amino acid with two side-chain torsion angles. Grid points are coloured red for *outlier* values, yellow for *allowed* values, green for *favoured* values, and grey for *unknown* - where a coordinate is not listed in the original contour grid file. The bottom figure illustrates the compression process: starting with the conversion from floating point to integer data points, followed by the type conversion from dictionary to integer array, which includes the addition of zeros to represent null data points, and finally the compression of Python integers to two-bit binary values. It should be noted that the original contour grid values are given to a much higher precision than is shown here. *From Rochira and Agirre, 2020* (75).

Satisfied with the discrete metrics calculations implemented, the next task was to adapt the chain-view chart to be able to plot them. A new definition was added to the package definitions file alongside the metric names and polarities, containing each metric's type (continuous or discrete). Then, when called, the chain view chart-generating function would use this definition

to decide the arrangement of continuous or discrete axes. The discrete axis was to be composed of a sequence of solid traffic-light colours, with red representing *outlier* or *unknown* values, amber representing *allowed* values, and green representing *favoured* values. Grey would be used to represent missing (null) discrete metric values. These colours were implemented as segments that would join to form a contiguous band of colour around the axis (*Figure 33*).

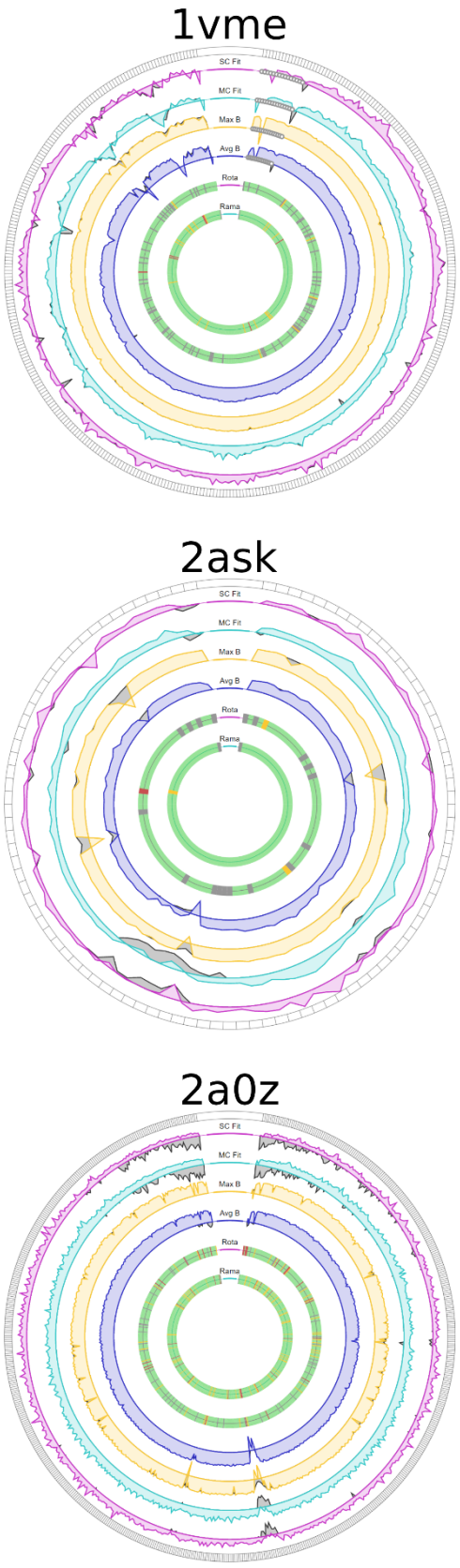

Figure 33: First tests of the discrete axes. These charts were generated using chain A of each of the three models.

2.3.3.4 Missing-residue shading

This discrete classification system had the added benefit of eliminating the need for the null-data point markers which, except for in the case of entirely missing residues, only applied to the rotamer and Ramachandran metrics. The null-data points made the chart cluttered, and could often be quite misleading. Hence, the null-data points were removed, and a new style was added to indicate missing residues: shading. For completely missing residues, the entire sector would be shaded in, to much more clearly indicate the fact that the residue is not present in the iteration being viewed. The shading would be grey for missing residues in the previous iteration, or pink for missing residues in the latest iteration. Additionally, it was felt that the grey colouring for segments that represented null data points was counter-intuitive. Because the discrete axes are mostly composed of green segments, any differently-coloured segments immediately stand out as exceptions. This is perfect for emphasising the location of outlier values, because they are areas that should draw the modeller's attention, but counterproductive if it also draws attention to null data points, which are irrelevant to the modeller. To fix this, null data points on the discrete axes would be coloured green, identically to the *favoured* values. The result of both these changes is shown in *Figure 34*.

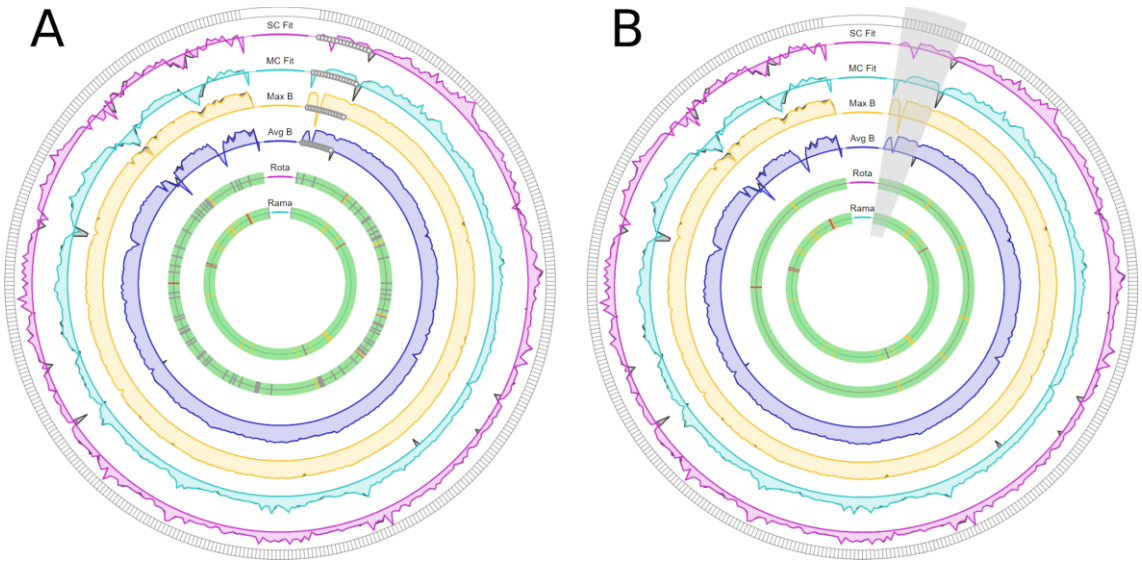

Figure 34: Chain-view chart for chain A of the Buccaneer model, before (A) and after (B) missing-residue shading was introduced and the null data point colouring was corrected to match that of *favoured* data points.

2.3.3.5 Animation

A major side effect of the addition of discrete axes was that there was now no ghosting for the discrete metrics. The current ghosting implementation was beginning to seem unsatisfactory in general. For example, there were frequently areas of the structure where the difference between the two models was masked, as a consequence of the latest iteration always being plotted over the top of the previous iteration. Because of this, and the fact that many other similar types of ghosting shading had already been tried and rejected, it was decided that in such an information-dense graphic, it would not always be possible to suitably showcase both iterations simultaneously, at least not in a way conducive to achieving the goal of making the graphics instantly comprehensible. Hence, the idea of showing two model iterations concurrently would be abandoned, and the ghosting mechanism be replaced by taking a completely different approach.

Instead, the iterations would be toggled between with a switch, positioned above the chain view. The chain view chart-generating function was modified, such that each axis had two groups of plots (one for each iteration) the first of which (previous iteration) would be hidden by default. Then, a function was written for the template interaction JS which would toggle the visibility of both groups for each axis, such that when the switch was pressed, the plots on all axes, both discrete and continuous, would jump between each of the two available iterations. To make the transition smoother and less jarring for the continuous axes, SVG's native animation support was utilised. SVG animation includes a mechanism by which all points on a polygon can be linearly translated to new coordinates, which was entirely suitable for transforming each axis' line graph plot, which were implemented as polygons. The *svgwrite* library has an *Animation* module for this purpose, which was trivial to implement, creating an animation element for each continuous axis. Each SVG animation element has a *beginElement* method, the call to which was inserted into the function called by the toggle switch. This worked perfectly without any additional changes. Different transition animations were also trialled for the discrete axes, but ultimately, all seemed to be distracting or confusing, rather than useful. Therefore, when the switch is clicked, the line graph morphs over a period of a few hundred milliseconds, and the discrete axes change instantaneously.

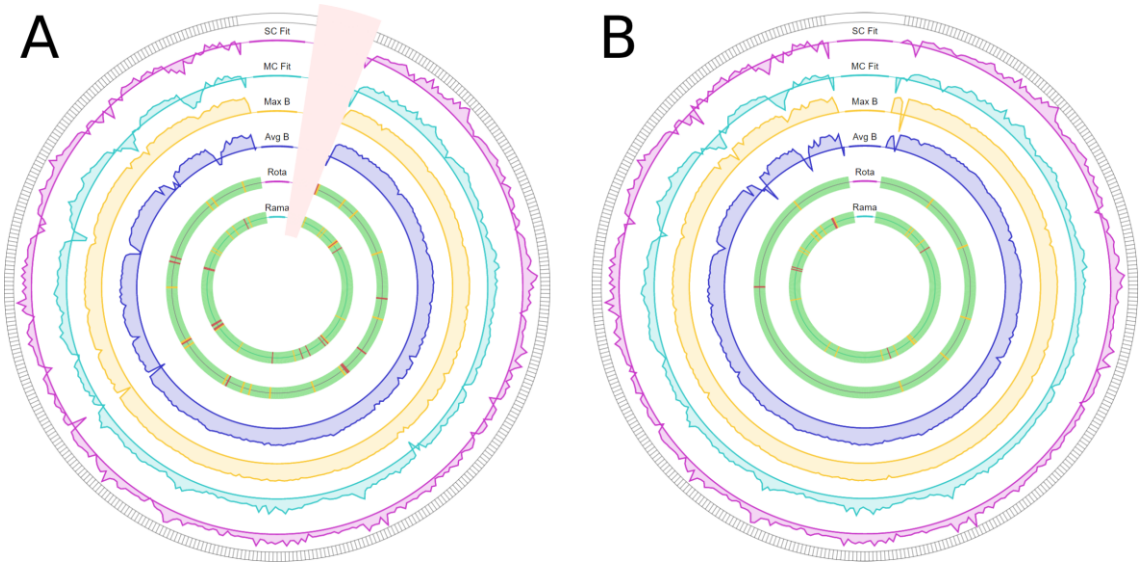

Figure 35: Both iteration chain-view charts for chain A of the Buccaneer model (1vme).

2.3.3.6 Plot formula revision

Since switching to real-world data chart generation, there had been a long-running and significant problem with the chain-view chart: it was not emphasising the worst areas of each chain. While most of each chain was appearing on the positive side of the axis, as intended, the poor areas were often being shown as minor dips, often not even crossing the axis line. Clearly, the square-difference formula, while suitable for the synthetic data, was not working for real-world data. Therefore, a new formula had to be developed.

It appeared that the main reason for the failure of the square-difference formula was that there would often be one or two very low values for a chain's set of values of a metric. These low values, when converted to square differences from the mean, would have magnitudes far greater than any other values of that metric, and thus, the normalisation process would skew the plot such that almost all of the values would fall on the positive side of the axis, even other low scores. One option to overcome this would be to apply some outlier detection before normalisation, to limit the axis bounds at, for example, the mean plus-or-minus two standard deviations. However, this would not be suitable for all metric value samples; for example, samples of especially high variance. So instead, a new plotting formula was devised. Rather than plotting the square difference from the mean, the absolute difference from the mean would be plotted, and axis transformations would be used to produce a useful plot. A Python script was written with a command-line interface that would show the result of different axis-scaling

methods on an example chain-view chart as they were input by the user. Testing with this program led to the development of a two-step axis transformation that produced excellent results. This transformation is illustrated in *Figure 36*.

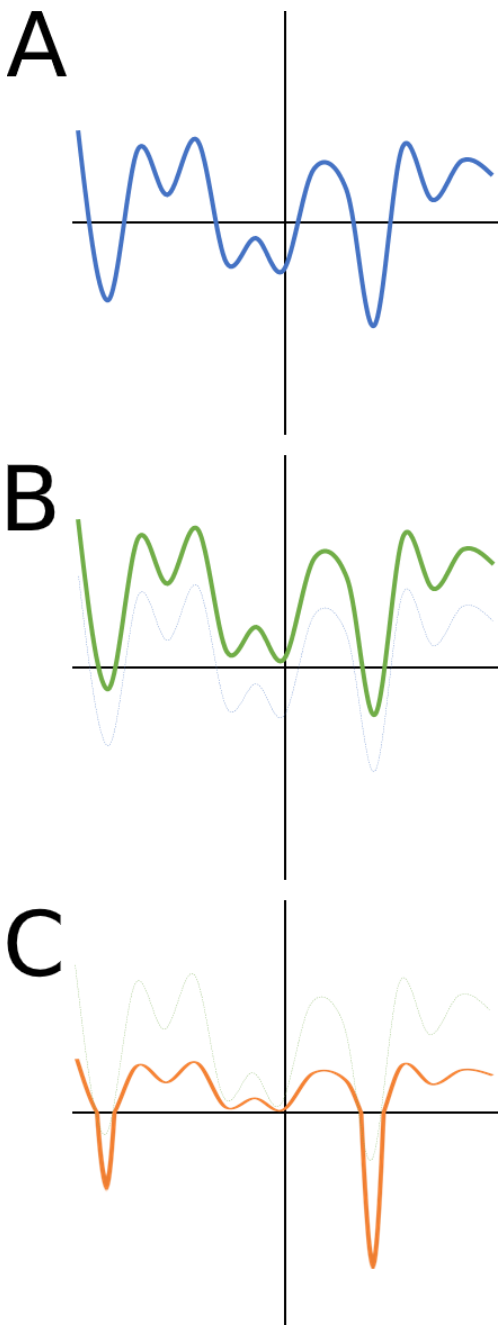

Figure 36: Step-by-step representation of the new axis scaling method. Graph A represents the raw metrics data, where each point is plotted as its distance from the mean value. Graph B represents these same points after the average negative value has been added to every point. Graph C represents the points after the positive values are divided, and the negative values are multiplied, by some constant.

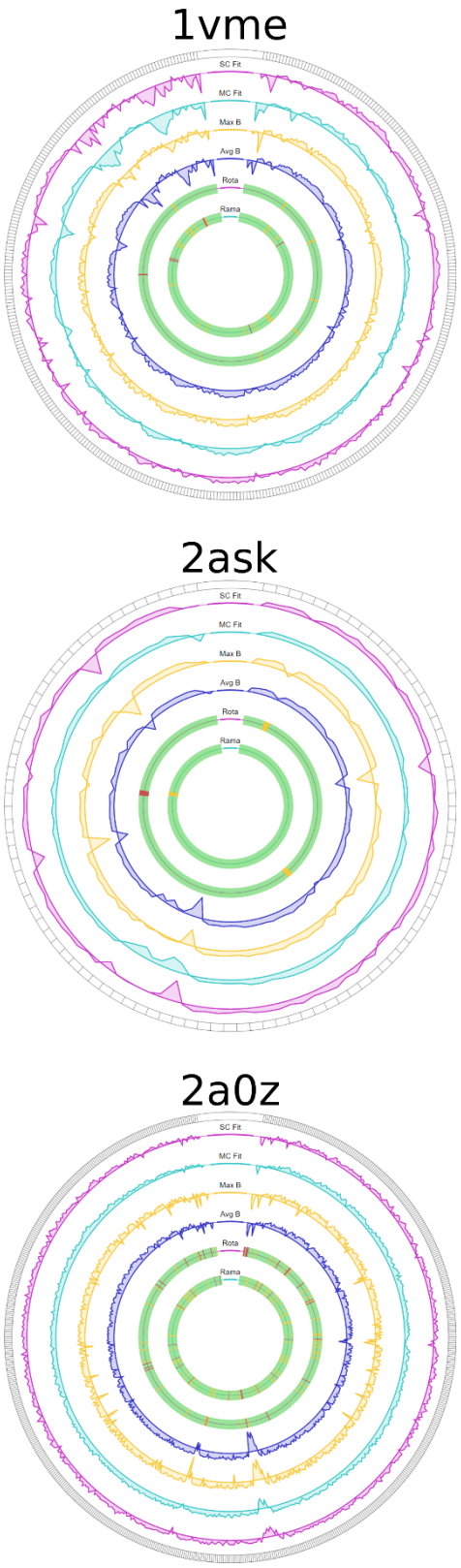

Figure 37: Chain-view chart for the latest dataset of each of the three test models after all the aforementioned changes had been made. These charts were generated using chain A of each of the three models.

2.3.4 Residue view changes

2.3.4.1 New design

Although satisfied with the adaptation of the chain view to the new discrete metrics, the accompanying radar chart now seemed mismatched; it was still showing the percentiles-based scores for every metric, including those that were now represented discretely on the chain view. Due to the non-standard scoring technique used for the rotamers, the score on the radar chart would often not correspond to the discrete classification shown for the same residue on the chain-view chart. In addition, even if the score were to reliably correspond to the classification, as would be the case for Ramachandran likelihood, showing the Ramachandran continuous score on the residue view would misleadingly imply that the difference in Ramachandran percentile score for two residues with identical Ramachandran classification is as significant as the difference in, for example, their density fit quality percentile score, which is not the case.

It was decided that a new residue-view graphic should be designed, with the ability to show the discrete metrics as classifications in a clear way. The first design for this graphic was constructed around a grid-based layout that would have a section for continuous metrics, shown on bar charts, and a section for discrete metrics, with large checkboxes containing the traffic-light colours shown on the chain-view chart. The design was trialled using the synthetic data class that had been used to prototype the other charts, and is shown in *Figure 38*.

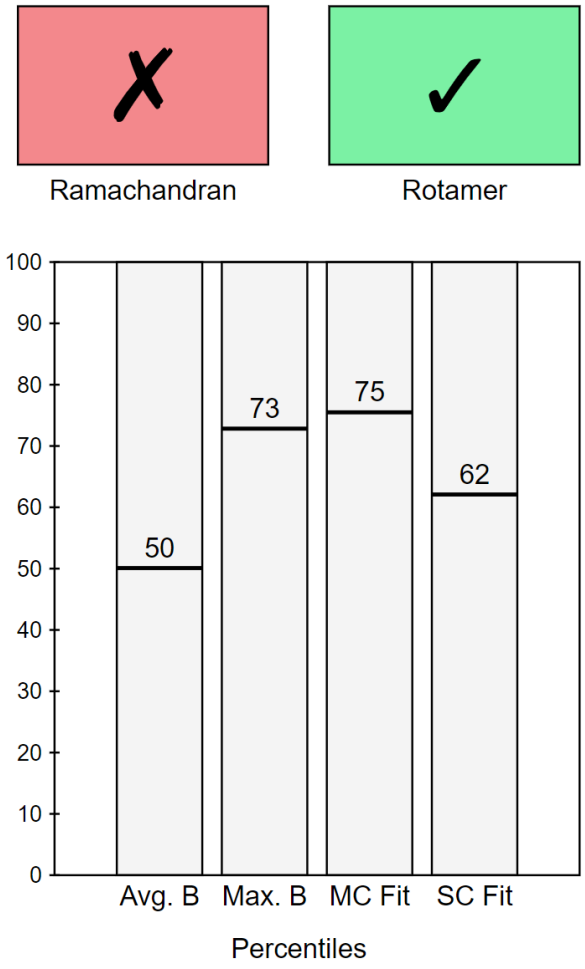

Figure 38: Initial design of the new residue-view graphic.

Though satisfied with the basis of the design, user feedback suggested some modifications that needed to be made. Many felt that the discrete classification boxes needed more detail than just the colour to adequately explain the value they indicated. In response to this, text labels were added to the boxes, explaining the classification value they represented. Feedback from non-experts revealed a common assumption that the value on each bar chart was linked in some way to the checkbox above it. While this may not confuse an expert, it certainly detracted from the intuitiveness of the graphic. Therefore, a divider line was added to clarify the delineation between the discrete and continuous sections of the graphic. Comments from many suggested that they felt the graphic seemed unfinished, or otherwise in its early stages of development. Although that was indeed the case, and such feedback may in part have been prompted by the way the graphic was introduced to the reviewers, further enquiry revealed that the reasoning behind such comments was that the bar charts appeared skeletal and empty. In response to this, traffic-light colouring was also added to the bar charts, with values less than or equal to 33

coloured red, values between 34 and 66 (inclusive) coloured amber, and values more than or equal to 67 coloured green. With these changes made, a new example was generated (*Figure 39*) and feedback was much more positive, with all reviewers stating that their reservations had been addressed.

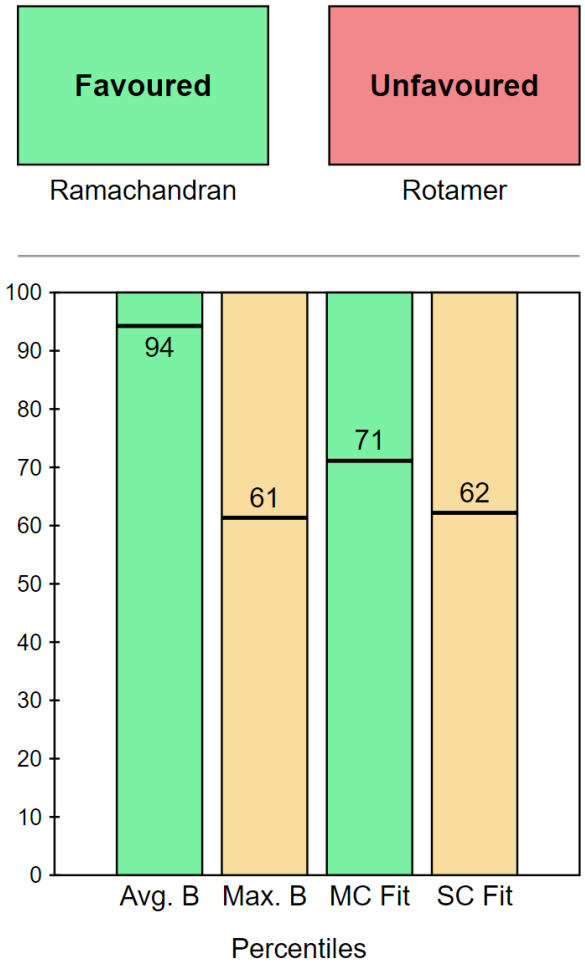

Figure 39: First revision of the grid-based residue-view graphic, with a divider bar and colour added.

It was then suggested that a more useful way to utilise the space allocated to the continuous metrics would be to collate the B-factor bars together, and the density fit bars together, leaving just two bars. This advice was heeded, because it allowed the entire graphic to be compressed horizontally. This was desirable, because it had always been intended that the chain view be the more prominent feature of the graphical panel of the report, and until this point, the chain view and residue view graphics both had a similar aspect ratio: roughly 1:1, meaning they both took up a similar amount of space. With the residue-view graphic compressed horizontally, the layout

would permit a larger chain-view by default, eliminating the need for the enlarge button in the current report template. This change was implemented, with the mean and maximum B-factor percentiles plotted in one bar, and the main- and side-chain density fit percentiles in another (*Figure 40*).

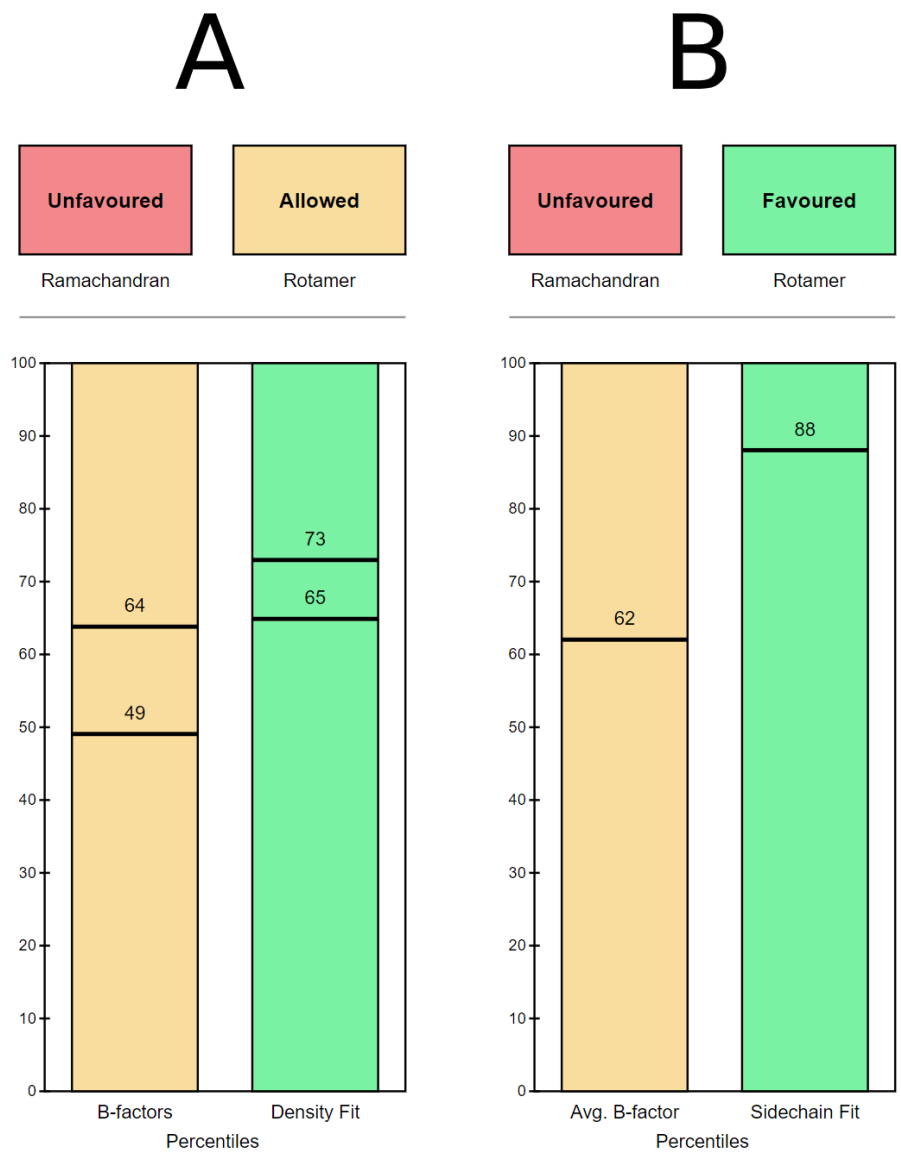

Figure 40: Second revision of the grid-based residue-view graphic, with continuous metrics collated. In (A), the left-hand bar shows both the mean and maximum B-factor percentile for the selected residue, and the right-hand bar shows both main- and side-chain density fit percentiles. In (B), only the average B-factor and side-chain fit percentiles are shown.

Of the two proposed designs, design B was more popular, and was selected as the working design.

2.3.4.2 Distribution indicators

In place of the largely superficial traffic-light colours behind the bar charts, it was decided that this space might be better utilised by introducing box plots to illustrate the distribution of metric values across the model. A few rough designs were made for this (*Figure 41*), and the favoured design was then implemented in the residue view graphic-drawing function. The distributions values displayed in the bars were based on chain-wide distributions, rather than model-wide distributions, because this felt more intuitive. To do this, the interaction JS had to be modified so that the residue-view distributions were updated each time a new chain was selected. Because the calculation time was negligible, the distribution threshold calculations were implemented in the JS code to be calculated each time a chain was selected, rather than in the Python code to be passed to the JS as variables, keeping the code tidy.

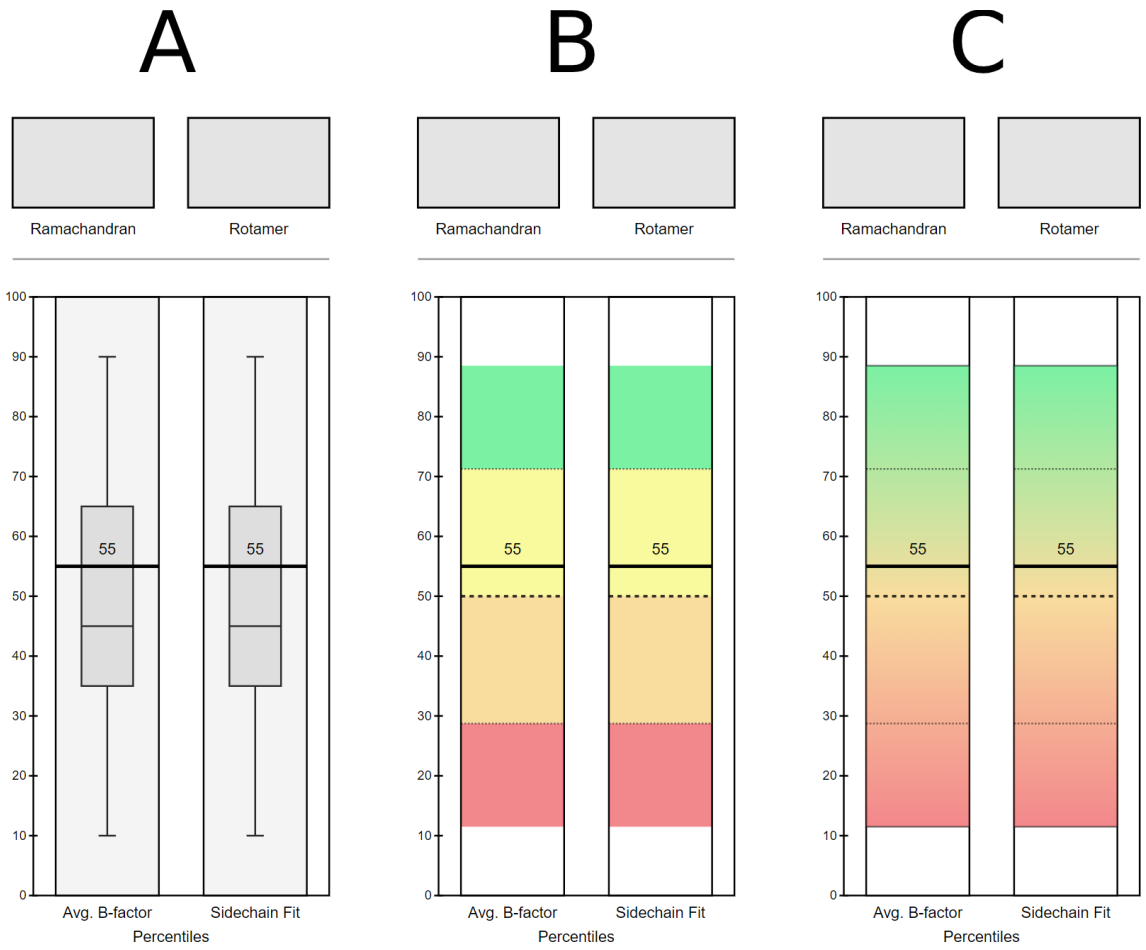

Figure 41: Different designs for bar distribution backgrounds. Chart A features classic box plots, where the boxes represent the values of the three quartiles (Q1, Q2, Q3) and the tails of each represent the minimum and maximum values. These were often quite difficult to read; the line and text label indicating the selected residue's value would often overlap with some aspect of

the box plot and, like the original designs for the graphic, they did not make any use of colour. Charts B and C use coloured areas across the whole bar to represent the chain's metric value distributions. In both of these charts, the thick dashed line represents the mean of the distribution, the thin dashed lines represent one standard deviation from the mean in each direction, and the bounds of the coloured shading represent the minimum and maximum values of the distribution. The only difference between charts B and C is that chart B colours these areas discretely, in an effort to more clearly delineate them, whereas chart C colours them on a continuous gradient, to better represent the continuous nature of the distribution being illustrated. Of these designs, design C was the most popular, so this was selected for implementation.

2.3.5 Report changes

The HTML report template was updated with a few minor updates to accommodate the updated chain- and residue-view designs. These updates included: resizing the chain view to a wider default size, and removing the accompanying resize button; adding a model iteration toggle switch above the chain view; and a number of changes to the JS code (*Figure 42*). A test report was generated for each of the test models (*Figure 43*).

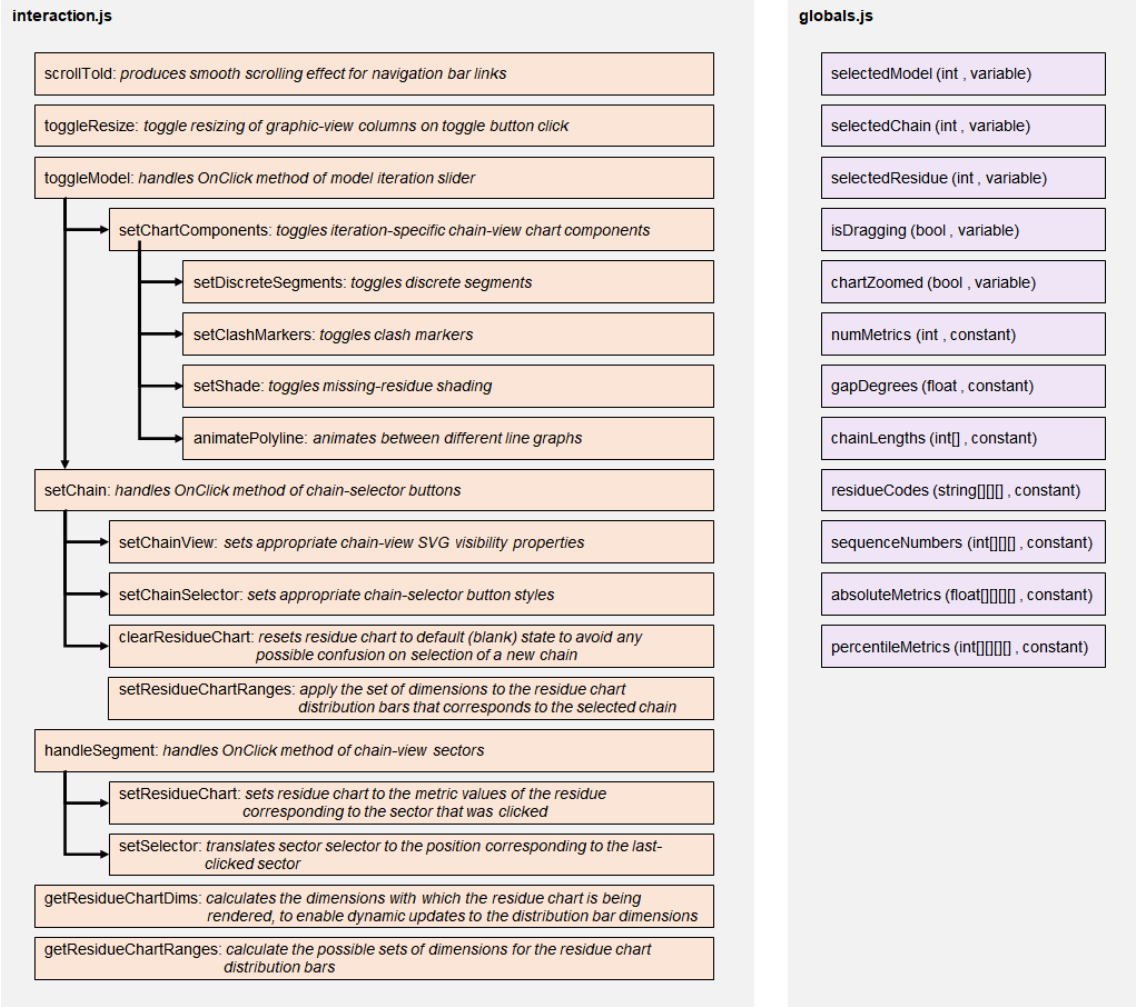

Figure 42: Finalised functions and variables of the report JS.

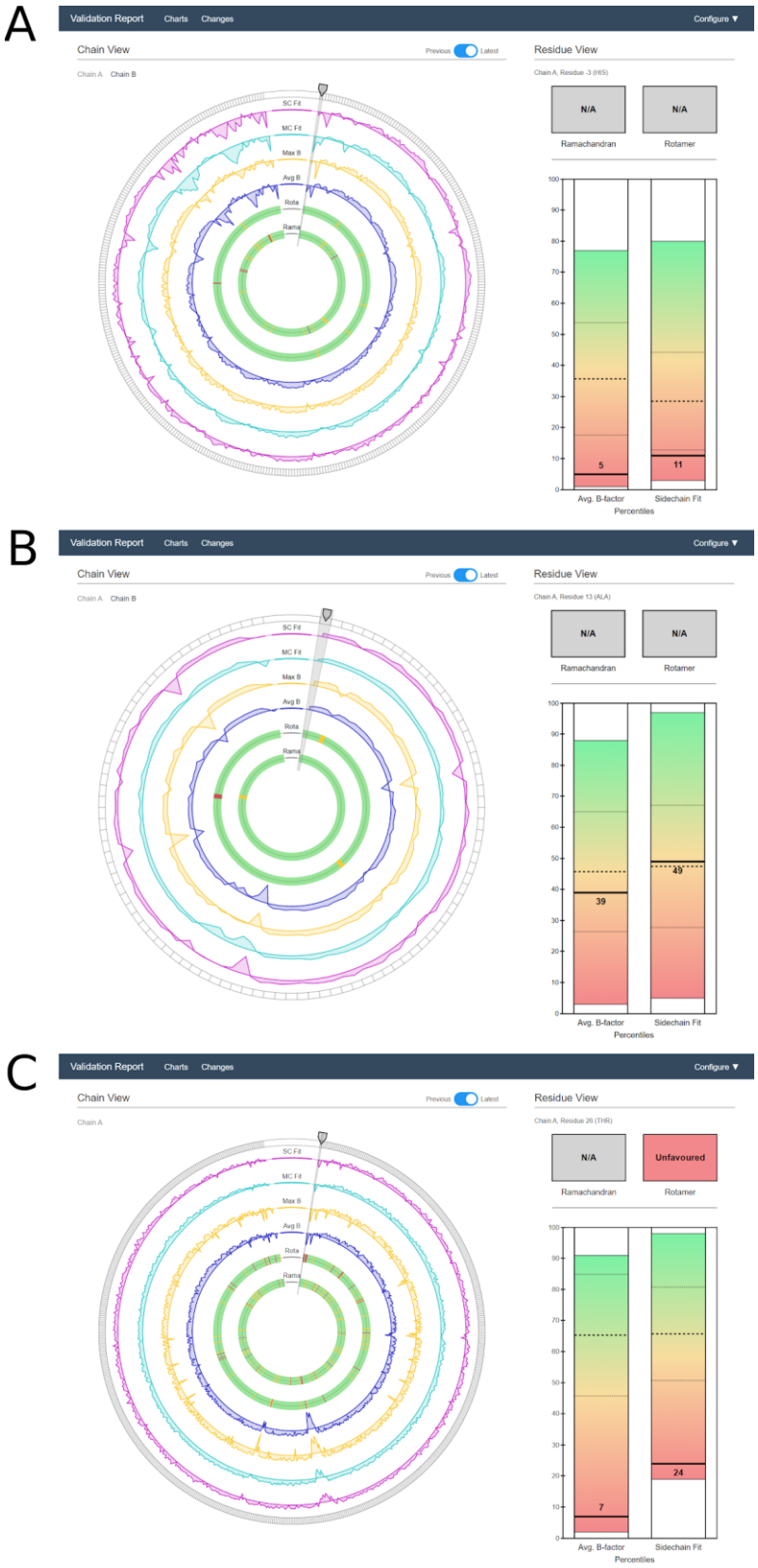

Figure 43: Regenerated test reports, featuring the updated designs of both graphics. 1vme (A), 2ask (B), 2a0z (C).

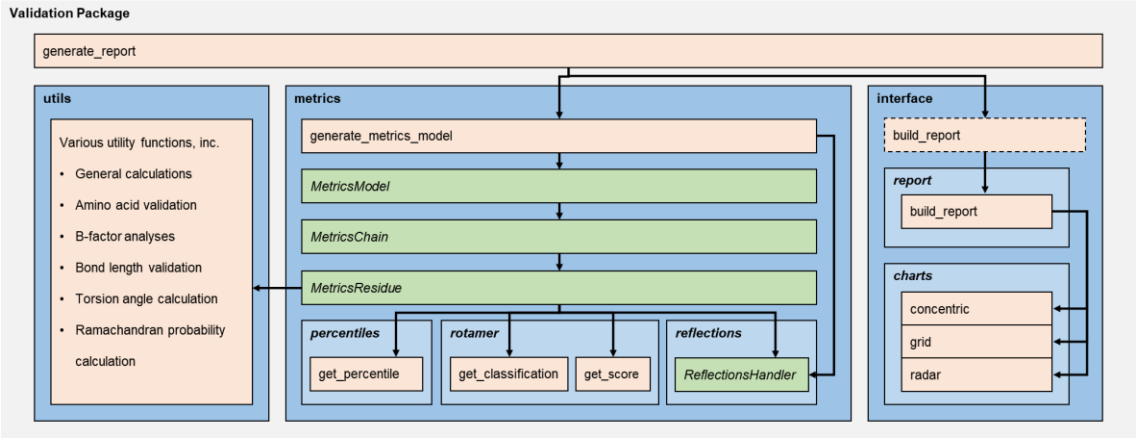

Figure 44: Overview of the structure of the entire validation package. The *generate_report* function at the top of the package first calls the *generate_metrics_model* function from the *metrics* module. This function instantiates a *ReflectionsHandler* object from the *reflections* submodule, firstly to calculate a map from the reflection data file, and then to initialise the metrics calculation cascade using the coordinates file. Each *MetricsResidue* object then performs analyses on itself using functions from the *utils* module, *rotamer* submodule, and *ReflectionsHandler* object, then runs the calculated metrics through the *percentiles* submodule. Once the metrics calculations are finished, the *generate_report* function calls the *build_report* function of the interface module, which generates the graphics and produces the finished report.

2.4 Optimisations

Because the codebase was written entirely in scripting languages (Python and JS), it was originally developed with readability in mind, as a higher priority than computational efficiency. Consequently, there was some scope for computational efficiency optimisation.

The first optimisation related to the chart-generation process. The *charts* submodule had a private function that would calculate the coordinates for a point at a given angle and distance from some centrepoint. This function would be called repeatedly during the chain-view chart generation function, and would often repeat the exact same calculations a number of times for any given chart or set of charts. To prevent this, a cache object was added such that any time the function was called, it would first check the cache to see if that calculation had already been

performed, and if it had, recall the previous result. The performance difference resulting from this change was small but significant.

Next, all matrices and matrix calculations in the library were rewritten as calls to the NumPy library (95), which both increased calculation speed and decreased memory usage.

Beyond a certain point, further computational optimisation would have to come at the cost of readability, which would not be a worthwhile compromise in the context of this project. However, there were other areas to optimise in addition to computational efficiency. For example, it was noticed that the coordinate values calculated by the chart generation function were given to a very high precision, and were being stored in the SVG files to as many as 10 decimal places. This level of precision was many orders of magnitude higher than was required, and because of the high number of elements contained within each SVG, it had a substantial effect on file size. To counteract this, the coordinate values of all SVG elements were rounded to just two decimal places before export, leading to a significant reduction in overall validation report file size.

2.5 Implementation in CCP4i2

2.5.1 Introduction

The final section of the project was to implement the now-complete validation tool within an existing validation package. CCP4i2 is the graphical, Python-based (PyQt) interface of the CCP4 suite, which "provides a framework for writing structure-solution scripts that can be built up incrementally to create increasingly automatic procedures" (2).

The CCP4i2 interface comprises sections that display links to various tasks that provide interfaces to (mostly CCP4) programs. The format of the CCP4i2 task system is straightforward. Each task essentially consists of an input frame and an output ('report') frame. The files that constitute each task are one XML properties file and three Python scripts: one to define a subclass of the *CCP4TaskWidget.CTaskWidget* class to script the task's input frame; another to define a subclass of the *CCP4PluginScript.CPluginScript* class to script the task's backend processing; and the last to define a subclass of the *CCP4ReportParser.Report* class to script the task's output report.

2.5.2 Existing validation task

The primary validation task of the interface was named *Multimetric model geometry validation*, under the *Validation and analysis* category.

The essential input fields for the task were the file paths for a model file and reflections file (*Figure 45*). Like the validation tool written in this project, the CCP4i2 validation task would use the Clipper-Python library to generate various validation metrics, which it would present to the user in the form of an HTML report (*Figure 46*). The task would also launch MolProbity analyses, via packages available in the CCP4-Python environment, and incorporate some of the results in the report. Although the task would accept reflection data as input, it would not perform any reflection-based analyses. Neither the flow of the metrics generation nor the presentation of the metrics (*Figure 47*) were as efficient as in the validation package written in this project.

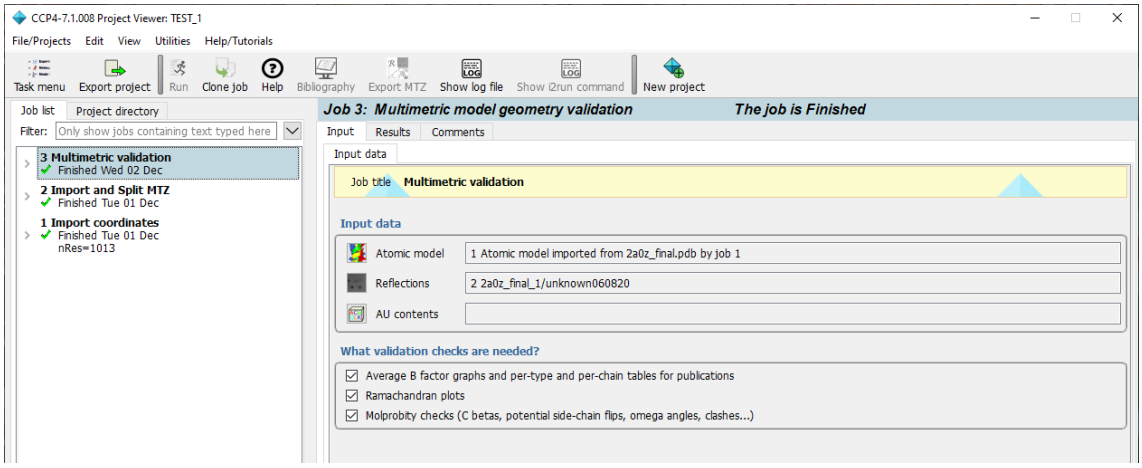

Figure 45: Input pane of the original CCP4i2 task. The input fields for the pane include file paths for a model and reflection data file, as well as for asymmetric unit descriptions. There are also three tick-box inputs to customise the contents of the output report.

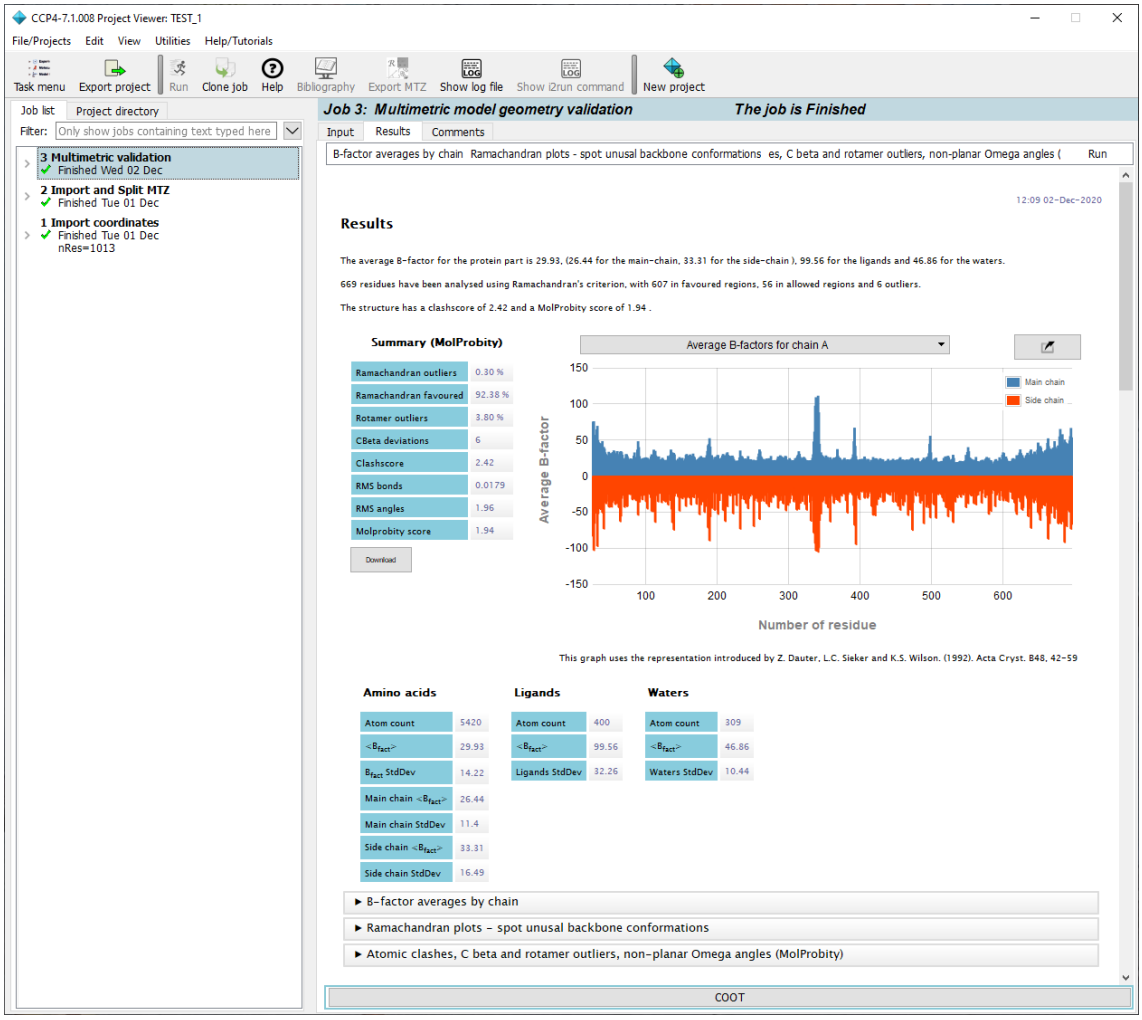

Figure 46: Example output pane of the original CCP4i2 task. In this image, only the summary section is fully expanded, which shows some summary text, a single-chain chart of residue-by-residue B-factors, some summary MolProbity analysis data, and some summary B-factor data. The three contracted sections contain more detailed B-factor analyses, Ramachandran plots, and more detailed MolProbity analyses.

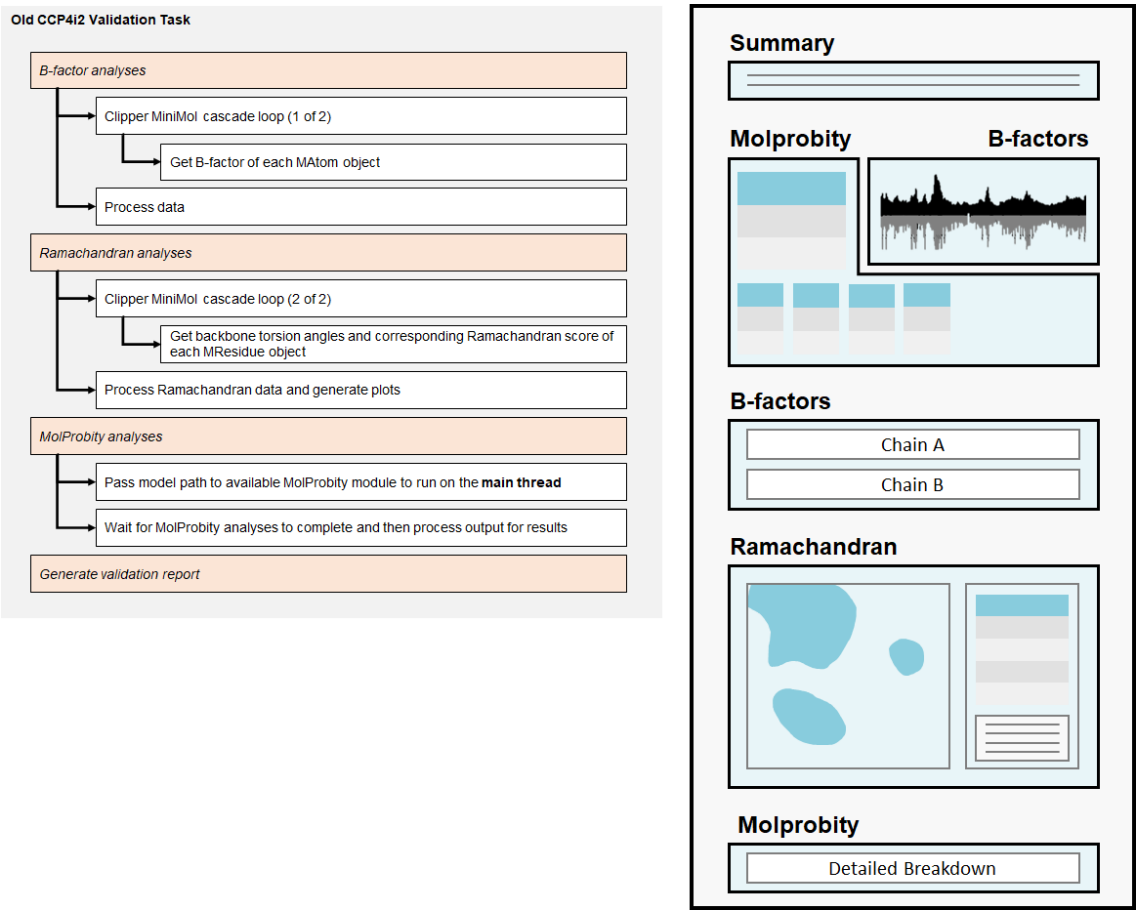

Figure 47: Summary of the flow (left) and full report format (right) of the original CCP4i2 validation task. Of note is the fact that the task would perform two MiniMol cascade loops, one for B-factor analyses and one for Ramachandran analyses. In Python, loops are unusually slow, so this two-loop system was quite inefficient.

The goal of this subsection of the project was to overhaul this task and replace it with one that would implement both the back-end (metrics calculation) and front-end (graphics generation) components of the validation tool created in this project.

2.5.3 Task redesign plan

Both the metrics generation script and output report were redesigned. The main goal for the back-end was to restructure the flow of the task to be more efficient, and for the front-end, to feature the information-dense graphical panel of the new validation package. As a consequence, this would also involve implementation of multi-model support. The redesigned flow and report format of the package are shown in *Figure 48*.

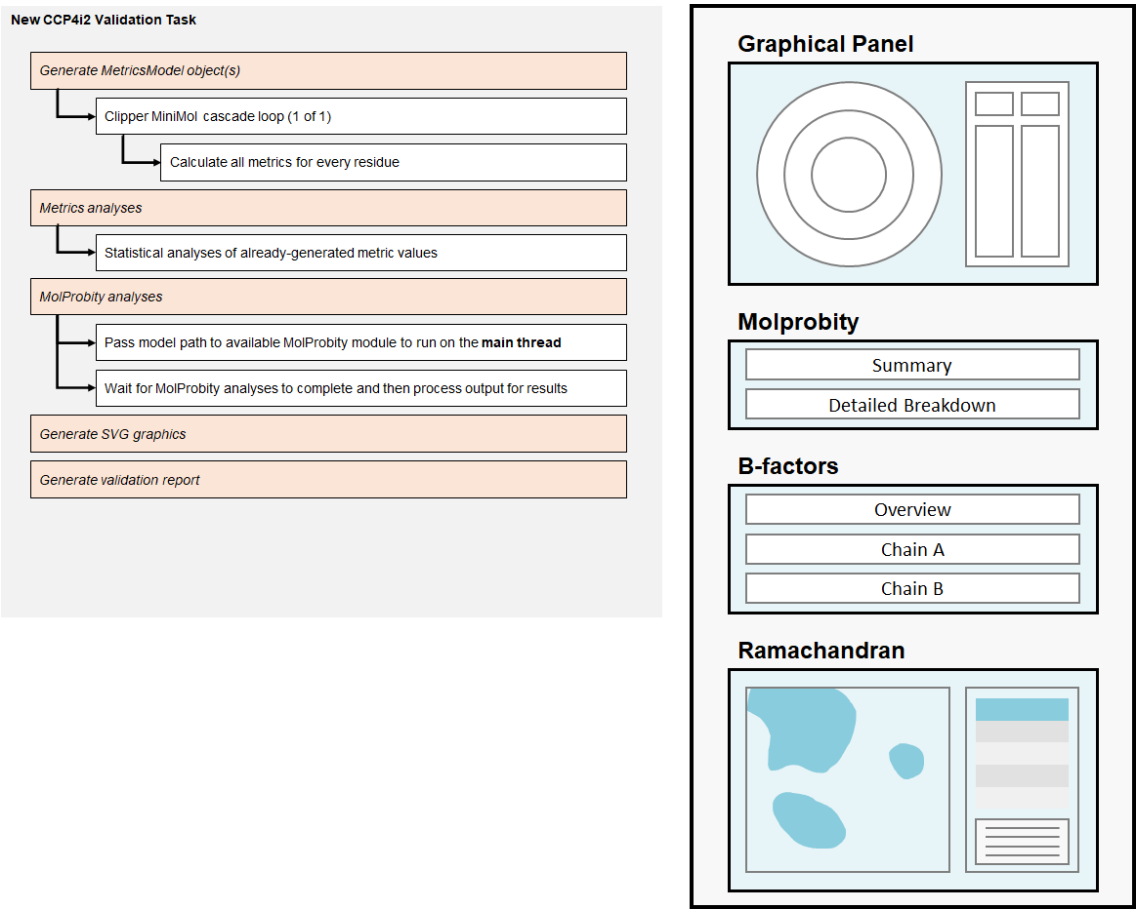

Figure 48: Summary of the redesigned flow (left) and full report format (right) of the CCP4i2 validation task. The redesigned validation task was expected to provide a few valuable speed increases. Significantly, where the original task would perform two (slow) *MiniMol* cascade loops, the redesigned task only needed to perform one, to generate the *MetricsModel* object (or objects, if two model iterations were provided). In addition to the changes to the flow, the output report was to follow a different structure, such that the graphical panel would be presented first and foremost, with the detailed MolProbity, B-factor, and Ramachandran analyses still available, each in their own contractible section.

2.5.4 Initial development

Once the task had been redesigned, its development could begin. Before assembling the redesigned task, the new validation package was added to the CCP4 Python site-packages directory, to emulate the package having been installed as part of the CCP4 suite.

The first goal was to create a test validation task that could import the new validation package and run it to create a validation report. At this stage, the objective was just to confirm that the validation report would be generated, *not* that it be rendered by the CCP4i2 interface output pane. This would verify compatibility of the package with the Python environment interfaced-with by CCP4i2.

To accomplish this, the task GUI input was modified to accept two sets of model inputs, rather than just one, to allow the user to provide model and reflection data from two iterations of model refinement (*Figure 49*). The XML properties file was updated accordingly so that the specified file paths would be assigned as globally-scoped attributes, to be accessed by the other components of the task. Finally, the main plugin script was edited to import the new validation package, and call its *generate_report* function with the file paths passed from the task GUI, to create a validation report in the default output directory of the task. Tests were successful; running the tasks would yield a blank output pane, and the production of a new subdirectory, containing the HTML validation report and accompanying scripts and stylesheets.

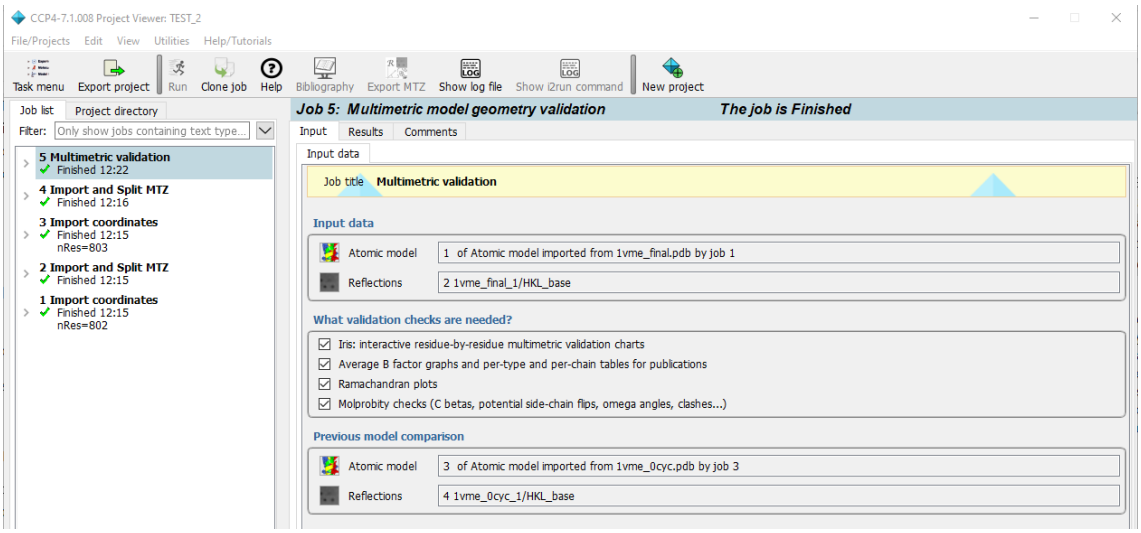

Figure 49: Input pane of the new CCP4i2 task. The input fields for the pane now include file paths for two sets of model and reflection data files. There is also a new tick-box input for users to specify whether or not they would like to produce the interactive graphical panel.

The next step was to get the HTML report integrated within the task's output pane. The first plan to accomplish this was to modify the package to introduce an option to produce a stripped-down version of the validation report, containing only the graphical panel, which could be implemented in the CCP4i2 report and other software alike, by application of an iframe element.

Modifications were made to the package to do exactly this, by including a version of the template that attached the requisite CSS and JS files, but stripped out the entirety of the rest of the template other than the div element housing the two graphics. A string argument was added to the *build_report* function of the *interface* submodule to accept the user's choice of report mode; the options being *full* for the full report (default), or *panel* for the new mode.

To implement the panel mode in the CCP4i2 task, rather than calling the module's *generate_report* function directly, the individual components of the *metrics* and *interface* submodules were called individually. This had the benefit of providing the task with direct access to the *MetricsModel* objects produced by the *metrics* module, which would be required for the more detailed analyses. The initialiser function at the start of the main plugin script would call the *generate_metrics_model* function of the *metrics* submodule, to generate a *MetricsModel* object for each of the two provided iterations, followed by the *build_report* function of the *interface* submodule, with the mode set to *panel*, and the output directory again set to a subdirectory within the task's output directory. Then, the report script was modified such that an iframe element was introduced at the top of the output HTML, with the source set to the nascent subdirectory containing the graphical panel. Unfortunately, when the task concluded and the output panel was shown, the in-app browser would consistently crash. It appeared that the CCP4i2 integrated browser did not support iframe elements within the report HTML. This process was repeated a number of times, each time with different iframe source files, all to the same effect: the browser would throw an exception and crash. It seemed that using an iframe element was not viable in the case of the CCP4i2 suite.

Therefore, a new route was taken. Rather than outputting a whole report directory, the module could be engineered to produce just the bare minimum HTML code required to contain the SVG graphics, which could then be inserted inline within the HTML of the task output pane. The difficulty here was that because the CSS and JS of the package would therefore have to be included inline with the output pane, there was potential for overlap in terms of HTML element names, CSS class names, or JS function names. Testing this method revealed that there were indeed some clashes that had to be overcome.

The first problem was that some of the generic CSS style names used by the Bootstrap template had been utilised within the default CCP4i2 report template CSS, causing overlap which led to improper styling of many elements of the report. This was simply resolved by changing CSS style names in the report template of the *interface* submodule to less common ones.

Another difficulty encountered was that none of the interaction was functioning correctly. Debugging using the JS developer console revealed that there were parse errors where the browser was not able to resolve some keywords; specifically, the *let* and *const* keywords. These were introduced to the ECMAScript standard in 2015, to provide JS with *block scope* capability (where previously the only possible scopes were *global scope* and *local scope*) with the *const* keyword providing the additional functionality that variables defined using this keyword cannot be reassigned to, behaviour that was not previously achievable. Since it is good practice to adhere to the latest edition of the ECMAScript standard when writing JS code, the JS in the *interface* module used these keywords throughout. However, although the keywords had been supported by all major web browsers for years, the integrated browser of the CCP4i2 suite did not recognise them, and so could not execute any of the included JS functions. To circumvent this, another mode was added to the report, which produced the same output as the *panel* mode, but replaced all instances of the *let* and *const* keywords with the older *var* keyword, which would define all values as global variables. This had the potential to introduce a number of problems: aside from the usual potential for namespace overlap that can arise from globally scoping all values, in JS, variables defined with *var* become attributes of the shared *window* object, which adds a whole other dimension for potential namespace overlap. Fortunately, the code was robust to this change without needing any major modifications to its overall structure.

2.5.5 Multithreading

The MolProbity analyses were by far the most time-consuming part of the validation task. In the original task, every stage of the task was performed serially, on the main thread, with the MolProbity analyses performed following the native metrics calculations. To increase the efficiency of the task, a multithreading approach was conceived. The redesigned task would use the Python multiprocessing library to start a separate process (or two processes in the case of provision of two iterations) for the MolProbity analyses, which would be initiated at the very start of the task. The MolProbity analyses would then run in the background while the main thread generated the *MetricsModel* object. Once the main thread had completed this job, it could wait for the MolProbity process(es) to conclude, and process the results.

Unfortunately, due to some idiosyncrasies of the Python 2 multiprocessing module, spawning a process from within the CCP4i2 plugin subclass was not supported under Windows. Therefore,

an operating system check had to be incorporated such that the analyses could still be run from a Windows environment, just without parallelisation (as in the original task).

2.5.6 MolProbity integration

As described in *Section 2.2.4.5*, one of the originally desired *metrics* module calculations was an atomic clash score, the implementation of which was previously postponed due to time-span concerns. With the availability of MolProbity analyses from the CCP4i2 Python environment, the results of a comprehensive MolProbity analysis were now available without any extra modifications to the *metrics* module, making the MolProbity all-atom clash results available for implementation within the graphical panel. Rather than integrate this alongside the other discrete metrics, it was decided that this metric should be presented separately, in a manner that portrays its exogeneity from the *metrics* module. To do this, an argument was added to the chain-view chart generation function to allow the addition of a new axis around the edge of the chart, whereby a marker could be placed at the edge of any residue's sector (*Figure 50*). The task's main plugin script was modified to supply this argument with an array of clash markers from the MolProbity analyses.

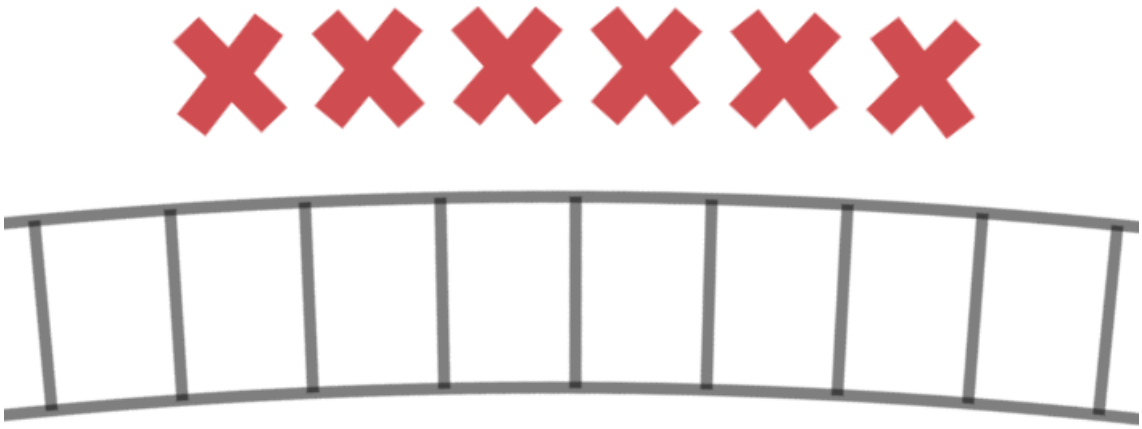

Figure 50: Appearance of the new outer markers.

In addition to the clash score, the MolProbity analyses would produce a number of other useful geometric analyses, including some of those that were also implemented within the *metrics* module; for example, the main-chain and side-chain favourability scores. In the case of side-chain conformation, the MolProbity results would be much more precise than those of the *metrics* module, thanks to the inclusion of the uncompressed reference data. Hence, it was

realised that the user may well wish to feature MolProbity analyses within the chain-view if and when they were available. Therefore, the *interface* module was modified to allow for this, and it was enabled by default within the CCP4i2 interface. When MolProbity data were being incorporated within the chart, a 'MolProbity Enabled' watermark would be shown in the centre (*Figure 51*).

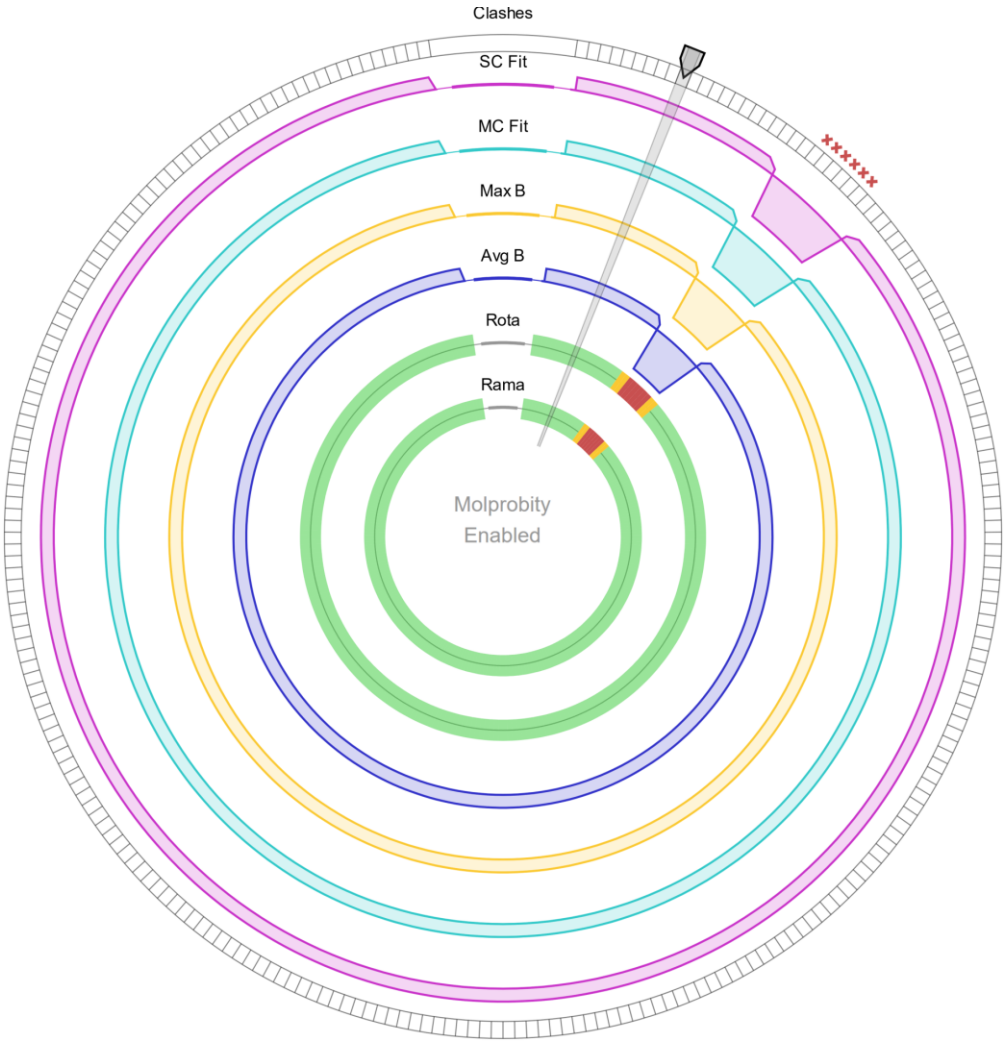

Figure 51: Watermark shown if MolProbity data has been incorporated into the chart. Generated using synthetic data.

With this, the CCP4i2 implementation was complete. See *Section 3.4* for a screenshot of an example output pane of the finished task.

3 Results and discussion

3.1 Metric accuracy

Metric values produced by the *metrics* module were tested against results from other reputable validation software packages.

3.1.1 Ramachandran likelihood score

Ramachandran scores were tested against those from MolProbity.

Ramachandran Classification		*metrics* module		
		Outlier	Allowed	Favoured
MolProbity	Outlier	68.1% (565 521)	0.9% (32 354)	0.0% (4 612)
	Allowed	31.8% (264 528)	76.2% (2 812 719)	0.8% (708 297)
	Favoured	0.1% (754)	22.9% (846 853)	99.2% (84 542 053)

Figure 52: Confusion matrix showing Ramachandran classification agreement between the metrics module and MolProbity. Percentages are of column sums. Figures in brackets are numbers of residues. Discrepancies arise as a result of the different formats of the reference data; MolProbity has access to the entire original dataset, allowing for very accurate interpolation for each case, whereas the compression used by the *metrics* module to store the reference data yields less precise classifications, especially at the interfaces between classifications (borderline cases).

3.1.2 Rotamer likelihood score

Rotamer classifications were tested against those from MolProbity.

Rotamer Classification		*metrics* module		
		Outlier	Allowed	Favoured
MolProbity	Outlier	91.5% (3 036 506)	1.9% (95 641)	0.0% (47)
	Allowed	8.5% (281 141)	87.4% (4 397 337)	0.4% (281 369)
	Favoured	0.0% (1 377)	10.7% (538 672)	99.6% (67 408 352)

Figure 53: Confusion matrix showing rotamer classification agreement between the metrics module and MolProbity. Percentages are of column sums. Figures in brackets are numbers of residues. Discrepancies are partly due to the differing interpolation methods applied by MolProbity and Clipper, but more significantly to the fact that the thresholds are arbitrary; and those selected for the *metrics* module are the ones that are used in Coot, to facilitate the transition between a CCP4i2 report and the Coot validation tools (see *Section 3.4*). These are not the same as those used by MolProbity.

3.1.3 Density fit score

The custom density fit score was tested against RSCC, calculated using EDSTATS (48).

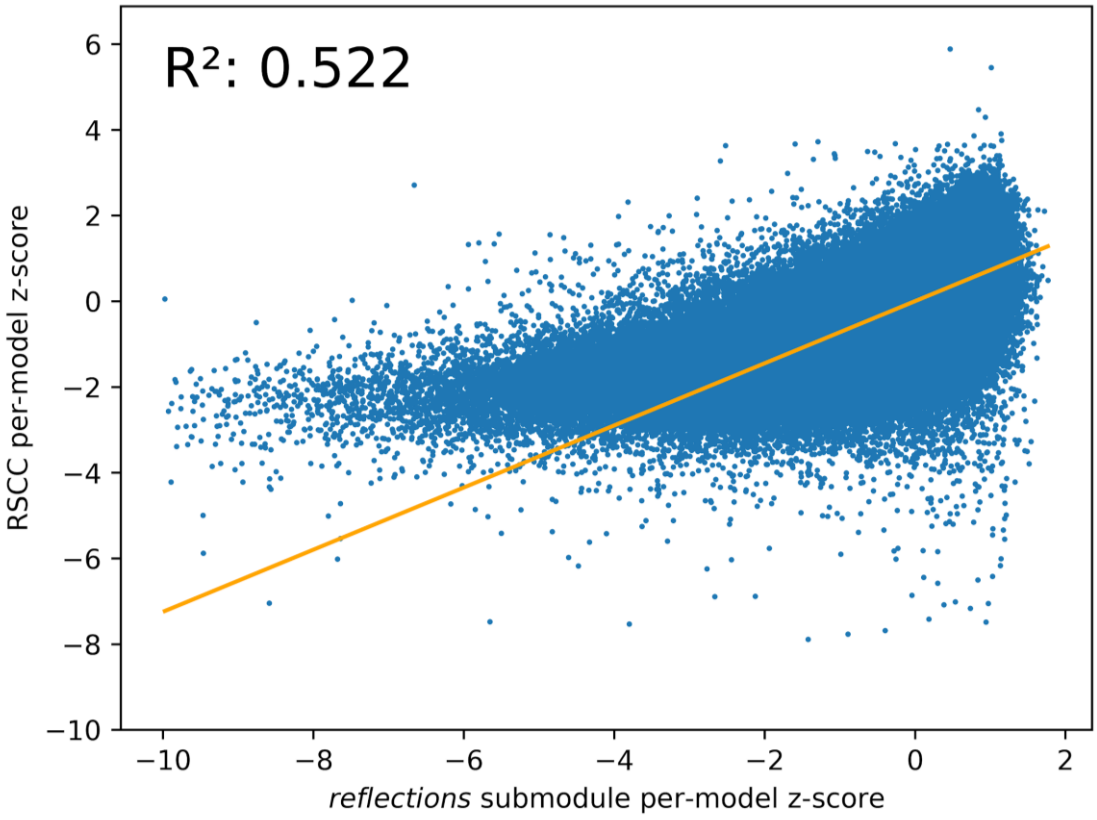

Figure 54: Line graph illustrating correlation between the reflection submodule per-residue fit scores and per-residue RSCC scores. Scores are plotted as z-scores, calculated for each residue from the population of all residue scores of the relevant structure.

Clearly, the correlation between the two metrics was quite poor. However, that was to be expected; this project's electron density fit score was not designed to correlate with RSCC, but to be an easily calculable indicator of poor fit quality. To assess the ability of the metric to identify areas of especially poor quality, a classification-based test was devised, similar to that used for Ramachandran classification. Residues with fit scores less than one standard deviation below the model mean were classified as *outliers*, and the rest were classified as *allowed*. The same classification method was applied based on RSCC, and the results were compared via a confusion matrix. This revealed a satisfactory outcome.

Density Fit Classification		*reflections* submodule	
		Outlier	Allowed
RSCC	Outlier	78.6% (24 093 747)	12.4% (8 067 799)
	Allowed	21.4% (6 546 809)	87.6% (56 909 152)

Figure 55: Confusion matrix showing density fit classification agreement between the reflections submodule and RSCC. Percentages are of column sums. Figures in brackets are numbers of residues. F_1 score is 0.767, and Matthews correlation coefficient (MCC) is 0.654. Discrepancies are simply a result of the fact that the two scoring methods are mathematically very different.

3.2 Timing

To perform timing analyses, a random selection of 20,000 models from the PDB-REDO database was made. Timing analyses were performed on multiple aspects of the software, through a high-throughput approach that best utilised the available hardware. The computer allocated for testing had an Intel i9-9900k processor, with 8 physical cores and 16 threads, running at stock frequency. To maximise the processing power available, timings would be run in parallel such that all 16 threads of the processor would be occupied concurrently. Potential avenues for bottlenecks were investigated and eliminated: the processor was liquid cooled, and maximum temperatures were around 70 degrees Celsius, so thermal throttling would not be a concern; there was a large amount of available high-frequency memory; and the storage device's read and write speeds on the order of gigabytes per second, so file input/output would not be a bottleneck.

The first timing analysed was the time taken for the entire process, i.e., the time taken to analyse the metric values of, and produce a full standalone validation report for, a given dataset (two pairs of model and reflection data). In an effort to split these time values into their respective

components, more in-depth analyses were performed. Firstly, the time taken just to calculate metrics was measured; that is, the time taken to initialise a pair of *MetricsModel* objects for a given dataset to produce metrics data. Secondly, the time taken to produce a validation report from a given pair of *MetricsModel* objects. The results of all these analyses are shown in *Figure 56*.

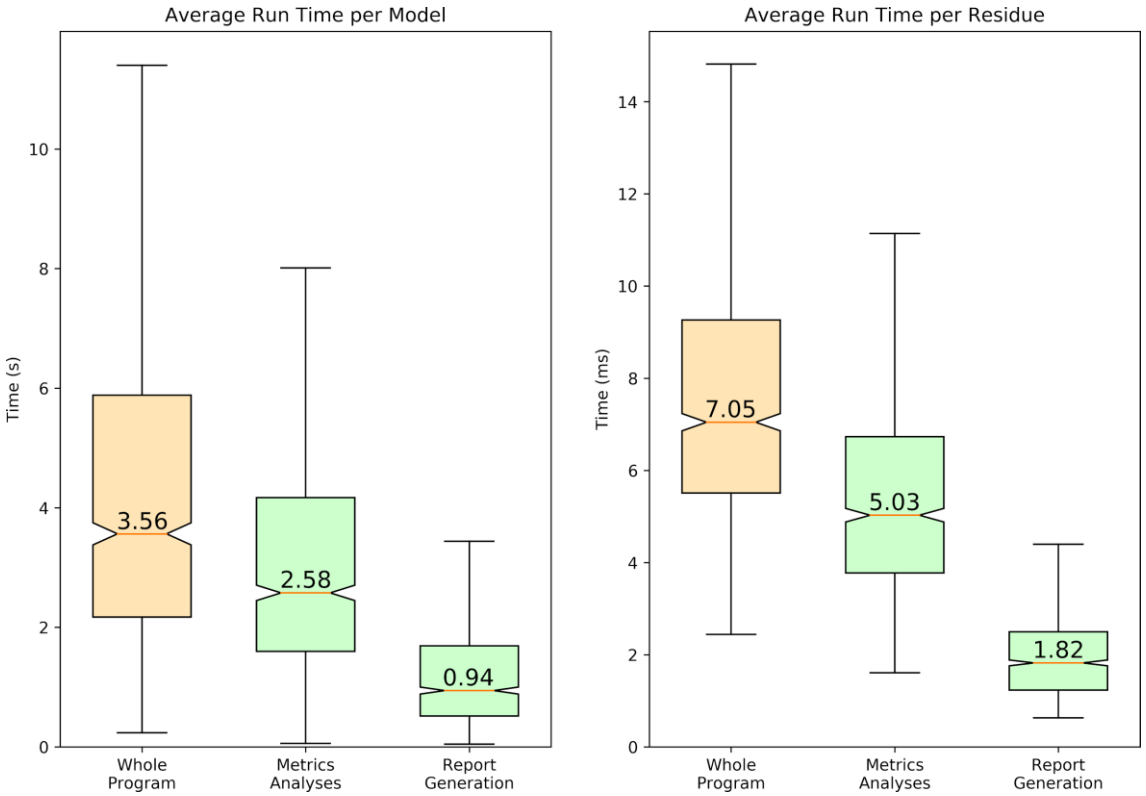

Figure 56: Box plots illustrating the distribution of results of timing analyses. On the left are the run times per model (in seconds), and on the right are the run times normalised by residue count (in milliseconds). The median value of each distribution is labelled. The breakdown reveals that metric analyses account for roughly 73% of the entire run time, on average.

Next, the implementation within the i2 interface was timed, and compared to the previously implemented task. Because of the MolProbity multithreading introduced in the new CCP4i2 validation task, each instance of *CCP4i2* could have up to three intensive processes running at once (one main thread plus two *MolProbity* threads). Therefore, to reduce the likelihood of processor thread saturation, the number of simultaneously-timed instances was reduced from 16 (as in previous tests) to 8. The results of these tests are shown in *Figure 57*. Of course, these timings would differ significantly under a Windows environment as a result of the multithreading constraints outlined in *Section 2.5.5*. If two models are provided, and MolProbity is enabled,

validation may take significantly longer than it otherwise would on a unix-based operating system such as Linux or MacOS.

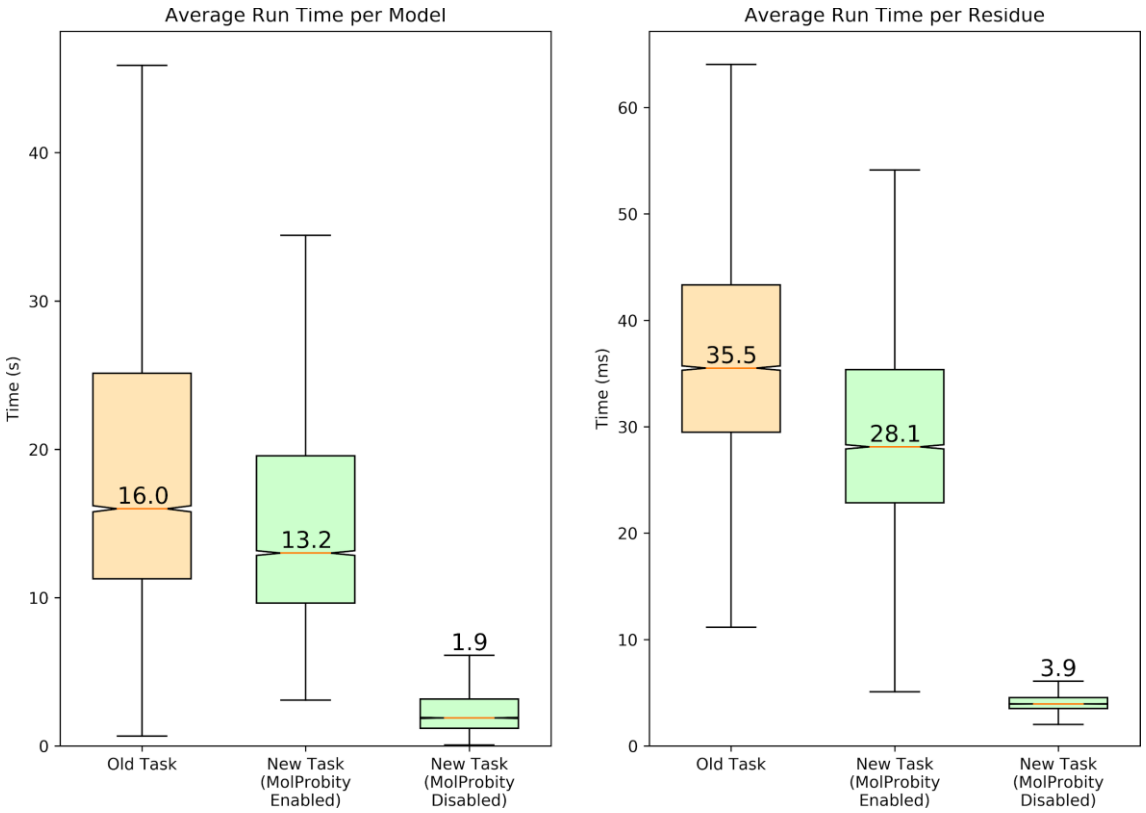

Figure 57: Box plots illustrating the distribution of average (n=5 repeats) times taken to run models (both coordinates and reflection data) through both versions of the CCP4i2 Multimetric Validation task. On the left are the run times per model (in seconds), and on the right are the run times normalised by residue count (in milliseconds). The median value of each distribution is labelled. The breakdown reveals that the new task is significantly faster than the original, both with or without MolProbity analyses enabled, despite performing twice as many analyses (two iterations).

3.3 Interface

Some test reports were generated for structures with known defects.

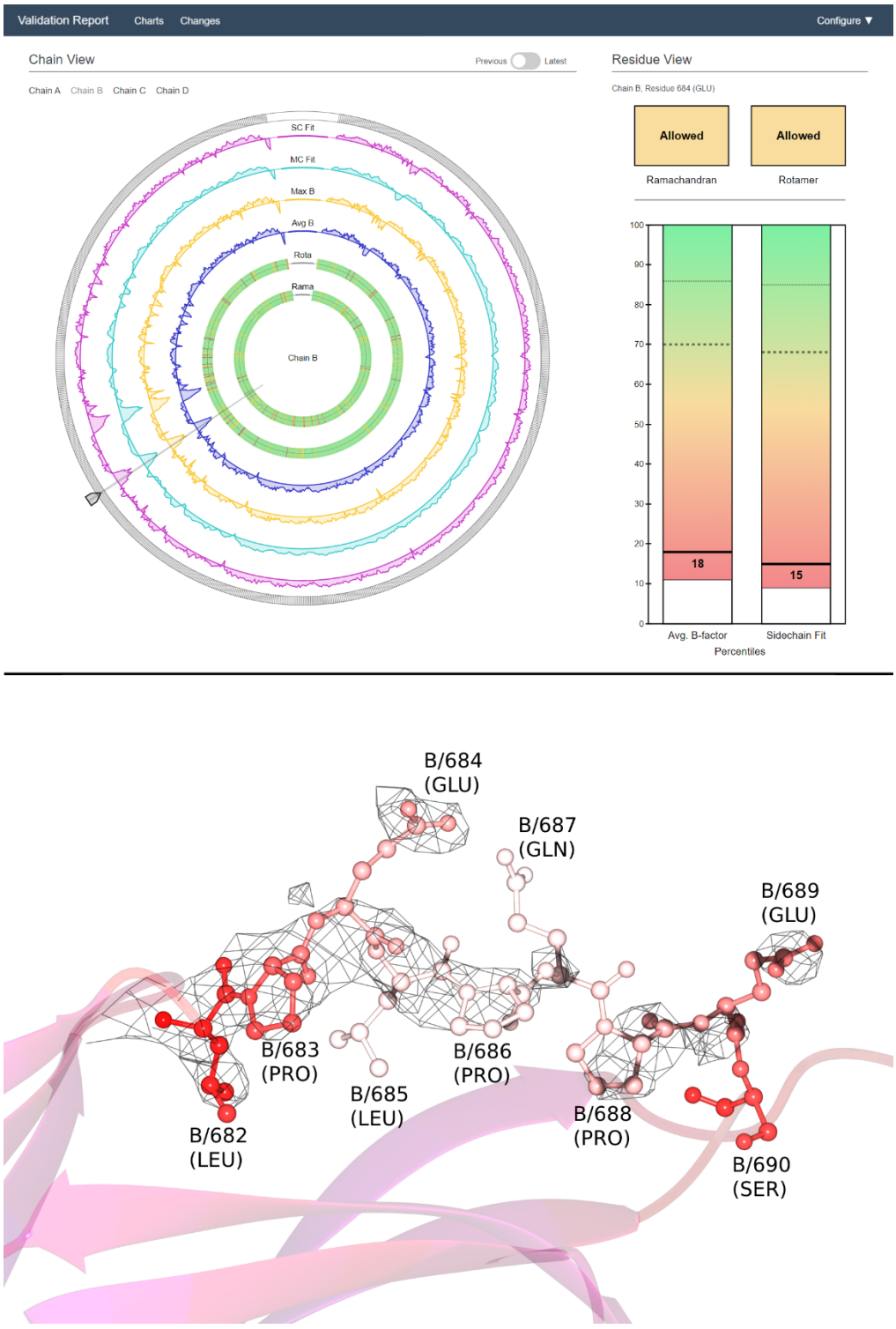

Figure 58: Example validation report for structure 3vd3 (top) and accompanying model visualisation (bottom). The screenshot shows a validation report for 3vd3, with chain B,

residue 684 selected. The iteration slider in the *previous* position, corresponding in this case to the originally-deposited model, before refinement by PDB-REDO. The selected chain comprises more than a thousand residues, demonstrating the robustness of the design to high residue-counts. For the bottom panel, the corresponding model has been coloured by B-factor (blue for low values, red and then white for high relative values) to highlight the mobility of this region. The map shows 2mFo-DFc density contoured at 1σ; the fact that the map does not cover all the residues at this level hints at the region's mobility and/or disorder. *From Rochira and Agirre, 2020* (75).

3.4 CCP4i2 implementation

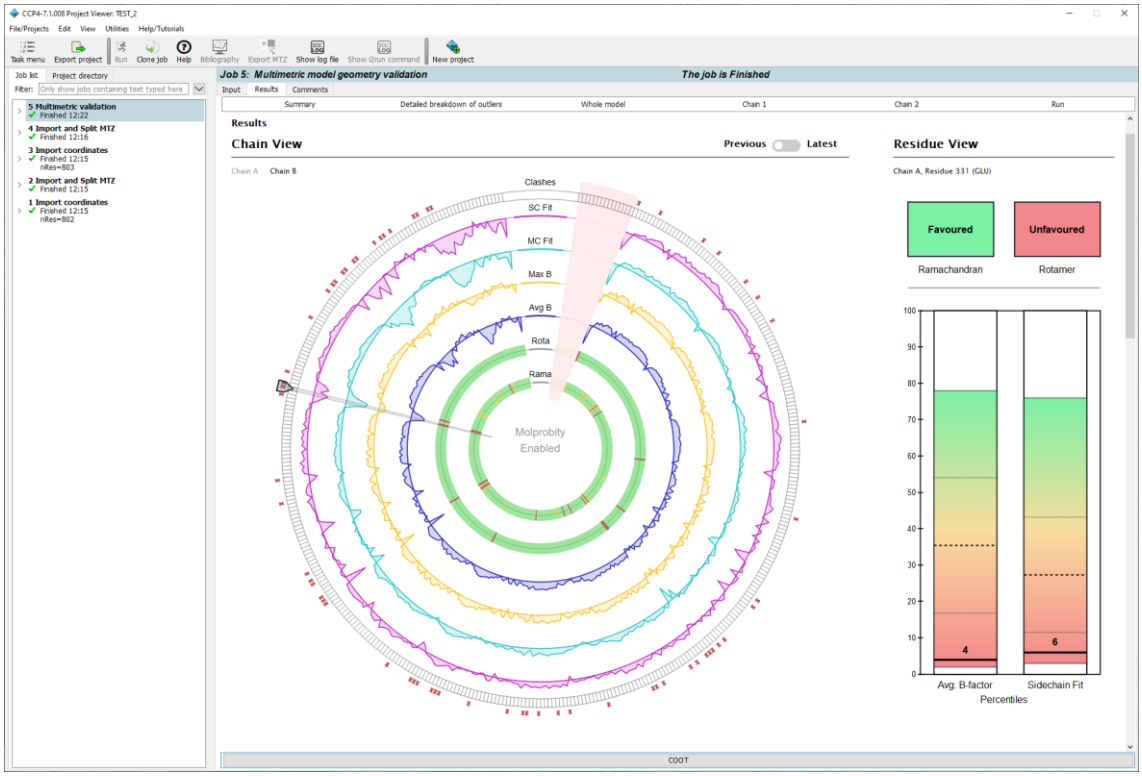

Figure 59: Example validation report from the new CCP4i2 task with MolProbity all-atom clash markers enabled. The screenshot shows the CCP4i2 interface, with the output pane of a validation task selected. This particular task validated the model 1vme. Chain A, residue 331 is selected. The iteration slider is in the *previous* position, corresponding again to the originally-deposited model, before refinement by PDB-REDO. In this screenshot, all the mentioned features of the interface are visible together, including the MolProbity integration.

At the very bottom of the task window is a button that opens the model file in Coot. This button was present in the original validation task, and was left unchanged. It was because of this integration that the Coot Ramachandran thresholds were selected for this package, so that the Ramachandran classifications shown in the graphical panel would correspond to those shown in Coot. This proved to be successful in testing.

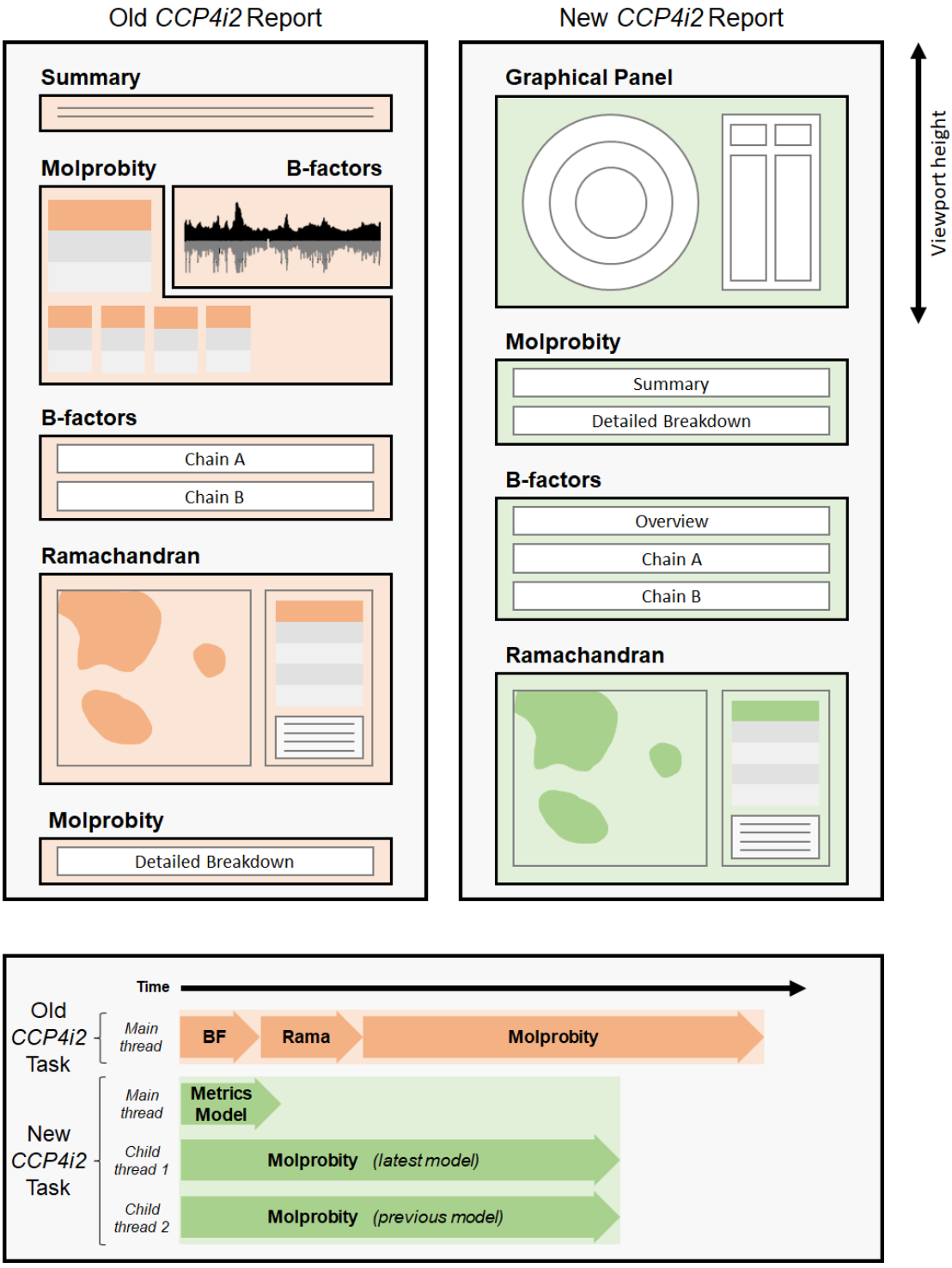

Figure 60: Design (top) and flow (bottom) of the CCP4i2 validation report before and after integration of the new validation package. The most noticeable difference in the design is that the graphical panel is now the first view presented to the user when the page is loaded, and fills the viewport to maximize the size of the chain-view display. The flow of the task has changed more significantly; where the old task performed simple B-factor and Ramachandran

analyses, then executed MolProbity analyses and compiled the results all on the same thread, the new version of the task uses Python's multiprocessing library to run concurrent MolProbity analyses on separate threads while the *metrics* module calculations are performed on the main thread. This significantly reduces the run time, despite doubling the number of analyses being performed. Because the metrics are all calculated within the same cascade, the task only has to perform one set of (slow) Python loops, as opposed to the serial repeats of loops in the original report; hence the newly-structured report has shorter run times both with and without MolProbity enabled. Timings are not to exact scale. *From Rochira and Agirre, 2020* (75).

4 Conclusions and future work

The aim of this project was to produce a tool to enable interactive all-in-one graphical validation of 3D protein model iterations, meeting the specific criteria described in *Section 1.3*. Although all the overall goals were achieved, there are a number of avenues for potential improvement and expansion.

At the most basic level, further optimisations could be made. One that stands out is multithreading: the iteration through the Clipper MiniMol cascade, and the identical operations performed repeatedly at each level are perfect candidates for multithreading. That could be implemented in several different ways, and at several different levels. For example, at a high level, each chain of a model could be iterated-through by individual worker threads in parallel with one another. Or at a lower level, a pool of worker threads could be spawned before analysis, to have individual residues divided up amongst them as the main thread iterates through the cascade. Both of these methods would be viable in theory; however, due to limitations of the Python 2 multiprocessing library, it would not have been a simple undertaking within this project. Similarly, efficiency would be improved substantially if some of the core analyses were repackaged as a C++ library to be wrapped for Python. However, considering the difficulties posed by the wrapped Clipper library, this would probably be a step in the wrong direction. Indeed, one of the overarching goals of the project was that the code be easy to read and modify to make it extensible, which would have to be sacrificed entirely if it were repackaged in a compiled language.

Aside from optimisations, there are a number of routes for expanding the functionality of the software, such as adding support for cryo-EM. As discussed in *Section 1.2.2*, cryo-EM models are becoming increasingly prevalent in the field, and experimental data obtained from cryo-EM has unique requirements that differ to those of MX data. The tools to implement cryo-EM data are already available; the Clipper *NXmap* (non-crystallographic map) class provides suitable encapsulation for finite electron density map data, and the methods to deal with it. This could be implemented with a future update to the package.

Perhaps the most pressing improvement to implement is expansion of the range of available metrics; for example, alpha-carbon torsion via the CaBLAM dataset. Since it would only be useful in the case of very low-resolution structures, such as those generated by cryo-EM, the Cα torsion reference data were not implemented in this project. However, if cryo-EM compatibility were

implemented, this metric would certainly have to be revisited. Likewise, other electron density fit scoring methods could be implemented, to expand the range of available metrics. Although the fit score invented for use in this project was demonstrably suitable, it is non-standard, and its values are not comparable to those from other validation software packages. To counteract this, traditional fit scores such as RSCC and RSR could be implemented, either calculated directly by the *metrics* module or by hooking some external program, such as EDSTATS (48). In addition, MolProbity analyses could be implemented directly within the *metrics* module, via the CCTBX Python package. In this way, the user would be able to choose either built-in or MolProbity analyses when using the package in any context, including as a standalone solution, rather than having to choose an implementation of the package that makes MolProbity analyses available, such as CCP4i2.

An intrinsic problem with representing a three-dimensional structure with a two-dimensional graphic is that some in-space interactions become very difficult to represent meaningfully and intuitively. There are often important interactions involving residues that are close in tertiary or quaternary structure, but far apart in primary structure, and thus are difficult to illustrate on a flat, sequential chart. Such interactions include residue-residue interactions, such as hydrogen bonds and disulfide bridges, and also residue-molecule interactions, where many residues may interact with the same ligand or cofactor. Further developments to the graphic would lead to the development of a satisfactory way of displaying such information, perhaps with the application of a customisable layered system. In a similar vein, representation of post-translational modifications, such as glycosylation, could be added. For example, via implementation of the two-dimensional glycan notation generated by Privateer (99).

Although the integration within the CCP4i2 suite provided an apt demonstration of the pluggable nature of the code, this has ignored the majority of validation pipelines. In time, the software could be integrated in a number of other validation programs. Likely candidates include CCP4mg (47), Coot (46), and ChimeraX (100). Additionally, modifications could be made to the standalone report to allow users to tie the standalone validation reports into their existing validation pipelines; for instance, by adding dynamically-updated hyperlinks to the report which open the model already centred on a selected residue in model-viewing software such as Coot. This could also be used to prompt users with suggestions for a number of automatically-detected actions, such as peptide flips.

Finally, the most important and longest-term goal is the inception of new validation metrics that are entirely separate from the refinement process, such that they cannot be targeted by automated refinement procedures. This would open the door to truly independent model evaluation, and is an avenue that should be explored.

5 Bibliography

1. Davis IW, Leaver-Fay A, Chen VB, Block JN, Kapral GJ, Wang X, et al. MolProbity: all-atom contacts and structure validation for proteins and nucleic acids. Nucleic Acids Res. 2007 Jul;35(Web Server issue):W375-83.

2. Potterton L, Agirre J, Ballard C, Cowtan K, Dodson E, Evans PR, et al. CCP4i2: the new graphical user interface to the CCP4 program suite. Acta Crystallogr D Struct Biol. 2018 Feb 1;74(Pt 2):68–84.

3. Tiessen A, Pérez-Rodríguez P, Delaye-Arredondo LJ. Mathematical modeling and comparison of protein size distribution in different plant, animal, fungal and microbial species reveals a negative correlation between protein size and protein number, thus providing insight into the evolution of proteomes. BMC Res Notes. 2012 Feb 1;5:85.

4. Ambrogelly A, Palioura S, Söll D. Natural expansion of the genetic code. Nat Chem Biol. 2007 Jan;3(1):29–35.

5. Lodish H, Berk A, Lawrence Zipursky S, Matsudaira P, Baltimore D, Darnell J. Hierarchical Structure of Proteins. W. H. Freeman; 2000.

6. Lomize AL, Pogozheva ID, Lomize MA, Mosberg HI. The role of hydrophobic interactions in positioning of peripheral proteins in membranes. BMC Struct Biol. 2007 Jun 29;7:44.

7. Mehler EL, Fuxreiter M, Simon I, Garcia-Moreno EB. The role of hydrophobic microenvironments in modulating pKa shifts in proteins. Proteins. 2002 Aug 1;48(2):283–92.

8. Fratzl P. Collagen: Structure and Mechanics, an Introduction. In: Fratzl P, editor. Collagen: Structure and Mechanics. Boston, MA: Springer US; 2008. p. 1–13.

9. Berman HM, Westbrook J, Feng Z, Gilliland G, Bhat TN, Weissig H, et al. The Protein Data Bank. Nucleic Acids Res. 2000 Jan 1;28(1):235–42.

10. Röthlisberger D, Khersonsky O, Wollacott AM, Jiang L, DeChancie J, Betker J, et al. Kemp elimination catalysts by computational enzyme design. Nature. 2008 May 8;453(7192):190–5.

11. World's first artificial enzymes created using synthetic biology [Internet]. 2014 [cited 2020 Dec 4]. Available from: https://www.cam.ac.uk/research/news/worlds-first-artificial-enzymes-created-using-synthetic-biology

12. Terkeltaub R, Sundy JS, Schumacher HR, Murphy F, Bookbinder S, Biedermann S, et al. The interleukin 1 inhibitor rilonacept in treatment of chronic gouty arthritis: results of a placebo-controlled, monosequence crossover, non-randomised, single-blind pilot study. Ann Rheum Dis. 2009 Oct;68(10):1613–7.

13. Erbas-Cakmak S, Leigh DA, McTernan CT, Nussbaumer AL. Artificial Molecular Machines. Chem Rev. 2015 Sep 23;115(18):10081–206.

14. On a New Kind of Rays. Nature. 1896 Jan 1;53(1369):274–6.

15. Friedrich W, Knipping P, Laue M. Interferenzerscheinungen bei Röntgenstrahlen. Ann Phys. 1913;346(10):971–88.

16. Bernal JD, Crowfoot D. X-Ray Photographs of Crystalline Pepsin. Nature. 1934 May 1;133(3369):794–5.

17. Kendrew JC, Bodo G, Dintzis HM, Parrish RG, Wyckoff H, Phillips DC. A three-dimensional model of the myoglobin molecule obtained by x-ray analysis. Nature. 1958 Mar 8;181(4610):662–6.

18. Cochran W, Crick FH, Vand V. The structure of synthetic polypeptides. I. The transform of atoms on a helix. Acta Crystallogr. 1952 Sep 10;5(5):581–6.

19. Watson JD, Crick FH. Molecular structure of nucleic acids; a structure for deoxyribose nucleic acid. Nature. 1953 Apr 25;171(4356):737–8.

20. Source DL. Diamond Light Source [Internet]. [cited 2020 Nov 5]. Available from: https://www.diamond.ac.uk/

21. Knapek E, Dubochet J. Beam damage to organic material is considerably reduced in cryo-electron microscopy. J Mol Biol. 1980 Aug 5;141(2):147–61.

22. Newmark P. Cryo-transmission microscopy: Fading hopes. Nature. 1982 Sep 1;299(5882):386–7.

23. Adrian M, Dubochet J, Lepault J, McDowall AW. Cryo-electron microscopy of viruses. Nature. 1984;308(5954):32–6.

24. Murata K, Wolf M. Cryo-electron microscopy for structural analysis of dynamic biological macromolecules. Biochim Biophys Acta Gen Subj. 2018 Feb;1862(2):324–34.

25. Frank J. Three-Dimensional Electron Microscopy of Macromolecular Assemblies: Visualization of Biological Molecules in Their Native State. Oxford University Press; 2006. 432 p.

26. van Heel M, Gowen B, Matadeen R, Orlova EV, Finn R, Pape T, et al. Single-particle electron cryo-microscopy: towards atomic resolution. Q Rev Biophys. 2000 Nov;33(4):307–69.

27. Kühlbrandt W. Biochemistry. The resolution revolution. Science. 2014 Mar 28;343(6178):1443–4.

28. Kühlbrandt W. Cryo-EM enters a new era. Elife. 2014 Aug 13;3:e03678.

29. Cross TA, Opella SJ. Solid-state NMR structural studies of peptides and proteins in membranes. Curr Opin Struct Biol. 1994 Jan 1;4(4):574–81.

30. Marassi FM, Opella SJ. A solid-state NMR index of helical membrane protein structure and topology. J Magn Reson. 2000 May;144(1):150–5.

31. Castellani F, van Rossum B, Diehl A, Schubert M, Rehbein K, Oschkinat H. Structure of a protein determined by solid-state magic-angle-spinning NMR spectroscopy. Nature. 2002 Nov 7;420(6911):98–102.

32. Zuiderweg ERP. Mapping protein-protein interactions in solution by NMR spectroscopy. Biochemistry. 2002 Jan 8;41(1):1–7.

33. Takeuchi K, Wagner G. NMR studies of protein interactions. Curr Opin Struct Biol. 2006 Feb;16(1):109–17.

34. Bonvin AMJJ, Boelens R, Kaptein R. NMR analysis of protein interactions. Curr Opin Chem Biol. 2005 Oct;9(5):501–8.

35. Vaynberg J, Qin J. Weak protein-protein interactions as probed by NMR spectroscopy. Trends Biotechnol. 2006 Jan;24(1):22–7.

36. Guzzo AV. The influence of amino-acid sequence on protein structure. Biophys J. 1965 Nov;5(6):809–22.

37. Prothero JW. Correlation between the distribution of amino acids and alpha helices. Biophys J. 1966 May;6(3):367–70.

38. Senior AW, Evans R, Jumper J, Kirkpatrick J, Sifre L, Green T, et al. Improved protein structure prediction using potentials from deep learning. Nature. 2020 Jan;577(7792):706–10.

39. Callaway E. "It will change everything": DeepMind's AI makes gigantic leap in solving protein structures. Nature [Internet]. 2020 Nov 30; Available from: http://dx.doi.org/10.1038/d41586-020-03348-4

40. Bank RPD. RCSB PDB [Internet]. [cited 2020 Nov 4]. Available from: https://www.rcsb.org/stats/summary

41. Bank RPD. RCSB PDB: Homepage. Rcsb Pdb [Internet]. 2019; Available from: https://www.rcsb.org/stats/summary

42. Collaborative Computational Project, Number. The CCP4 suite: programs for protein crystallography. Acta Crystallogr D Biol Crystallogr. 1994 Sep 1;50(5):760–3.

43. Adams PD, Afonine PV, Bunkóczi G, Chen VB, Davis IW, Echols N, et al. PHENIX: a comprehensive Python-based system for macromolecular structure solution. Acta Crystallogr D Biol Crystallogr. 2010 Feb;66(Pt 2):213–21.

44. Cowtan K. Automated model building with Buccaneer [Internet]. CCP4 Crystallography School and Workshop; 2017 Oct; The Guangzhou Institute of Biomedicine and Health, Chinese Academy of Sciences, Guangzhou, China. Available from: https://www.ccp4.ac.uk/schools/China-2017/lectures/buccaneer.pdf

45. Coordinate Section [Internet]. [cited 2020 Nov 22]. Available from: https://www.wwpdb.org/documentation/file-format-content/format33/sect9.html

46. Emsley P, Cowtan K. Coot: model-building tools for molecular graphics. Acta Crystallogr D Biol Crystallogr. 2004 Dec;60(Pt 12 Pt 1):2126–32.

47. McNicholas S, Potterton E, Wilson KS, Noble MEM. Presenting your structures: the CCP4mg molecular-graphics software. Acta Crystallogr D Biol Crystallogr. 2011 Apr;67(Pt 4):386–94.

48. Tickle IJ. Statistical quality indicators for electron-density maps. Acta Crystallogr D Biol Crystallogr. 2012 Apr;68(Pt 4):454–67.

49. Read RJ, Adams PD, Arendall WB 3rd, Brunger AT, Emsley P, Joosten RP, et al. A new generation of crystallographic validation tools for the protein data bank. Structure. 2011 Oct 12;19(10):1395–412.

50. Kleywegt GJ, Jones TA. Where freedom is given, liberties are taken. Structure. 1995 Jun 15;3(6):535–40.

51. Weichenberger CX, Pozharski E, Rupp B. Visualizing ligand molecules in Twilight electron density. Acta Crystallogr Sect F Struct Biol Cryst Commun. 2013 Feb 1;69(Pt 2):195–200.

52. Crispin M, Stuart DI, Jones EY. Building meaningful models of glycoproteins. Nat Struct Mol Biol. 2007 May;14(5):354; discussion 354-5.

53. Agirre J, Davies G, Wilson K, Cowtan K. Carbohydrate anomalies in the PDB. Nat Chem Biol. 2015 May;11(5):303.

54. Hooft RW, Vriend G, Sander C, Abola EE. Errors in protein structures. Nature. 1996 May 23;381(6580):272.

55. Joosten RP, Womack T, Vriend G, Bricogne G. Re-refinement from deposited X-ray data can deliver improved models for most PDB entries. Acta Crystallogr D Biol Crystallogr. 2009 Feb;65(Pt 2):176–85.

56. Diamond R. A real-space refinement procedure for proteins. Acta Crystallogr A. 1971 Sep 1;27(5):436–52.

57. Sheldrick GM. SHELX-76, Program for Crystal Structure Determination, University of Cambridge, Cambridge, UK, 1976. Search PubMed. 1986;

58. Sheldrick GM. A short history of SHELX. Acta Crystallogr A. 2008 Jan;64(Pt 1):112–22.

59. Sussman JL, Holbrook SR, Church GM, Kim S-H. A structure-factor least-squares refinement procedure for macromolecular structures using constrained and restrained parameters. Acta Crystallogr A. 1977 Sep 1;33(5):800–4.

60. Dodson EJ, Isaacs NW, Rollett JS. A method for fitting satisfactory models to sets of atomic positions in protein structure refinements. Acta Crystallogr A. 1976 Mar 1;32(2):311–5.

61. Laskowski RA, MacArthur MW, Moss DS, Thornton JM. PROCHECK: a program to check the stereochemical quality of protein structures. J Appl Crystallogr. 1993 Apr 1;26(2):283–91.

62. Vriend G. WHAT IF: a molecular modeling and drug design program. J Mol Graph. 1990 Mar;8(1):52–6, 29.

63. Wilson KS, Butterworth S, Dauter Z, Lamzin VS, Walsh M, Wodak S, et al. Who checks the checkers? Four validation tools applied to eight atomic resolution structures. J Mol Biol. 1998;276(2):417.

64. Bowie JU, Lüthy R, Eisenberg D. A method to identify protein sequences that fold into a

known three-dimensional structure. Science. 1991 Jul 12;253(5016):164–70.

65. Lüthy R, Bowie JU, Eisenberg D. Assessment of protein models with three-dimensional profiles. Nature. 1992 Mar 5;356(6364):83–5.

66. Sippl MJ. Recognition of errors in three-dimensional structures of proteins. Proteins. 1993 Dec;17(4):355–62.

67. Colovos C, Yeates TO. Verification of protein structures: patterns of nonbonded atomic interactions. Protein Sci. 1993 Sep;2(9):1511–9.

68. Jones TA. Interactive electron-density map interpretation: from INTER to O. Acta Crystallogr D Biol Crystallogr. 2004 Dec;60(Pt 12 Pt 1):2115–25.

69. Cowtan K. The Clipper C++ libraries for X-ray crystallography. IUCr Computing Commission Newsletter. 2003;2(4):9.

70. Urzhumtseva L, Afonine PV, Adams PD, Urzhumtsev A. Crystallographic model quality at a glance. Acta Crystallogr D Biol Crystallogr. 2009 Mar;65(Pt 3):297–300.

71. Young JY, Westbrook JD, Feng Z, Sala R, Peisach E, Oldfield TJ, et al. OneDep: Unified wwPDB System for Deposition, Biocuration, and Validation of Macromolecular Structures in the PDB Archive. Structure. 2017 Mar 7;25(3):536–45.

72. Kleywegt GJ, Jones TA. OOPS-a-daisy. ESF/CCP4 Newsletter. 1994;30:20–4.

73. Lamzin VS, Perrakis A, Wilson KS. ARP/wARP – automated model building and refinement. In Chester, England: International Union of Crystallography; p. 525–8.

74. Terwilliger TC, Grosse-Kunstleve RW, Afonine PV, Moriarty NW, Zwart PH, Hung LW, et al. Iterative model building, structure refinement and density modification with the PHENIX AutoBuild wizard. Acta Crystallogr D Biol Crystallogr. 2008 Jan;64(Pt 1):61–9.

75. Rochira W, Agirre J. Iris: Interactive all-in-one graphical validation of 3D protein model iterations. Protein Sci. 2020 Oct 19;67:386.

76. Hunter JD. Matplotlib: A 2D Graphics Environment. Computing in Science Engineering. 2007 May;9(3):90–5.

77. Chart.js [Internet]. [cited 2020 Dec 10]. Available from: https://www.chartjs.org/

78. Moitzi M. svgwrite [Internet]. Github; [cited 2020 Sep 6]. Available from: https://github.com/mozman/svgwrite

79. Otto M, Thornton J, Rebert C, Thilo J, XhmikosR FH, Others. Bootstrap. Retrieved May. 2011;15:2019.

80. Otto M, Thornton J, Bootstrap contributors. Examples [Internet]. [cited 2020 Nov 23]. Available from: https://getbootstrap.com/docs/4.0/examples/

81. Joosten RP, Salzemann J, Bloch V, Stockinger H, Berglund A-C, Blanchet C, et al. PDB_REDO: automated re-refinement of X-ray structure models in the PDB. J Appl Crystallogr. 2009 Jun 1;42(Pt 3):376–84.

82. Joosten RP, Joosten K, Cohen SX, Vriend G, Perrakis A. Automatic rebuilding and optimization of crystallographic structures in the Protein Data Bank. Bioinformatics. 2011 Dec 15;27(24):3392–8.

83. Joosten RP, Joosten K, Murshudov GN, Perrakis A. PDB_REDO: constructive validation, more than just looking for errors. Acta Crystallogr D Biol Crystallogr. 2012 Apr;68(Pt 4):484–96.

84. Cowtan K. The Buccaneer software for automated model building. 1. Tracing protein chains. Acta Crystallogr D Biol Crystallogr. 2006 Sep;62(Pt 9):1002–11.

85. McNicholas S, Croll T, Burnley T, Palmer CM, Hoh SW, Jenkins HT, et al. Automating tasks in protein structure determination with the clipper python module. Protein Sci. 2018 Jan;27(1):207–16.

86. Hintze BJ, Lewis SM, Richardson JS, Richardson DC. Molprobity's ultimate rotamer-library distributions for model validation. Proteins. 2016 Sep;84(9):1177–89.

87. Prisant MG, Williams CJ, Chen VB, Richardson JS, Richardson DC. New tools in MolProbity validation: CaBLAM for CryoEM backbone, UnDowser to rethink "waters," and NGL Viewer to recapture online 3D graphics. Protein Sci. 2020 Jan;29(1):315–29.

88. Chen VB, Davis IW, Richardson DC. KING (Kinemage, Next Generation): a versatile interactive molecular and scientific visualization program. Protein Sci. 2009 Nov;18(11):2403–9.

89. Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. Nat Methods. 2020 Mar;17(3):261–72.

90. Hubbard RE, Kamran Haider M. Hydrogen Bonds in Proteins: Role and Strength. Elsevier Oceanogr Ser [Internet]. 2010 Feb 15; Available from: https://doi.org/10.1002/9780470015902.a0003011.pub2

91. Fleming PJ, Rose GD. Do all backbone polar groups in proteins form hydrogen bonds? Protein Sci. 2005 Jul;14(7):1911–7.

92. Emsley P. coot [Internet]. Github; [cited 2020 Oct 13]. Available from: https://github.com/pemsley/coot

93. Bond P. coot_prune.py. University of York, Department of Chemistry; (Unpublished).

94. Padua D. FFTW. In: Padua D, editor. Encyclopedia of Parallel Computing. Boston, MA: Springer US; 2011. p. 671–671.

95. Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with NumPy. Nature. 2020 Sep;585(7825):357–62.

96. (jcsg) JCFSG, Joint Center for Structural Genomics (JCSG). Crystal structure of Flavoprotein (TM0755) from Thermotoga maritima at 1.80 A resolution [Internet]. 2004. Available from: http://dx.doi.org/10.2210/pdb1vme/pdb

97. Silvian L, Jin P, Carmillo P, Boriack-Sjodin PA, Pelletier C, Rushe M, et al. Artemin crystal structure reveals insights into heparan sulfate binding. Biochemistry. 2006 Jun 6;45(22):6801–12.

98. Bell JK, Botos I, Hall PR, Askins J, Shiloach J, Segal DM, et al. The molecular structure of the Toll-like receptor 3 ligand-binding domain. Proc Natl Acad Sci U S A. 2005 Aug 2;102(31):10976–80.

99. Agirre J, Iglesias-Fernández J, Rovira C, Davies GJ, Wilson KS, Cowtan KD. Privateer: software for the conformational validation of carbohydrate structures. Nat Struct Mol Biol. 2015 Nov;22(11):833–4.

100. Goddard TD, Huang CC, Meng EC, Pettersen EF, Couch GS, Morris JH, et al. UCSF ChimeraX: Meeting modern challenges in visualization and analysis [Internet]. Vol. 27, Protein Science. 2018. p. 14–25. Available from: http://dx.doi.org/10.1002/pro.3235