Enhanced real-time data analysis for Serial Femtosecond Crystallography

Stijn de Graaf

University of Cambridge, UK

September 6, 2017

Abstract

Serial Femtosecond Crystallography makes it possible to determine the structure of proteins, for which large crystals are hard to obtain, by combining the results from very many incomplete diffraction patterns. The extreme rate of data collection this requires, with up to thousands of detector readouts per second, introduces unique computational challenges. Two programs designed to deal with these challenges are OnDA, which provides real-time experimental feedback to users, and CrystFEL, which analyses the diffraction data, locating Bragg peaks and indexing them. Currently, however, the interface between the two is highly inefficient, with OnDA writing its output to files, which CrystFEL must then open and read. Here, we describe the implementation of a direct link between the two programs. With data serialised using MessagePack, and sent via ZMQ, this pipeline greatly reduces the time taken to obtain indexed peaks from experimental data.

Contents

1 Introduction

Serial Femtosecond Crystallography has provided biologists with a powerful tool for determining the structure of proteins.[1] Many biomolecules have proved tough to investigate using traditional crystallography methods, largely due to the difficulty of producing sufficiently large crystals.[2] Proteins are not naturally found in such large crystals - they are more likely to be found in solution - and forcing them into such arrangements typically alters their shape.[3] In fact, it can take several years to produce high quality crystals for particularly stubborn molecules such as ribosome.[4]

The resolution of the diffraction pattern obtained from a single crystal depends both on its size and the radiation dose it is exposed to. In order to obtain high-quality patterns from smaller crystals, we must correspondingly expose them to a greater X-ray dose. However, for protein crystals the radiation damage that this induces is highly problematic, smearing out the diffraction image and reducing the intensity of the Bragg peaks.[2]

Even when cooled to minimise the impact of the radiation, a typical protein crystal can only tolerate around 10MGy per angstrom of resolution before photoabsorption too greatly degrades the diffraction image.[5] Without large crystals, single crystal x-ray diffraction is therefore no longer a viable method.

Serial Femtosecond Crystallography (SFX) offers a solution to this problem. It involves taking diffraction images from a large number of small crystals, typically fired in a liquid jet across the path of the x-ray beam, and merging the results to obtain a single structure.[2] The pattern quality from each individual crystal is not sufficient to determine the structure of our protein, but when combined, they can provide us with a well-defined image. Furthermore, by making use of the femtosecond pulses available at X-ray free-electron lasers (XFELs), we are able to apply a very high radiation dose without compromising the quality of our data. The damage induced in the crystal begins over the course of around 100 femtoseconds, but our X-ray pulses are of the order of tens of femtoseconds in duration.[1] As a result, we are able to get a high resolution image before the onset of significant damage, in a technique dubbed "detection before destruction".

SFX provides unique computational challenges for experimentalists, in particular with regards to the extreme rates of data acquisition. For example, the European XFEL is expected to produce up to 27,000 pulses per second, with each pulse producing an image several megapixels in size.[13] Two software packages developed at CFEL to deal with these challenges are OnDA and CrystFEL.[6][7]

OnDA provides those running SFX investigations with fast online feedback on the progress of their experiments, displaying the "hit rate" (fraction of shots that contain a single microcrystal), saturation rate and a virtual powder pattern, in close to real-time. The feedback OnDA provides allows experimentalists to optimise their data quality as it is being recorded, saving precious beamtime.

CrystFEL is the world-leading software for analysing SFX diffraction data. In particular, it is able to find Bragg peaks in the diffraction data, and then index and integrate them.

Currently, however, the interaction between these two programs is highly inefficient. When OnDA has finished processing the experimental data, it then writes it to HDF5 files in the file system, which CrystFEL then reads as input. The process of reading and writing this enormous dataset is very slow, and the strain it places on the file system tends to affect the performance of each program.

The aim of this project was to streamline the path from data production at the beamline to a set of indexed diffraction peaks by connecting OnDA and CrystFEL directly, without the need for any intermediate files. Furthermore, it was hoped that such a connection might pave the way for a web interface for OnDA that would give users remote access to live experimental data.

2 Serialisation

The primary consideration when interfacing OnDA with CrystFEL was deciding on the serialisation - a protocol for converting raw data to a format that could be sent over the connection. OnDA provides the raw data for each event in the form of a Python dictionary, with key/value pairs as shown in Listing 1

Listing 1: Python dictionary to be sent by OnDA to CrystFEL

```
results_dict = {
    "timestamp":          time at which image was taken, in seconds since 1/1/1970
    " hit_flag ":         boolean stating whether image is a "hit"
    " detector_distance ": distance from sample to detector, in mm
    "beam_energy":        beam energy, in eV
    "max_num_peaks":      maximum number of peaks to return from peak-finder
    " corr_data":         NumPy array of the reading from each pixel in the
                          detector, as 32-bit floats
    " peak_list ":        a tuple of 3 lists containing the x-position, y-position
                          and intensity (respectively) of each peak found in the
                          image
}
```

The criteria for an ideal serialisation format were that it be:

Fast As discussed in the opening remarks, the connection between OnDA and CrystFEL has to handle a very large data throughput and so could become a bottleneck. As such, the faster we can transmit data, the more of it we can analyse while maintaining near real-time feedback. This will likely require us to compress the raw data, so that we can send more images for a given data transfer rate, while ensuring that doing so does not introduce a significant time penalty.

Portable While OnDA is developed in Python, CrystFEL is written entirely in C, so we would like to use a format that provides support in both languages. Ideally, it should also be able to serialise NumPy arrays, the data type in which OnDA to stores the raw pixel data. Given that we wish to eventually incorporate a web interface, support in Javascript would be an added benefit.

Readable Both OnDA and CrystFEL are open-source libraries, designed to be readily usable by the SFX community. As such, obscurity is to be avoided where possible. The meaning of the data should be as clear a possible in serialised and unserialised form, and should conform to existing standards where possible.

2.1 Crystallography-specific Serialisation

imgCIF is a file format defined by the IUCR specifically for use by crystallographers, combining key/value pairs of well-defined crystallographic parameters, with an image in binary format (either compressed or uncompressed).[8] imgCIF was originally considered as a serialisation on the basis it would be both readable and fast. Fast because it allowed us to compress the image data (the vast majority of the file size), and readable because the other parameters were retained in text format with clear accepted definitions of their meanings.

Unfortunately, many of the values in our results dictionary are not included in the imgCIF specification. While some have nearby equivalents (eg. diffrn_scan_frame.date for timestamp, pd_peak_intensity for peak intensity, and pd_proc_energy_incident for beam energy)[1], others such as hit_flag and max_num_peaks have no corresponding parameter. We would have to define these parameters ourselves, undermining the purpose of using a well-established format.

Moreover, formal support for imgCIF serialisation is unavailable in Python and Javascript, and is only provided in C through the closed-source CBFLib. Writing parsers for these languages ourselves was deemed too time-consuming

2.2 Binary Serialisation

In the absence of a usable crystallography-specific format, we moved on to considering a generic serialisation where we define the parameters ourselves. In particular, we considered msgpack and BSON, two well-established examples of a binary serialisation format.[9] By encoding our data as binary, as opposed to text, we would obtain optimal compression.

[1] I say "nearby" equivalents because we would be using them outside of the context they are really intended. For example, both pd_proc_energy_incident and pd_peak_intensity belong to pdCIF (the powder diffraction dictionary), and should only really be used in the context of powder diffraction experiments.

Both msgpack and BSON have well-maintained libraries in Python and C (as well as support for Javascript). However, while BSON has a package (bson-numpy) to convert single NumPy arrays to and from binary, there is no straightforward way to serialise a NumPy array contained *within* a Python dictionary, as we require (see Listing 1).[11] msgpack-numpy, on the other hand, is able to do so quickly, understandably, and in very few lines of code.[10] In fact, the Python code to serialise our results-dict is only 4 lines long:

Listing 2: Serialiser for the results_dict shown in Listing 1

```
import msgpack
import msgpack_numpy
msgpack_numpy.patch()
msgpack.packb( results_dict )
```

The difference between a NumPy array and say, a list of lists, is that the data type of every element is the same, and does not need to be specified entry-by-entry. This reduces the size of the transmitted data. All we require to recreate this array is the data type of its entries, the shape of the array (eg. 5 x 4), and the values of each entry, one after another, in binary.

What msgpack-numpy does is override the serialiser, so that when it encounters a NumPy array, it replaces it with a Python dictionary containing this information. The whole results_dict is then converted to binary as per the usual msgpack protocol. For example the following Python dictionary

Listing 3: Sample Python dictionary to be serialised using msgpack-numpy

```
my_dict = {
    "array" : numpy.random.random((3,3))        # random 3 x 3 NumPy array
}
```

is converted by msgpack-numpy to

```
my_dict = {
    "array" : {
        b"nd" :          True,
        b"type" :        "<f8",  # Entries are little -endian 8-byte floats
        b"kind" :        "",
        b"shape" :       [3, 3], # Array has 3 x 3 shape
        b"data" :        b"\xec\xfc\xc6,\x01 ... "        # Raw binary data
    }
}
```

Listing 4: Output of msgpack-numpy serialisation of dictionary in Listing 3, after being converted to a human-readable format (JSON).

The entries "nd" and "kind" are used to distinguish NumPy arrays from other NumPy objects that msgpack-numpy can serialise. We are only interested in arrays, and so can safely ignore these two parameters. In C, there is no equivalent NumPy deserialiser, so the readable presentation of this information is very helpful when we need to recreate the array in CrystFEL.

2.2.1 Speed comparison

We have established that msgpack is highly portable - with support for Python, C and Javascript - and, despite relying on us to define the parameters ourselves, quite readable with regards to how it handles arrays. Given that msgpack is a binary serialisation, we would expect data transmission to be relatively fast. However, we would like to formally establish this by comparing its speed to that of pickle, the standard Python-to-Python serialisation.

Pickle is used by OnDA to compress the information that it sends to the GUI (also written in Python). While not readily usable with in other languages, it is the standard method for serialising objects within Python and is a useful benchmark to compare our serialisation with.

	Time per message / s		
Size of Array	msgpack	pickle	Time difference
250 x 250	0.00030	0.00031	-2.8%
500 x 500	0.00187	0.00161	15.6%
1000 x 1000	0.00697	0.00572	21.9%
2000 x 2000	0.02543	0.01949	30.5%
4000 x 4000	0.22699	0.17848	27.2%

Table 1: Results of speed test comparing msgpack to pickle, the standard serialisation format within Python. A square numpy array of random 32-bit floats was serialised and sent from a server, written in Python, to a client, also written in Python, which deserialised the array. The time was measured from before serialisation to after deserialisation. The results shown here are the averages times per message from 3 separate runs of 100 messages each.

While msgpack is slower than pickle for larger array sizes, the difference is not great enough that we need to worry about substantially compromised performance. With this speed, the rate at which CrystFEL can process the data will continue to be the bottleneck.

On the grounds that it provides the best combination of speed, portability and readability, msgpack was therefore chosen as the serialisation format.

3 Data Transfer

The communication between OnDA and CrystFEL was done using ZeroMQ (ZMQ), a networking protocol also used within OnDA to transfer processed data to the graphical user interface (GUI).[12] It handles the transfer of messages between different processes, ensuring that they arrive in the right order, and that data is not lost.

ZMQ can be used over various transports, including TCP, allowing for communication between programs on the same computer or over a network. This is particularly useful for any potential future web application, as it will allow users around the world to connect to OnDA. Another benefit of ZMQ is that it is well-supported in many languages, including those relevant for our project (C, Python, and Javascript).[12]

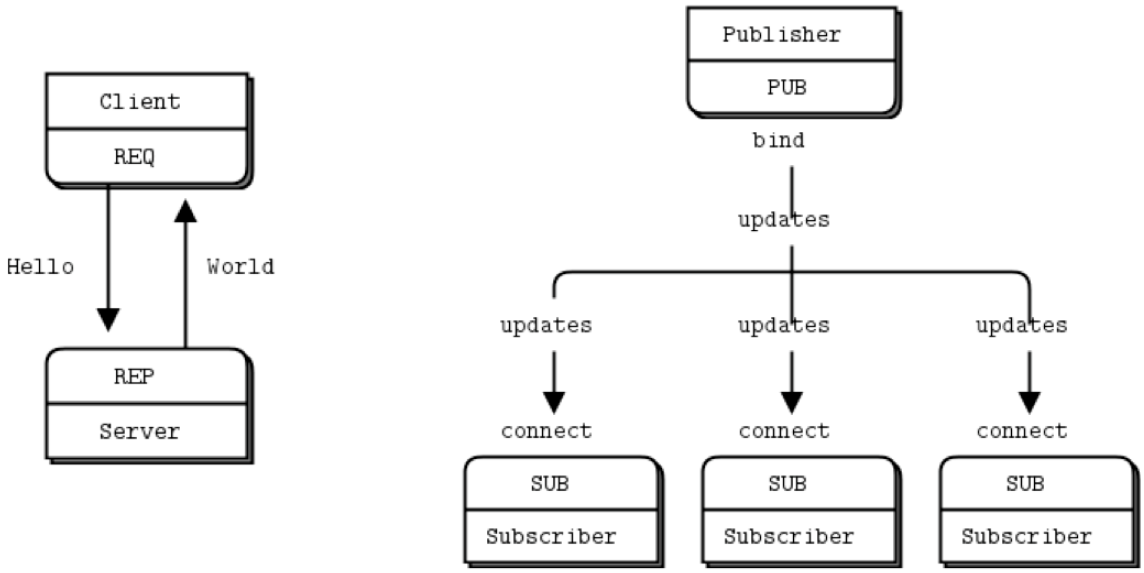

Figure 1: Request-reply and publish-subscribe ZMQ socket combinations. While the REQ-REP connection is two-way, the PUB-SUB connection only allows messages to pass from publisher to subscriber. Images reproduced with permission from ZMQ - The Guide, under cc-by-sa license.[12]

3.1 ZMQ sockets

ZMQ operates using a variety of network sockets. By determining the type of socket on each end of our connection, we can choose how these two endpoints interact.

The most simple combination, and the one used to link OnDA and CrystFEL, is a REQ (request) socket connected to a REP (reply) socket. The two operate in 'lockstep', with each sending a message in turn, unable to send another until it has received a response. This 'blocking' behaviour, waiting until CrystFEL is ready for more data

before OnDA sends it, ensures that all the images are successfully received. While Figure 1 only shows one request socket and one reply socket, ZMQ actually allows us to connect many request sockets to the same reply socket. This is ideal for our purpose, as it allows many CrystFEL worker nodes (operating in parallel) to retrieve data from the same OnDA master node.

The downside of REQ/REP's blocking behaviour is that any holdups in CrystFEL have a knock-on effect in OnDA, introducing a time delay. When we are interested in processing the complete dataset, this is necessary, but in the context of a remote OnDA viewer (the web browser application), where receiving up-to-date information is more important than receiving complete information, we can use a PUB/SUB (publish/subscribe) combination.

In this combination, the publisher does not wait for requests from its subscribers but simply broadcasts data whenever it is ready. Now, a slow client will not have a knock-on effect on the rest of the system, but will simply receive an incomplete set of data, missing any events it was not ready for.

4 Implementation

For our project, we wanted to implement a proof of concept that would be able to send some sample diffraction images from OnDA to CrystFEL, and then to index the peaks. Figure 2 demonstrates the intended flow of the interaction between OnDA and CrystFEL.

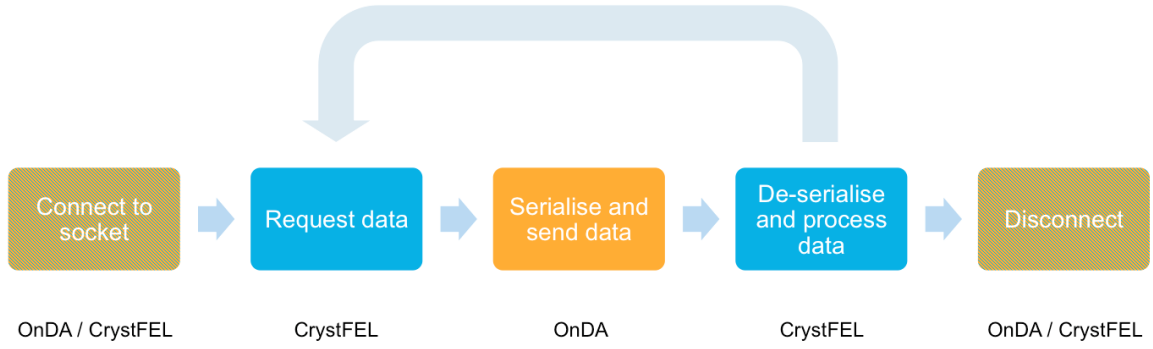

Figure 2: Overview of how OnDA is to interact with CrystFEL

4.1 OnDA

In OnDA the socket functionality is abstracted in the class ZMQOndaReplySocket, allowing the main program to send and receive data without needing to know the inner workings.

Listing 5: Class to handle ZMQ communication in OnDA

```
class ZMQOndaReplySocket:
    def __init__(self, reply_ip, reply_port):

        self._context = zmq.Context()

        # Indicates that we require a REPLY socket
        self._sock = self._context.socket(zmq.REP)

        # Bind socket to the address and port provided
        self._sock.bind("tcp://%s:%d" % (reply_ip, reply_port))

    def receive_request(self):
        return self._sock.recv()

    def send_data(self, message):
        self._sock.send(message)
```

When the main program initialises ZMQOndaReplySocket, it generates a reply socket and binds it to the IP address and port provided by the input. We can then call receive_request whenever we need to wait for a client to send a request, and send_data when we are ready to return them some data.

The rest of the work in OnDA is performed by the *master node*. The master node's job is to collect output data from a series of worker nodes, each processing the input in parallel, and sends it on to the next stage (in this case, CrystFEL). When the master node is created, it initialises the above socket, using an IP address and port provided in the configuration file. It then performs the function collect_data, every time a worker node produces more output.

Listing 6: Function executed by master node in OnDA

```
def collect_data(self, new):
    results_dict, _ = new       # Recover results_dict from worker
    self._num_events += 1       # Increment event counter

    # If event is a hit
    if results_dict['hit_flag'] is True:

        # Increment hit counter
        self._num_hits += 1

        # Serialise results_dict
        serialised_data = msgpack.packb(results_dict)
```

```
        # Wait for request on socket, then send data
        _ = self._sending_socket.receive_request()
        self.sending_socket.send_data(serialised_data)
```

For each event that is processed, one of the worker nodes sends the data in the form described in Listing 1. The only events CrystFEL is interested in are 'hits' (clear diffraction patterns from a single crystal), so the master checks whether this is true. If not, it moves on to the next incoming event from the workers. If, however, it *is* a hit, we then serialise our results_dict in msgpack format, await a request for data on the socket, and then transmit it.

4.2 CrystFEL

Before explaining what was implemented on the CrystFEL side, it is worth mentioning how the program typically operates. Normally, one provides the program with a list of filenames containing the raw data, and optionally the specific events within these files one wants to analyse. Using this list, CrystFEL fills a queue with all the events that need to be processed, indicating where in the memory the relevant data is stored. Then, it initialises a series of worker nodes which will process the data in parallel. Each of these takes an event from the front of the queue, reads the data from the indicated location, and stores it in an *image* struct. CrystFEL then performs the processing on *image* to index and integrate the peaks. By default, CrystFEL will also find the peaks within each image itself, although if the user specifies, these may be read directly from the files.

Our aim was to alter this code so that it accepts a stream of data coming from a network socket, instead of reading the data from a file. Thus, instead of receiving a list of filenames, CrystFEL now receives a file containing both the IP address and port on which to connect to the stream, as with OnDA. Replacing the previous fill-queue function, we now have an equivalent fill_socket_queue, which parses the contents of the file, and stores the output as an ip_plus _port struct, to which all the worker nodes have access. Then, when each worker is initialised, we use ZMQ to open a request socket which connects to this port. As discussed previously, ZMQ allows us to connect as many workers as we want to the OnDA master node.

Once the worker is connected, it enters a loop. First, it sends OnDA a request for data, then when it receives the binary data in reply, msgpack's C unpacker deserialises it and creates as its output a msgpack_object. obj_read replaces imagefile_read and performs the equivalent job, storing the detector readout in an image struct.

This process is not entirely straightforward, and is where the specific experimental setup becomes relevant. What OnDA provides is the complete output of the detector, pixel by pixel. However, detectors in SFX experiments are generally composed of several

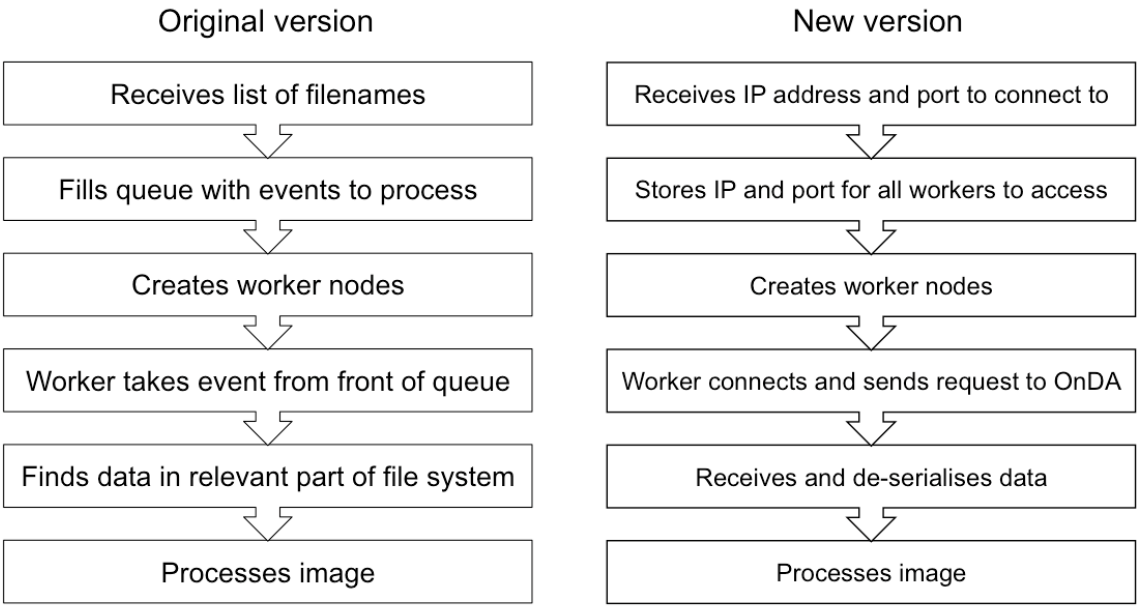

Figure 3: Comparison of how the original and modified versions of CrystFEL operate

different panels, with positions in space that can move relative to each other[2]. It is therefore necessary to provide CrystFEL with a geometry file that specifies these relative positions, and which pixel in the data array belongs to which panel. In order to account for this, pixel readings are stored panel-by-panel within image, not in one large dump. obj_read therefore makes use of the provided geometry file to assign the values from our reconstituted NumPy array to the corresponding pixel in the correct panel.

Since OnDA provides a list of peak intensities and positions in the results, one can also opt to read this data from the deserialised msgpack_object, using get_peaks_onda. Once this is complete, the rest of the processing happens as usual, with CrystFEL indexing the peaks within the image, and presenting the output to the user. Having processed the image, the program returns to the start of the loop and sends another request to OnDA for more data.

5 Outcome and future direction

The result of this project is a working pipeline that is able to run OnDA and CrystFEL simultaneously, without the need to write any intermediate data. By directly connecting the two programs, we are able to greatly reduce the time taken to get from experimental data to indexed diffraction peaks, while eliminating the strain on resources caused by reading and writing huge amounts of data to and from files.

[2]It is assumed that within each panel, the positions of pixels are fixed relative to each other.

While this implementation entirely removes the ability of CrystFEL to access data from a list of files, the next step will be to incorporate this modified version into the existing code, so that users have a choice - either they can accept data from a list of files, or over a ZMQ connection directly from OnDA. And in the more distant future, it is hoped that this proof of concept will provide the building blocks for a web-based remote viewer for OnDA, also utilising msgpack and ZMQ.

References

[1] Chapman, H. N., Fromme, P., Barty, A., White, T. A., Kirian, R. A., Aquila, A., ... Spence, J. C. H. (2011). Femtosecond X-ray protein nanocrystallography. *Nature, 470*(7332), 73-77.

[2] Stellato, F., Oberthur, D., Liang, M., Bean, R., Gati, C., Yefanov, O., ... Chapman, H. N. (2014). Room-temperature macromolecular serial crystallography using synchrotron radiation. *IUCrJ 1*, 204-212.

[3] Rossmann, M. G. (2014). *IUCrJ 1*, 84-86.

[4] Yonath, A. (2014). Hibernating Bears, Antibiotics and the Evolving Ribosome. *Nobel Lectures in Chemistry (2006-2010)*, 297.

[5] Howells, M. R., Beetz, T., Chapman, H. N., Cui, C., Holton, J. M., Jacobsen, C. J., ... Sayre, D. (2009). An assessment of the resolution limitation due to radiation-damage in x-ray diffraction microscopy. *Journal of electron spectroscopy and related phenomena, 170*(1), 4-12.

[6] Mariani, V., Morgan, A., Yoon, C. H., Lane, T. J., White, T. A., O'Grady, C., ... Chapman, H. N. (2016). OnDA: online data analysis and feedback for serial X-ray imaging. *Journal of applied crystallography, 49*(3), 1073-1080.

[7] White, T.A., Kirian, R. A., Martin, A. V. , Aquila, A., Nass, K., ... Chapman, H. N. (2012). CrystFEL: a software suite for snapshot serial crystallography. *J. Appl. Cryst. 45*, 335?341.

[8] Bernstein, H. J. and Ellis, P. J. (2005). *International Tables For Crystallography, Vol. G, Definition and Exchange of Crystallographic Data, edited by S. R. Hall and B. McMahon*, ch. 5.6, pp. 544?556. Heidelberg: Springer.

[9] Furuhashi, S. (2013) MessagePack specification. https://github.com/msgpack/msgpack/blob/master/spec.md

[10] Givon, E. L. msgpack-numpy. https://github.com/lebedov/msgpack-numpy

[11] bson-numpy. https://github.com/mongodb/bson-numpy

[12] Hintjens, P. *ZeroMQ - The Guide*. http://zguide.zeromq.org/page:all

[13] Altarelli, M. (2006). *XFEL: the European X-ray Free-Electron Laser: Technical Design Report*. DESY XFEL Project Group, Hamburg, Germany.